Zur Homepage der Dissertation

CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

An Environment for Compositional Specification Verification of Complex Embedded Systems

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

von

Dipl.-Inform. Hartmut Wittke

Gutachter:

Prof. Dr. Werner Damm
Prof. Dr. Martin Fränzle

Tag der Disputation: 18.11.2005

Zusammenfassung

Modellbasierte Entwurfsprozesse sind eine weitgehend akzeptierte Maßnahme zur Vermeidung folgenschwerer Fehler in der Entwicklung sicherheitskritischer eingebetteter Systeme. Modelle dienen in frühen Phasen der Entwicklung als ausführbare Spezifikationen und als abstrakte Implementierungen, anhand derer Anforderungen analysiert und Probleme identifiziert werden können. Durch den Einsatz von Model Checking kann der formale Nachweis vollautomatisch erbracht werden, dass ein Modell die geforderten Eigenschaften erfüllt.

Diese Arbeit stellt eine Verifikations-Umgebung für Modelle vor, die mit dem CASE-tool STATEMATE erstellt werden. STATEMATE Modelle können in zwei unterschiedlichen Simulations-Semantiken ausgeführt werden, einer Schritt- und einer sogenannten Super-Step-Semantik. In der gebräuchlicheren Super-Step-Semantik reagiert ein Modell auf externe Stimuli mit Sequenzen von Einzelschritten bis die Reaktion auf ein externes Ereignis abgeschlossen ist, wobei Simulationzeit nur zwischen stabilen Zuständen vergeht. Die Komponenten eines Super-Step-Modells reagieren dabei jedoch sequentiell in jedem Schritt auf modellinterne Änderungen.

Die Verifikations-Umgebung erlaubt die Anwendung von Model Checking für eine Reihe von Robustheits-Eigenschaften von Modellen als Push-Button-Technik, wie zum Beispiel der Erkennung konfliktierender Schreibzugriffe auf Variablen oder die Untersuchung von Nichtdeterminismus. Darüberhinaus bietet die Verifikations-Umgebung den Einsatz von Model Checking als formale Debugging-Technik an, um den Nachweis der (Nicht-)Erreichbarkeit von z.B. Zustandskombinationen oder graphischen Transitionen des Modells zu erbringen. Integriert in der Verifikations-Umgebung ist eine Bibliothek vordefinierter Spezifikations-Muster, die für den formalen Nachweis einfacher, benutzerdefinierter Spezifikationen instanziert werden können.

Entscheidend für die Anwendbarkeit formaler Verifikation in der Praxis ist die Formalisierung der Anforderungen. Den Mittelpunkt der Arbeit bildet daher eine Real-Zeit Erweiterung Symbolischer Zeit-Diagramme als intuitiver graphischen Spezifikations-Formalismus für Real-Zeit Eigenschaften von STATEMATE Modellen. Eine formale Semantik für die vorgestellte Variante Symbolischer Zeit-Diagramme durch Abwicklung zu Zeitbehafteten Symbolischen Automaten wird in der Arbeit definiert. Aus diesen Automaten werden Observer Module generiert. Durch Einhaltung einiger weniger Einschänkungen kann die Akzeptanz-Bedingung dieser Observer durch eine einfache Invariante ausgedrückt werden. Dadurch kann formale Verifikation durch einfache und effiziente Erreichbarkeits-Untersuchungen realisiert werden.

Die Anwendbarkeit von Model Checking auf System-Modelle ist in der Praxis durch die Komplexität der Modelle limitiert. Zur Reduktion der Verifikations-Komplexität können kompositionale Techniken verwandt werden, die es ermöglichen System-Anforderungen durch eine Menge von Teilbeweisen zu verifizieren. Basierend auf Symbolischen Zeit-Diagrammen wird ein kompositionaler Verifikationsansatz für STATEMATE Super-Step-Modelle vorgestellt, in dem die Gültigkeit einer System-Anforderung durch den Nachweis bewiesen werden kann, dass sie aus einer Menge von gültigen Komponenten-Anforderungen folgt. Die Gültigkeit der betrachteten Komponenten-Anforderungen kann dann auf der weniger komplexen Ebene der jeweiligen Komponenten verifiziert werden. Da bezüglich der Komposition von Komponenten eines Super-Step Modells sowohl Aussagen über Einzelschritte als auch über Zeit spezifizierbar sein müssen, unterstützt der vorgestellte Spezifikations-Formalismus sowohl die quantitative Erfassung von Einzelschritten als auch von Super-Steps.

Die Beweisführung und die Anwendung der Schlussregeln sind in dem vorgestellten Ansatz au-

tomatisiert. Durch ein Gültigkeits-Management werden die Auswirkungen von Änderungen an Modell und Spezifikationen auf Verifikationsergebnisse, unter Berücksichtigung hierarchischer Abhängigkeiten, automatisch verwaltet.

Abstract

Model-based development processes are a widely accepted measure to avoid errors in the development of safety-critical embedded systems. Models serve as executable specifications and abstract implementations in early phases of the development. Using Modeling, requirements can be analyzed and problems can be identified in these early phases. Application of model checking can yield the formal proof that a model fulfills the requirements.

This work presents a verification environment for models which are developed using the CASE-tool STATEMATE. STATEMATE models are executable w.r.t. two different time models, a step semantics and a so-called super-step semantics. According to the more usual super-step semantics, a model reacts on external stimuli with series of steps, unless the reaction is completed. Time is assumed to advance only between consecutive stable states. The series of steps, of which a super-step consists is assumed to take no time, even though the steps of the series are sequentially ordered.

The STATEMATE Verification Environment (STVE) offers application of model checking to establish a collection of robustness properties by push-button-techniques, such as detection of conflicting write accesses to variables or examination of non-determinism. Furthermore, the STVE supports the application of model checking as a formal debugging aid, e.g. for reachability checking of certain state configurations or particular graphical transitions. Integrated with the STVE is also a library of pre-defined specification patterns, which can be instantiated in order to formally proof simple user-defined requirement specifications.

In practice, the formalization of requirement specifications is a crucial point for the applicability of formal verification. Hence, a major concern of this work is a real-time extension of Symbolic Timing Diagrams as intuitive graphical specification formalism for real-time requirements regarding STATEMATE models. A formal semantics of this variant of Symbolic Timing Diagrams is defined by unwinding the diagrams into Timed Symbolic Automata. From these, observer modules for the application in formal verification can be generated. We show in this work that using only a few restrictions, the acceptance criteria of the observers can be reduced to simple invariants. Hence, formal verification can be applied by simple and efficient reachability-based model checking techniques.

Applicability of model checking is limited by the model complexity in practice. Compositional techniques can be applied in order to cope with the complexity. This way, system requirements can be established by a set of less complex proof-tasks for the components of a system. A compositional proof rule on the basis of Symbolic Timing Diagrams is presented in this work, allowing the verification of system requirements by conclusion from fulfilled sub-component requirements. Composition of sub-components of a super-step model requires the specification formalism to be capable of expressing both constraints w.r.t. steps as well as constraints w.r.t. time. Hence, the presented extension of Symbolic Timing Diagrams supports quantitative treatment of steps as well as quantitative treatment of super-steps.

Proof-task execution as well as management of proof-results are automated in the presented verification environment. Book-keeping of results and validity management take the effects of modifications of model and formal specifications into account, such that results are maintained w.r.t. hierarchical dependences.

Acknowledgements

I would like to thank the following persons who supported me throughout the preparation of this thesis. First of all, many thanks to Werner Damm for his supervision and for the very pleasant working environment throughout the years I had the pleasure to stay within his department. I have greatly benefitted from his encouragement and the creative working environment in the department. Many fruitful discussions about formal specification and verification technologies enabled me to finally write this thesis.

I furthermore thank Martin Fränzle for being so kind to act as second reviewer for my thesis.

An entire verification environment as presented in this thesis is not created by one person in isloation. A key ingredient for the realization of this thesis is the team of which I have been part during the last years. Many fruitful discussions and valuable ideas have contributed to the work presented in this thesis - and of course many powerful tools and libraries which have been developed by members of the team and which have been assembled in the verification environment.

I would like to thank the following colleagues (in no special order): Bernhard Josko for sharing his huge amount of knowledge about specification and verification techniques with me, Tom Bienmüller for the intensive and fruitful co-operation in the realization of the verification environment throughout the years. I thank Rainer Schlör for long years of instructive collaboration in the area of Symbolic Timing Diagrams, Ingo Schinz for his contribution to the realization of the proof-manager and Bertrand Gregoire for his initial implementation of observer generation for verification. Special thanks to Jochen Klose and to Boris Wirtz for proof reading parts of this thesis and for the instructive comments regarding my work. Thanks also to Udo Brockmeyer and Hans Jürgen Holberg for their deep insights into STATEMATE and development process related topics.

In a special way I would like to thank my parents Erika and Dieter Wittke for enabling my studies and always supporting me.

Finally, I like to thank my girlfriend Trude Lüppen; without her sympathy and support I would probably not have finished this thesis.

Contents

1 Introduction

1.1 Reactive Safety Critical Embedded Systems

Electronic control systems have become an essential part of our every day life. Digital electronic control devices are utilized in vehicle control, communication systems, industrial process automation, household appliances and in various other fields of application. In many cases, the user of a device does not even know of its control units. Such systems, which use a computer to perform a specific function, but are neither used nor perceived as a computer, are generally known as embedded systems.

In earlier times embedded control systems were mainly applied to situation analysis - e.g. speed-display - or situation assessment - e.g. warning systems. Enabled by the increase of efficiency of modern electronic controls, nowadays embedded systems take over also execution of actions (e.g. drive by wire) or even active control in many cases (e.g. autopilot). The more these applications take over active control the more they become safety or mission critical.

Sensors and actuators form the environment of these systems. Embedded systems are required to continuously interact with their environment at the speed of the environment. In contrast to normal programs, which are invoked with some parameters, prompt for inputs to be processed and terminate after their computations, embedded systems are expected not to terminate and to always react to external stimuli. This is why these systems are called reactive. Events to be noticed and processed by the embedded system may arise unpredictably at any state of operation. Quite often stringent timing constraints have to be satisfied for reactions.

Incorrect behavior or failures of safety critical systems may endanger human lifes or can lead to severe environmental pollution. For example, a failure of a car's brakes control system may have catastrophic consequences. Embedded systems are often used in such life critical application areas, where reliability and safety play an important role - and where the user does not perceive the requested function to be performed by a computer.

The development of embedded control systems takes a growing time portion of the overall development time and is a price factor of increasing importance in safety critical application domains. For example, up to 80 embedded controllers are utilized in a high class automobile and about 25-30% of the overall development costs are spent for their development [Bec99]. Among them are ABS (Automatic Brakes System), ESP (Electronic Stabilization Program), Central Locking Systems. Many embedded control systems - as for example a Central Locking System - seem not to be safety critical at first glance. But, in order not to endanger the inmates of the car it must be guaranteed that the system releases the door locks in case of an accident under all circumstances. Analogical important requirements have to be fulfilled in many application areas for embedded control systems.

Independent of the amount of embedded control systems also the complexity of their functionality still increases. While earlier applications, such as a speed-display, had a rather straightforward realization, modern embedded control systems often control a wide range of functions, as for example

an autopilot.

Meeting the safety requirements is a major aspect in the development process of complex reactive safety critical embedded systems. It depends on the criticality of the system under development (SUD) how much effort needs to be spent for the validation and verification of safety critical properties during the development process. Establishing evidence for meeting the requirements not only for the final product, but also for each development activity, avoids deep iteration cycles or, worst case, a complete failure of the development.

Taking these risks into account, many development process regulations elaborate on strategies aimed at avoidance or at least earlier detection of development flaws.

1.2 Model Based Development

The increasing complexity of computer-controlled devices, combined with tightening development deadlines, force the developers of embedded system products to change the development process. Concept-to-code solutions are increasingly used to integrate the analysis, design, implementation and testing phases of embedded systems design. Modeling instead of coding is an attractive alternative to conventional development processes, in which design errors and flaws can often only be revealed by intensive testing at the end of development. In particular, inconsistencies in only textual specifications are a severe source of errors, which can become very expensive or lead to a complete failure of development. By enabling programmers to build applications in an iterative manner, linking every facet of the design process, product behavior can be completely validated up front, before anything is built.

Modeling techniques are used to capture and structure requirements of a system. The approaches vary in detail between different projects and different CASE tools employed in the projects. However there are some common principles. Their goal is to generate a well organized description of the system to be built. System modeling approaches use executable models containing additional information within the model structure, and simulation techniques for the presentation and analysis of concepts.

Embedded designs can be optimized using such models in a variety of ways. The interaction of the system with its environment can be captured using graphical formalisms. Functional decomposition into sub-components and their communication can be modeled in an iterative manner. Critical functionality can be explored in a depth first analysis, first modeling the critical parts of the system, while non-critical parts are left as abstract as possible. Requirements can be captured at an appropriate abstract level and analyzed using simulation capabilities of the respective CASE-tool. The overall system architecture can be derived in an iterative process exploring a number of design alternatives.

STATEMATE, distributed by I-Logix, Inc., USA, is a graphical design, simulation and prototyping tool for the rapid development of complex embedded systems. Requirements and specifications are captured using graphical formalisms. Models can be simulated, allowing the developer to discover errors in the requirements and specifications early in the process when they are inexpensive to correct. The simulation tool is also likely to be used by the software engineers as a simple debugging tool. It supports the users by allowing detailed control over the execution and current status of the model. Graphic panels are often used at this stage to help understand the overall execution of a model.

> "Using this graphical language, engineers can analyze and simulate the graphical model

> to show customers how a system is expected to behave and to actually test that it functions correctly. This validation of system definition and operation identifies defects early in the design process, eliminating problems before they become expensive, difficult or even impossible to detect." [LIS02]

In general, executable specifications provide the basis for model validation through animated simulation. The system under development is functionally validated with reference to the predefined requirements. The dynamic interaction of system components is simulated for virtual system integration testing. Such validation phases lead to the production of iterative prototypes. They can be employed to evolve the model in the design phase toward a definition of its software-architecture, while automatic code-generation tools can speed up the implementation phase. Test-vectors generated from executions of the model can be kept for application with later software products of the design process.

The models are used by engineers to communicate the behavior of the model to customers and domain experts, and to discuss whether the modeled behavior is an appropriate solution for certain requirements. This communication with the company's external suppliers or customers allows the engineers to "test drive" specifications before writing any software, thereby eliminating ambiguities common with textual specifications, resulting in a higher product quality.

STATEMATE's integrated tool for checking consistency is used by the engineers to expose badly formed models before time is wasted in simulation. But STATEMATE models may show other undesired effects which can not be detected by static analysis performed by the check tool, because detecting these effects requires complex dynamic analyses. Among these modeling artefacts are, for example, conflicting transition-triggers and hazardous data accesses. STATEMATE explicitly supports non-deterministic choice of conflicting graphical transitions. While non-determinism might be acceptable in an abstract modeling approach, a reference model for later phases of system development should not contain such non-determinism, neither between graphical transitions nor between data accesses in concurrent parts of the model.

1.3 Formal Verification

In order to ensure correctness w.r.t. such undesired artefacts formal verification techniques can be employed. Formal verification aims at providing a mathematical proof of correctness w.r.t. a requirement specification for all possible sequences of input stimuli. Within the field of formal verification, there are two major approaches: theorem proving and model checking. A theorem prover supports the user by offering and applying tactics and rules to a proof-obligation in order to establish a mathematical proof. While theorem proving is in general an interactive technique, model checking is a fully automatic approach. A model checker determines if the model satisfies a given requirement under all circumstances. In case of a violation, a witness is provided by the model checker, showing the sequence of input stimuli and reactions of the model which led to the violation of the requirement specification.

Even though model checking is a fully automatic verification procedure, it requires a formal model as input as well as an unambiguous formal specification, usually written in a textual temporal logic formula. In general, formalization of requirements is difficult and requires expert knowledge. It is thus instrumental to tune handling of formal verification tools to easy-to-use applications, which can be utilized by engineers, requiring as little expert knowledge as possible.

The STATEMATE verification environment (STVE) presented in this work offers *robustness analyses* for transition non-determinism, read/write and write/write hazards as well as for range-violations in variable assignments. For each kind of conflict a specific analysis can be invoked. As result of a structural model analysis, potential conflicts are offered to the user for selection. For each selection, the verification tools searches the entire state space of the model for a computation sequence reaching a situation in which the conflict is observed. If any such computation sequence exists, a *simulation control program* is created which can be loaded and executed with the STATEMATE simulator to drive a simulation to the particular situation in the model. Alternatively the computation sequence can be visualized as a diagram showing the valuations of all data-items step-by-step. Since the verification environment is tightly integrated with the STATEMATE tool, such analyses can be performed by developers already during development. Application of the offered robustness checks and elimination of undesired artefacts increases the quality of the model, yielding a reference model for the development process.

Different 'formal debugging' techniques are offered by the verification framework for dynamic exploration of the model under construction. Developers can perform checks in order to search for computations that take selected transitions. Checks are offered to determine whether single selected basic states or combinations of states are reachable. Most generally, developers can check whether there exist computations reaching configurations, which are specified by arbitrary user defined expressions. Again, witnesses for such computations can be simulated with the STATEMATE simulator. Thus, developers are supported in detecting dead code or debugging their models using fully automated formal techniques. Robustness checks and debugging techniques are applicable at every decomposition level of models, allowing developers to focus on sub-components even if models have grown to a complexity which is not easy to handle by formal methods.

Beyond the robustness and debugging checks the verification environment offers an easy to use verification support for the verification of functional requirements. In order to formalize requirements the developer can instantiate specifications from a set of predefined specification patterns. For example, a pattern for 'Q happens at most X steps after occurrence of P' can be selected, and model specific expressions be mapped to the parameters Q,P and X. The set of predefined patterns covers a wide range of often used specification schemes. Patterns can be instantiated as commitments as well as assumptions about the environment of the model. Critical functional requirements such as safety, timing or mutual exclusion properties can be verified during the development iteration cycle, without requiring expert knowledge. Of course, pattern-based assumptions can also be used for 'formal debugging' and in robustness analyses.

Robustness analyses, formal debugging as well as pattern-based verification are optimized in the STVE for debugging and on-the-fly verification of less complicated properties, hiding away the details of the underlying mathematical machinery. All techniques are integrated with a graphical user interface, that allows to apply the various checks by a simple push-button handling.

For the verification of more complex system specific properties, a graphical specification formalism is offered by the verification environment. Symbolic Timing Diagrams (STDx) is a visual specification language allowing the user to specify temporal requirements in an intuitive graphical formalism. Symbolic waveforms are used to specify sequences of valuations for variables of a model. Temporal relations of events, i.e. changes of valuations, can be specified by drawing arrows between them. These arrows can be annotated with qualitative or quantitative timing constraints. Diagrams can be used to specify required behavior (commitments) as well as to express assumptions about the environment. From the individual diagrams, specifications can be built by grouping commitment diagrams with assumption diagrams. We feel that using a graphical specification formalism is

more intuitive than formalizing specifications in temporal logic formulae. Like the other techniques, specification verification using STDx is integrated with a graphical user interface that hides away the details of controlling the formal verification tools.

A proof-manager offers proof obligation construction and execution of verification tasks at every decomposition level of the model. The proof-manager keeps track of the already verified requirements and supports compositional verification and hierarchical reasoning with respect to the system structure. Validity of a top-level requirement of a system can be conluded from verified requirements of its components. Thus, properties of distributed protocols can be concluded from verification results for the contribution of sub-components involved in the protocol.

1.4 Organization of this Thesis

Chapters 2,3,4 and 5 describe fundamentals. Chapter 2 gives an overview of some prominent development process regulations and highlights the role of formal models in the development of safety-critical embedded systems. In particular the relationship between model based development and the German V-model standard [ESt97] and the placement of formal verification in this relationship are discussed.

Basic features and capabilities of the CASE-tool STATEMATE are described in chapter 3, aspects regarding the execution of models are discussed in section 3.2. STATEMATE supports two different time models for simulation, a step-oriented time model and a so-called asynchronous execution semantics, in which a model reacts to stimuli with series of steps unless it stabilizes and no further reaction is possible without new external stimuli. Simulation time only passes between consecutive stabilizations, even though internal reactions adhere strictly to the sequentiality of steps.

Section 3.3 explains the case-study of a radio-based signaling system, which serves as running example throughout this thesis. This case-study models the interaction of a train with level crossing via a radio link. The case-study will also be referred to in chapter 8, where experiences and results of regarding application of the presented verification techniques are discussed.

An overview of automatic formal verification techniques is provided by chapter 4.

In chapter 5, the representation of STATEMATE models for verification is described. The chapter starts with short description of the compositional semantics of STATEMATE models as presented by Damm et al. in [DJHP97]. The description aims at summarizing the compositional semantics as far as possible w.r.t. presentation of the concepts of compositional verification (presented in chapter 7). In section 5.3, we describe the interpretation of time, which forms the basis of our treatment of time in the verification of models of embedded systems. We focus on the asynchronous execution semantics, for which steps as well as simulation time have to be regarded. Section 5.4 explains (1) the representation of the decomposed view to the behavioral descriptions of sub-components, and (2) the management of specification views to these sub-components.

Requirement capturing for embedded systems is the focus of chapter 6. The chapter contains a description of *robustness analyses* techniques as well as a description of *formal debugging* techniques. As a common basis of pattern verification as well as for verification using Symbolic Timing Diagrams (STDx) specifications, section 6.3 presents a formalization of Timed Symbolic Automata (TSA). Basically, TSA are automata on infinite words which are not only triggered by observation of particular events, but take also timing aspects into account. In particular, TSA are capable of quantitative treatment of both steps and simulation time for the asynchronous execution semantics of STATEMATE models. We show in this section that for a particular sub-class of TSA, the acceptance

criterion reduces to a simple invariant. The section provides also a description of observer module generation from TSA for application in verification. Observers are specification automata, which are combined with the model to be verified in a parallel composition. When executing the combined model, the observer tracks the computations of the model and judges whether the observed behavior complies with the specification. A specialized variant of TSA - pre-defined specification pattern - is briefly presented in section 6.4. Chapter 6 concludes with the presentation of a formal TSA-semantics for an extended variant of STDx, which is capable of quantitative specifications w.r.t. both steps and simulation time according to STATEMATE's asynchronous execution semantics.

In chapter 7, we overview the optimizations and abstractions, which can be applied within the verification environment. In section 7.3 rules for compositional verification using observers - obtained from TSA - are presented. Section 7.4 describes the integration of compositional verification with the STATEMATE verification environment. Dependencies between tautology proofs and the proofs for sub-component specifications contributing to compositional proof are managed by a proof-manager. A proof-manager is presented, which keeps track of the validity of proofs w.r.t. changes applied to the model or to the specifications. Reasoning rules for invalidation or revalidation of proofs according to the proof hierarchy and to changes in sub-component models or specifications are captured by a proof-graph.

Experiences and results regarding the application of the presented verification techniques are documented in 8. Chapter 9 concludes this work with a summary and identification of directions for future work.

2 Development Process

Several widely accepted standards and quasi-standards were established by national and international organizations regulating the development process for safety critical systems in the different application domains. Among them are the German V-Model [ESt97] by the "Bundesamt für Wehrbeschaffung", the CENELEC EN 50126,EN 50128 and EN 50129 [fES97] by the "Comite Europeen de Normalisation Electrotechnique", DO-178B [RTC92] by the "Radio Technical Commission for Aeronautics" (RTCA), ISO 9001 and ISO 9126 by the "International Standardization Organization".

DO-178B

RTCA is a private, non-profit organization that addresses requirements and technical concepts for aviation. The products of RTCA are recommended standards and guidance documents that focus on the application of electronics technology to implement new or modified concepts and to satisfy related requirements. The DO-178B standard and its predecessor DO-178A form the basis for the certification of many on-board systems involving software, especially in non-military projects [BPR98].

The standard defines software levels according to so-called "failure condition categories":

A. Catastrophic
B. Hazardous/Severe-Major
C. Major
D. Minor
E. No Effect

The software levels are based on the "contribution of software to potential failure conditions as determined by the system safety assessment process. The software level implies that the level of effort required to show compliance with certification requirements varies with the failure condition category" [RTC92]. The main focus of DO-178B is the regulation of the quality assessment process with respect to the software levels. On the basis of a risk evaluation an adequate validation and verification strategy is determined. If one component fulfills a very critical task this component is to be tested most thoroughly ("hardest/most critical first"). [fEA01, 22]

DO-178B identifies and describes process objectives for the different phases of the overall process and provides guidelines w.r.t. to the different software levels. These guidelines are in the form of:

- Objectives for software life-cycle processes.
- Descriptions of activities and design considerations for achieving those objectives.
- Descriptions of the evidence that indicate that the objectives have been satisfied.

The standard recommends various activities already in the early phases of software planning and development. In order to ensure that safety related requirements are properly implemented throughout the software life-cycle, already the sofware planning "should choose the methods and tools to achieve the error avoidance or detection necessary to satisfy the system safety objectives" [RTC92].

The standard requires that each process produces evidence that its outputs can be traced to their activity and inputs, showing the degree of independence of the activity, the environment, and the methods to be used.

Moreover, the standard discusses the characteristics, form, configuration management controls, and the content of the software life-cycle *data*.

The characteristics of the software life-cycle data are:

Unambiguous: Information is unambiguous if it is written in terms which only allow a single interpretation, aided if necessary by a definition.

Complete: Information is complete when it includes necessary, relevant requirements and/or descriptive material, responses are defined for the range of valid input data, figures used are labeled, and terms and units of measure are defined.

Verifiable: Information is verifiable if it can be checked for correctness by a person or tool.

Consistent: Information is consistent if there are no conflicts within it.

Modifiable: Information is modifiable if it is structured and has a style such that changes can be made completely, consistently, and correctly while retaining the structure.

Traceable: Information is traceable if the origin of its components can be determined.

EN5012x

The CENELEC EN5012x standards focus on the field of railway applications. In particular EN50126 regulates the specification and demonstration of Reliability, Availability, Maintainability and Safety, and hence addresses system issues on the widest scale. EN50129 mainly regards the approval of electronic systems in railway application, and hence has a focus on physical safety-related aspects of electronic systems. The third part in the group of related standards EN50128 regulates the software development process for railway applications:

> "The key concept of this European Norm is that of levels of software safety integrity. The more dangerous the consequences of a software failure, the higher the software safety integrity level will be.
>
> This European Standard has identified techniques and measures for 5 levels of software safety integrity where 0 is the minimum level and 4 the highest level. Four of these levels, 1 to 4, refer to safety-related software, whilst level 0 refers to non safety-related software." [fES97, 4]

The standard reflects the increasing need for advanced validation techniques in the development of train system applications, which increasingly involve both complex and safety critical control units. Besides quality assessment requirements the standard also regulates the development process itself. It highly recommends to develop applications with safety critical integrity levels using formal methods, in particular supporting various forms of analysis to check for different correctness properties [fES97, 50128 Annex A].

The software development life-cycle described by standard EN50128 is similar to the German V-Model, which will be considered in the following.

2.1 Model Based Development Process

The construction of reliable embedded systems can be significantly improved using a model-based and tool supported development process. Research results suggest that such a process contributes significantly to increase the quality of the developed product as well as to better efficiency of the development itself. In [Jon91] it has been shown that:

- more than 50 % of serious errors are made during design (25 % during implementation); about 30 % of medium class errors are made during design (30 % during implementation)
- analytical techniques performed on early-phase description of the product (e.g., structured approaches, design reviews) require generally at least less than 50 % of the effort in both error detection and correction needed for later-phase techniques (e.g., integration test, field test)
- those analytical techniques of the early phases are at least twice as effective to detect errors of the early phases than those later-phase techniques [SRS+03].

Modern CASE-tools (Computer Aided Software Engineering) allow the developer to easily create models representing requirements of the system. Modeling can be very early applied in the development process in order to evaluate conceptual aspects:

Hanxleden et al. [vHBK98] describe the model based development of a bus based airbag system:

> "The starting point is a concept of the product, which may include a set of requirements, market studies, rudimentary algorithms, legacy systems to build on, or hardware choices. The concept will exist in a variety of forms, such as verbal specifications, presentation materials, software, or only people's ideas and imaginations. Once the concept has reached a certain level of detail-enough that people can draw state transition diagrams to explain certain behaviors-it should be translated into a system model. This model contains the system's hardware and software components, such as the central control unit and peripheral units in the air bag example. It also contains a model of the environment, such as energy sources, sensors and actuators, or connecting buses. The model should be phrased in a precise, unambiguous syntax. "

During the different phases of the development process, different aspects of the system under development are addressed (for the domain of embedded systems, e.g., overall functionality, time and resource limitation, partitioning and deployment, scheduling). Specific models can be built for the different phases of the development process. Since models of embedded systems, often develop from monolithic single - functionality models to distributed, interactive multi-functionality networks, a central aspect of these models is treatment of interaction and communication as well as time-related aspects.

Modeling can be applied - in terms of the V-Model - in the system requirement analysis phase, in system design, for HW/SW-requirements analysis, during the architectural and partly in the detailed design phase. (Formal) verification techniques can be applied to verify that a formal specification of one phase is a refinement of the preceding phase.

The models can be handed to the customer and validated using simulation. The basis of communication between development teams is not only a textual requirement definition but also an executable (graphical) specification formalizing the requirements.

Furthermore, models must support specific aspects needed for the application domain. By supporting domain and application specific modeling elements, model information about the system under development becomes available, leading to stronger analysis and generation techniques (e.g, in the domain of embedded systems, checking the worst case time bounds of a task, or generating a bus schedule from the communication description of component models).

Models should contain only those aspects needed to support the development phase they are applied to, and this way reduce the complexity of descriptions as much as possible. Furthermore modeling helps to avoid producing faulty or inconsistent descriptions [SRS+03].

Regarding the focus on certain view points of the system under construction a distinction into conceptual and physical models can be made:

conceptual models are used to describe functional or behavioral concepts in an abstract manner, for example using statecharts. They focus on what a system should do and how the system or parts of it should behave. Identified sub-functions are modeled as components and relations and interactions between these parts are described. In essence, conceptual models do not expose implementation details. Nonetheless, they can be used to explore many interesting aspects of the overall behavior, such as safety or causality of in and output behaviors, making analysis and generation support available even at early stages of the development process. Conceptual models are typically built in requirement analysis and overall system design phases.

physical models consider implementation details, such as for example hardware/software partitioning, physical timing, communication bandwidth or power-consumption of hardware parts. Physical models are employed mainly in later phases of the development process, when concrete implementation details need to be considered.

One major benefit of a model based development process is, that integration of the overall system can be virtually applied to the system model in the decomposition phases. The proper interaction of sub-systems can be validated using virtual integration. System-level requirements can be verified using compositional verification techniques. Virtual integration can avoid deep iteration cycles and thus help to reduce overall development costs. "Numerous studies have shown that correcting an error during integration costs over 10 to 1000 times more than correcting it at specification time" [iL00].

"Virtual prototyping will not offset classical emulation or prototyping techniques; these still have their place in the development cycle, but rather complement them as an early equivalent in the design cycle and promote an early start of the software development cycle, reducing product risk and *Time To Market*. It can be successfully employed as first verification point of the hardware-software integration and the system as a whole" [Tho02].

EN 50128 highly recommends modeling and formal techniques for the development of system with higher safety integrity levels [fES97, Annex 1].

Without a model based development process, embedded software is often still coded by hand following textual descriptions of the desired product attributes. In this case, the product is validated at a very late stage in the design process, often resulting in expensive post-implementation surprises, as there are performance and safety obstacles and delays in release schedules. Software testing may take 30 percent to 60 percent of development time [DC01].

2.2 The V-Model

The standard V-Model (EStdIt - General Directive 250/251/252) by the "Bundesamt für Wehrbeschaffung" forms a guideline, how to develop software compliant to the ISO 900x family of standards (especially ISO 9001). Therefore, the V-Model describes the development process from a functional point of view. It does not describe any special organizational models because it shall be used in different organizations and companies. The V-Model is a generic process description, which has to be adapted to project-specific processes. The standard regulates all activities and products and the logical interdependencies between activities and products during the development process. The development process is improved by establishing quality assurance activities and allocating methods to steps in the design process. Identification of typical phases and intermediate products of the process form the basis of traceability which is seen as a major pre-requisite for process improvement.

We will not discuss similarities and differences of the mentioned standards in detail. The generic V-Model has the benefit of being applicable to almost all application areas and being intuitively understandable. Because of its process-based view point, we will consider the V-Model in more detail. The CENELEC standard refines the generic process description of the V-Model with respect to the Safety Integrity Levels[1]. For the development of safety critical embedded systems a combination of the process-based view of the V-Model with the strictness of the CENELEC standards seems to be the best fit.

The V-Model identifies four roles in the development process. Each of this roles has its clearly defined responsibilities:

Project Management. The project management is responsible for the overall project-planning, definition of the software development environment, initialization of the other sub-models, definition of milestones, e.t.c.

Software Development. Software developers are not only responsible for developing the pure software parts of the model. Quality assessment and assurance are part of the development process. Thus, software developers are responsible for producing process data of various kinds (e.g. documentation of software and process, quality assessment documents as well as test cases for later use etc.).

Configuration Management. Configuration management is in charge of providing the development tools and environment. Version control for all products is a key responsibility of the configuration management.

Quality Assurance. The role of quality assurance is very important in the V-Model. Quality assurance has not only to assess the quality of products, but also to define requirements and constructive measures in advance as inputs to the sub-model of software development.

The collaboration of these four roles is described in detail for each activity of the development process in [ESt97].

Product quality can be assured by applying *analytical* quality assurance measures as well as by applying *constructive* quality measures. Quality assurance defines the quality requirements according to the plan data defined by the project management. If the quality assurance measures are constructive, as for example the use of specific design tools or certified compilers, they take place in sub-model Software Development. Analytical measures can also be applied already in early phases during development. Typical examples are resource monitoring systems, simulation and debugging. Software developers and configuration management have to agree on a specific configuration

[1] without directly referring to it

structure, always reflecting the actual state of the development progress. Version control as well as change management are under control of configuration management, which provides - according to milestones - certain configurations of the product to quality assurance for quality assessment.

Normally, system development is handled in upgrades. An entire System is planned, but only realized in parts while the functionality grows continually. A first system version is made available (e.g. handed to the user) as soon as possible; this system version should include the basic functionality already meeting the basic quality requirements. Later versions of the system essentially expand this functionality.

This step-by-step approach is referred to as "incremental development". Within this incremental process products may take on the following states:

"planned" : The product is being planned. This is the initial state of all products.

"b.proc." : The product is being processed. It is either in the "private" development area of the developer or under control of the developer in the product library

"submitted" : From the developer's point of view the product is completed and subject to configuration management. It is subjected to quality assurance assessment. In case of the product being rejected by quality assurance, it is returned to the state "b.proc." or else it is considered as "accepted". Beginning with state "submitted", the developer can only change the product by updating its version number.

"accepted" : The product has being checked and released by quality assurance; it can only be modified if the version number is updated.

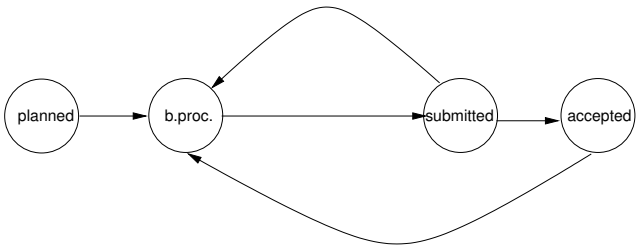


Figure 2.1: V-Model: States of a Product

The minimal state transitions of a product are shown in figure 2.1

The life-cycle Model

The standard is named after its life-cycle model, which looks like a large V. Figure 2.2 shows a rough sketch of the life-cycle model. The life-cycle model can be iteratively applied, either traversing all phases iteratively or looping in only a subset of the defined phases until the prerequisites for the following phase is fulfilled.

A project starts at the upper left position in the left leg of the V with *system requirement analysis*. In this phase the overall (and user-level) requirements are collected and analyzed. Technical, organizational and other conditions of the process itself, as well as quality requirements regarding

the product are defined. The system is structured from a user-level point of view and a detailed threat and risk analysis takes place in this phase. The central functionality must be clearly defined and described. Depending on the criticality of the system, quality criteria to be met by the process are defined. The results of this phase form the input for the following *system design phase.*

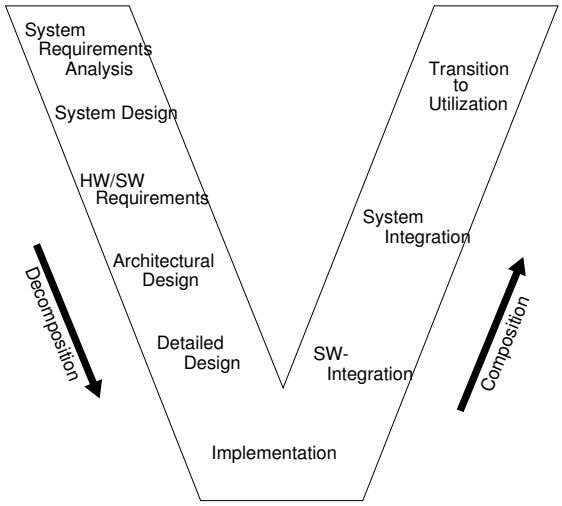


Figure 2.2: V-Model: schematic life-cycle model

Based on the requirements stipulated during system requirement analysis, a solution proposal is generated for a possible technical system structure. This proposal is evaluated and refined and technical requirements are derived from the requirements identified and defined in the previous phase. The proposed system architecture defines interfaces of the system to the environment and between proposed sub-systems. Also in this phase the specification of the system integration plan is set up, which describes interfaces and functionality of the sub-systems, defines quality criteria and allocates resources. The system architecture is assessed regarding the criticality of the system and its components. Safety integrity levels are assigned to critical components and to the system. The standard recommends to keep the number of interfaces between critical parts and non-critical parts of the system as small as possible. The feasibility of the proposed system architecture is investigated.

An allocation of the requirements defined in the *system requirement analysis phase* to elements of the proposed system architecture is performed. The standard demands that the allocation of requirements and marginal conditions to the elements of the technical architecture must fulfill the following criteria:

- Every requirement must be allocated to at least one element of the technical architecture, ideally exactly to one element.
- Each requirement is allocated to the lowest element in the refinement levels which makes it possible to meet the requirement completely. Normally, the total of the requirements have to be allocated to various refinement levels.
- Provided that a requirement is of general importance to elements, it must be considered within the scope of the allocation which individual architecture elements this requirement really has to fulfill.

- The allocation must be realized in such a manner that it will be possible to prove the fulfillment of the requirement by checking the corresponding architecture element [ESt97, 4-18].

The results of the *system design* phase are inputs to the next phase, the *HW/SW requirements analysis.* Dependent on the concrete requirements of the project, a partitioning into hardware and software components of the system is determined. Hardware parts of the system are not considered in the V-Model in detail. Technical requirements for the hardware and software parts of the system are derived from the requirements established in the system design phase and from constraints with respect to the partitioning.

Next follows the *architectural design* phase. For the software parts of the system, the decomposition into components, processes, modules, and databases is performed. Performance and security considerations play an important role in this phase. Internal and external software interface descriptions and the integration plan are refined according to the requirements identified in this phase.

Before starting the implementation, the process now enters the *detailed design* phase. The operational information is updated with design related details. The software components are specified down to the programming specification level, with regard to their environments, their realization, data handling, error- and exception handling, etc. Concrete resources and timing requirements are analyzed and determined. Again the integration plan is refined and quality criteria are defined to be met when putting the components together later in the process.

With respect to the derived specifications and requirements now the software components are implemented.

After the *implementation phase,* the software components have to be collected and to be put together to modules following the integration plan as defined in the *architectural* and *detailed design* phases.

Quality assessment as defined and prepared in the decomposition phases is performed during the *software integration* phase. The integration strategy can follow numerous different strategies: bottom up or top down or even a sandwich strategy. Especially for safety critical systems, it might be useful to early integrate critical and important functions. Missing parts can be represented by dummy implementations. Substructures can be iteratively replaced by improved versions. Simulated or emulated parts can be used instead of concrete implementations. The integration strategy should be defined already in the decomposition leg of the V.

At a certain state of module integration the *system integration* phase can be started. Again this phase can follow different strategies, depending on the integration plan defined in the corresponding decomposition phases.

With the first integrated version of the overall system the *transition to utilization* phase can be entered. It depends on the concrete project how such an possibly incomplete prototype can be used in a simulated or real environment.

While ascending the composition leg of the V, the system under development is assessed with respect to the integration plan and the criteria defined in the decomposition leg of the V. As described before, when moving down the V a representation is generated according to the requirements derived in the phase before. Cross-Checks (tests) are defined for later use in the integration phases. The generated representation is assessed with respect to the requirements regarding the product and the process. Figure 2.3 shows one refinement step in detail. Evidence of meeting requirements is established by validation and verification, often referred as V&V activities. Validation and verification aim at answering different questions:

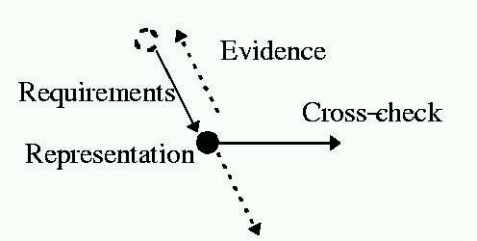


Figure 2.3: V-Model: Performing one Refinement Step

Validation : "Do we build the right product?"

Verification : "Do we build the product right?" [fEA01]

For validation of a product, the system developed so far is assessed with regard to functional (customer-) requirements. The primary focus of validation is to show that the expectation of the user is satisfied by the product. In contrast to validation, verification shows if a product meets the requirements specified during previous activities and does not contain undesired functionality. Verification focuses on non-functional requirements, such as reliability, safety, performance, etc. Prominent methods employed for verification are analytical, such as FMEA (Failure Modes and Effect Analysis), FTA (Fault Tree Analysis), exhaustive testing, reviews, formal verification and prototype studies. In order to enable successful application of verification techniques, the requirement specifications must be unambiguous, consistent, complete and verifiable.

Unambiguity means that there is only a single interpretation of the specification, consistence means that there are no conflicts between specifications. Completeness denotes the fact, that all necessary information is included in the specification. The specification is called verifiable if (a person or) a tool can check the specified system for correctness. Specifications must also be modifiable, i.e. have structuring and style, which allows to apply changes consistently and correctly [RTC92].

Informal specifications written in natural language are often ambiguous, it is difficult to guarantee consistence and completeness. In contrast, formal specifications can be easier assessed with respect to the above criteria. Since formal specifications follow a clearly defined syntax and semantics, analysis and assessment of formal specifications can be supported by software tools, such as code-analyzers, compilers, simulators or formal verification tools.

2.3 Placement of STVE-Techniques in the Model Based Development Process

STATEMATE allows the user to formalize requirements by models and thus to detect inconsistencies, ambiguities or incompleteness of specifications. Due to the simulation capabilities of the STATEMATE system, models can be used as executable specifications. The models can be modified or features can be added without much effort or risk, allowing assessment of the functionality of the designed system already very early in the development process.

Even though STATEMATE's integrated check tool is able to detect many completeness and correctness errors in badly formed models, syntactically correct models may show undesired effects which

can not be detected by the check tool, because finding these effects requires dynamic analyses. Among these modeling artefacts are conflicting transition-triggers and hazardous data accesses, which should not occur in reference models for later phases of system development. In view of the increasing complexity of embedded controllers, simulation is in general not sufficient to ensure correctness under all circumstances. The pure amount of possible computations is too large to be entirely covered by simulation. Thus, simulation can expose conflicts in a large model only at random.

Formal verification can be applied to a model, completing the repertoire of the developers for examining the dynamic behavior of the modeled system. The STATEMATE verification environment (STVE) offers simple, but formal analyses for conflicts that may occur in a model. These *robustness* checks cover:

- transition non-determinism (concurrently enabled transitions),
- read/write hazards (a variable is read and written simultaneously)
- write/write hazards (concurrent write accesses to a variable)
- range-violations in variable assignments (a variable is assigned an out-of range value)

For each kind of conflict a specific analysis of the model can be invoked without a need for expert knowledge regarding formal verification.

For each detected potential conflict, a verification task can be invoked searching the entire state space of the model for a dynamic occurrence of the conflict. If the potential conflict turns out to be a real one, a *simulation control program* is created which can be loaded and executed with the STATEMATE simulator to drive a simulation exposing the conflicting situation.

Besides these *'robustness analyses'*, different *'formal debugging'* aids are offered by the verification framework for purposeful dynamic exploration of the model under construction. Formal debugging can be applied in order to produce simulations

- reaching particular basic states (graphical states)
- reaching configurations of basic states (according to a user selected set of states)
- taking a particular transition (graphical transition)
- In the most general form, developers can check whether there exist computations reaching a configuration, which has been specified by a user defined expression.

Formal debugging can be applied in order to detect dead code or to produce simulations driving the model into particular configurations of interest.

Thus, developers are supported in examining models under development using fully automated formal techniques. Robustness checks and debugging techniques are applicable at every decomposition level, allowing developers to focus on sub-components even if a model has grown to a complexity which is not easy to handle by formal methods.

Beyond robustness and formal debugging checks the STVE offers an easy to use verification support for the verification of functional requirements. In order to express functional requirements the developer can instantiate specifications from a set of predefined specification patterns by mapping the formal parameters of the appropriate patterns to model specific expressions. The set of

predefined patterns covers a wide range of frequently used specification schemes, which can be instantiated as requirement as well as for specifying assumptions about the environment of the model. In contrast to robustness checks and formal debugging, patter-based verification is oriented towards certification, i.e. the specification is expected to hold for the model.

Critical functional requirements such as safety, timing or mutual exclusion properties can be verified during the development iteration cycle, without requiring expert knowledge.

Robustness checks, formal debugging and pattern verification can be applied as on-the-fly techniques by developers during the development cycle. In fig. 2.1, this corresponds to state `b.proc`. By this means, the techniques described so far can be seen as constructive quality assurance measures, which help to increase the quality of the developed system already during development. Formal verification enables the developer to ensure that the model adheres to basic quality criteria before submitting the models for acceptance. The same techniques can, of course, also be applied as analytical measure after submission of the model for acceptance to quality assurance.

The expressive power of pattern verification is limited by the extent of the pre-defined pattern library. For efficiency reasons, pattern are offered only with a limited amount of formal parameters. Hence, only relatively simple requirements can be captured by pattern instances, specifying a temporal relationship between up to four parameters. In order to gain a deeper insight into the inter-component communication and for the verification of more complex functional requirements, the graphical specification formalism Symbolic Timing Diagrams (STDx) can be used to specify temporal requirements graphically. Like pattern verification, also verification using STDx specification is oriented towards a true result (also referred to as *certification*), i.e. the expectation is that the specified property holds on the model.

While robustness checks, formal debugging and pattern verification require little knowledge about formal verification techniques, using STDx is not applicable without expert knowledge. In addition, formal specification and verification of intricate requirements is complicated and time consuming. Thus, verification using STDx should be performed at higher levels of model maturity and stability than during the fast development iteration. This, again corresponds to the iteration cycle (cf. Fig. 2.1) proposed by the V-model: When a development phase is completed and the model is submitted to quality assurance, more extensive verification may serve as quality criterion for acceptance. The effort for verification of all requirements is justified by the benefit of getting evidence for meeting the requirements. Establishing quality assessment by verification using STDx specifications requires expert knowledge at a critical phase of the development. The submission arc from `submitted` to state `accepted` is triggered by the quality assessment team. Formal techniques enrich the repertoire of quality assessment methods.

The offered verification techniques can be applied to the entire system model as well as only to sub-components. This can be used to assess the allocation of requirements to components of the system according to the regulations by the V-model as cited on page 13.

Formal verification also supports *virtual* integration. Especially for safety critical systems, critical and important parts of the system can be modeled first, while missing parts are represented by dummy implementations. For verification of the already modeled parts of the system, often assumptions about the environment must be formalized. These assumptions are claims about the missing parts of the model and can hence, serve as requirement specifications for their realization.

STATEMATE models serving as executable specifications are life-cycle data in a development process. Formal verification as offered by the STVE supports the development team in producing unambiguous, verifiable, consistent, modifiable, and traceable life-cycle data as required by the standards.

Formal techniques can be used in a model based development process to create a *golden device* that has a fully and rigorously validated specification. This golden device is taken as basis for all subsequent implementation steps, e.g. for automatic code generation or for successively replacing the sub-models by code.

3 Using Statemate

This chapter gives a short introduction to the design-tool STATEMATE. In section 3.1 we briefly describe the main concepts of statecharts, activity-charts and the expression language of STATEMATE. Section 3.2 brings the simulation capabilities of the tool set into focus and explains the two different time models regarding which Statemate models can be interpreted by the simulator. In section 3.3, we present the case study, which will serve as running example throughout this thesis. The case study will be used to evaluate the concepts and tools presented in the remainder of this work (cf. chapter 8).

3.1 Statemate

In order to improve both the efficiency of the development process and the quality of the product, model based development processes have been established by many companies developing safety critical embedded systems.

"Modeling is a proven and well-accepted engineering technique. We build models to communicate the desired structure and behavior of our system clearly, and unambiguously. We build models to improve our understanding of the system under consideration, uncovering errors and defects well before anything has been built. Building models allows us to try out new ideas and concepts with a minimum of cost and a minimum of risk. We build models to manage risk" [LIS02].

In a variety of companies STATEMATE is employed as CASE tool (Computer Aided Software Engineering).

The STATEMATE system is commercial tool-set, built around the visual language of *statecharts* [Har87]. It has been under development and extension since early 1984 [HLN+90]. Founded in 1987, i-Logix has maintained and extended the tool-set since then. In 1996 the tool-set was re-introduced as STATEMATE MAGNUM[1] with many advanced modeling features, providing powerful tools for development, analysis and documentation of complex reactive systems. STATEMATE enables engineers to rapidly design and validate complex system level products through a combination of graphic modeling, simulation, code generation and documentation generation. As a result, STATEMATE has emerged as a standard for high-end embedded systems development within the medical, automotive, aerospace, and defense industries[2].

[1] We will in the following use the shorter name 'STATEMATE' instead of always referring to 'STATEMATE MAGNUM'.

[2] Among the companies employing STATEMATE in their development process are such prominent companies as, for example:

- Aerospatiale, Boeing, BAE Systems, EADS, and Lockheed Martin in the avionics-domain,
- BMW, DaimlerChrysler, Denso, Nissan, Renault, Volkswagen, and Volvo in the automotive domain,
- CISCO, Motorola, Nokia, and Siemens Telecom in the telecommunications domain [iL04]

An up-to-date list of customers using STATEMATE can be found at http://www.ilogix.com.

General Concepts

The offered high-level modeling concepts qualify STATEMATE particularly for usage in the early phases of a model based development process. Conceptual models are built already in the requirement analysis phase in order to analyze functional aspects of the initial requirements and to explore design alternatives.

The underlying database concept allows multiple developers to work independently with their instances of the model. Changes and derivatives of the original model can be checked again into the database and thus made accessible to the team. The version control of the database provides a concept of configurations which individually reflect particular phases of the model evolution.

Part of the tool-set is a simulator, which can execute the model under construction. The simulation of models can be run in batch mode or controlled by user interaction, either with or without panels. A panel editor enables the user to build easy-to-handle graphical interfaces for the simulation. Construction and usage of panels supports the developer in exploring and demonstrating the behavior of models. Meaningful simulation of easy-to-build conceptual models is one of the key arguments for following a model based development process, where models capturing concepts at an abstract level serve as executable specifications and reference models for subsequent phases. Besides other visualizations such as a waveform viewer, the graphical design objects are animated during a simulation, such that the developer can easily observe e.g. which parts of the model are active, which transitions are taken, or which outputs are produced. Since the semantics of STATEMATE models is defined in terms of simulation we will come back to the different aspects of execution in section 3.2.

The visual formalisms offered by STATEMATE support different modeling concepts and styles. Three graphical modeling languages can be used to develop a model according to a functional, behavioral or structural point of view. *Activity-charts* serve for the description of functional aspects, *statecharts* [Har87] are used to model behavior, and *module-charts* describe a structural view. Module-charts are used as data-flow diagrams constituting the implementation of the system, its partitioning into hardware and software blocks, and the communication among these blocks. This physical model aims at specifying which physical module implements which given conceptual model - implemented by activity-charts[3]. Since this work focuses on conceptual modeling, we will not regard module-charts.

Conceptual models are built using activity-charts and statecharts. Activity-charts are used to partition the conceptual model into functional blocks. Ideally, each activity-chart of a STATEMATE model represents a separated function of a system. Flow of data and control among activity-charts can be specified graphically by flow-lines. Activity-charts can be used in a hierarchical manner. They can be decomposed again into activity-charts in order to structure a function into sub-functions. Leaves of this decomposition are *basic-activities*. Basic-activities are modeled either by *statecharts* or by so called *mini-specs*, which are programs written in the Statemate action-language[4].

[3] In STATEMATE module-charts are mainly used for documentation purposes. [HP98] explains the intended relation between activity-charts and module-charts.

[4] STATEMATE also supports activities implemented by other, external programming languages, such as C or VHDL. *External implementations* are useful for implementation of target-architecture dependent functions, such as e.g. integration with a particular operating system. Since such a detailed implementation leaves the conceptual level, this feature is out of the scope of this document.

Statecharts

Statecharts extend conventional input/output automata by a concept of hierarchy, parallelism, and broadcast communication. They are basically hierarchical state-transition diagrams, where states can contain entire statecharts. Regarding its possible sub-states, a state is called either *OR* state, *AND* state or *basic* state. Sub-states of an OR state can only mutually be active. A sub-state of an AND state can only be active if also the sibling sub-states of the AND state are active. Basic states are at the bottom of the hierarchy and do not have further sub-states.

A state is entered or activated by taking a transitions ending in that state, it is exited or deactivated by taking a transition starting in that particular state. Transitions are not restricted to a particular hierarchy level of the statechart. They can cross multiple levels of the state hierarchy. Figure 3.1 shows an example: States `C1`, `C2`, `B1`, `B2` and `D` are basic states, while `ST_TOP`, `C` and `B` are OR states and `A` is an AND state.

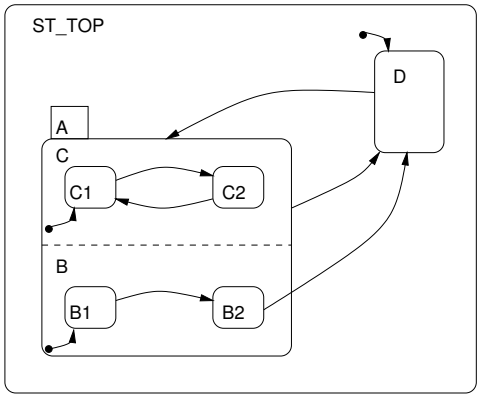


Figure 3.1: Simple Statechart

Transitions in STATEMATE statecharts are either so-called *default-transitions* or transitions between states. Default-transitions activate their target states when entering the enclosing state. In figure 3.1, states `C1`, `B1` and `D` are targeted by default-transitions. `D` is activated when `ST_TOP` is activated. States `C1` and `B1` are activated when `A` is activated.

In order to increase the readability of statecharts and the flexibility of the visual formalism, STATEMATE offers a variety of transition related concepts which will be explained in the following. [HN96] gives a detailed description of these concepts.

- Transition connectors : transitions do not need to start or end in states, but can also be connected to so-called *transition-connectors*. Each possible combination of transition-segments connected by transition-connectors between a start and an end state forms a so-called *compound transition*. Only an entire compound transition can be taken at once. It is not possible to take only a single segment of a compound-transition and get stuck at a connector.
  Connectors increase the readability of statecharts, avoiding the use of redundant (and possibly conflicting) transitions. Besides condition connectors STATEMATE offers default-, termination-, history- and deep-history-connectors which can be used as start- or end of transitions.

  – Default-connectors : default-transitions can be connected to transitions using a condition-connector. It is thus possible to determine the default-state of a statechart dynamically,

depending on which transition-segments connected to the default-transition by a default-connector are enabled.

- Termination-connectors : when a transition ending in a termination connector is taken, the enclosing activity is de-activated (stopped). The de-activated activity can only be activated again by an external activation. Thus, using termination connectors, self-terminating activities can be realized.
- History- and deep-history-connectors : if a transition ending in a history-connector is taken, all active sub-states of the enclosing OR state are memorized. If the OR state is de-activated and entered again using a history-connector the stored configuration becomes active again. A deep-history-connector not only memorizes the active sub-states of an OR state, but also the complete configuration of active states below the active sub-state.

- Inter-level-transitions : transitions do not need to only start or end in basic states but also in hierarchical states and may also cross multiple levels of the state-hierarchy. Thereby, the state hierarchy is used to determine the resolution of conflicts. If transitions of different levels could be taken in the same step, STATEMATE priorizes the highest level transition. For example, if in figure 3.1 the transitions starting in `A` and the one starting in `B2` are concurrently enabled, the transition starting in `A` will be chosen. Also the effect of taking a transition depends on the state hierarchy. If the transition from state `B2` to state `D` is taken, all sub-states of `B` are de-activated and since `B` is a sub-state of AND state `A,` also `A` is deactivated - which automatically de-activates also the sub-states of `C`.
  Hence, if an inter-level-transition is taken, always the entire state hierarchy is de-activated which is on lower hierarchy levels than the top-most state left by the transition. If a transition ending in an AND or OR state is taken, the default transition(s) or history-connector(s) in the entered hierarchical state determine which sub-states of the state are activated.

Transitions are labeled with a trigger- and an action-part. In general, transition labels are of the form '*e[c]/a*', where *e* is an expression ranging over events, *c* is a boolean expression - not referring to events - , and *a* is the action-part of the transition to be executed when taking the transition[5]. The conjunction of event-part *e* and condition-part *c* forms the trigger. A transition can only be taken if both parts of the trigger evaluate to *true*. All parts of a transition label are optional - the default for *e* and *c,* respectively*,* is *true*, whereas the default for action-part *a* is to perform no action at all. The transition-trigger of a compound transition is logically built from the triggers of all chosen transition segments connected by condition-connectors. Accordingly, the action-part of a compound transition is built from all action-parts of these segments. Triggers of transitions enabled at the same step need not be mutual exclusive and there is no static resolution strategy which transition is priorized in such a case, except for - of course - the hierarchy rules[6]. STATEMATE permits conflicting transitions - non-determinism can be utilized for abstract modeling. On the other hand, if non-determinism is not desired, it remains the responsibility of the developer to ensure determinism of a model.

[5] In Statemate conditions and events are strictly distinguished. While conditions can be arbitrary boolean expressions rranging over the variables of a model, events are dedicated volatile occurrences, which are visible only for one step in the execution of a model. We will consider this distinction in more detail in the sequel.

[6] Other CASE-tools provide e.g. a user defined weighting of transitions (ASCET) or a clockwise priorization (Stateflow).

In addition to the use of transitions, each state can be associated with a so-called *static reaction*, which has the same format like transition labels. A static reaction is considered (and the action is performed if the event and condition expressions both evaluate to true) only when the state in question is active - and no outgoing transition can be taken. As a variant of this concept reactions, can also be bound to entering or exiting a state.

Data-items

A data dictionary collects data type and variable declarations, as well as function and procedure definitions. Various data types can be used in the model: events, integers - optionally bound by upper and lower bounds or a number of bits - , enumerations, reals, bits, conditions, strings and bit-arrays. STATEMATE supports also compound types such as records, unions, arrays and queues. The designer may build user defined types of arbitrary nesting depth from these basic types using compound types. Based upon these types, variables can be defined and used in the model.

Variables are associated with a scope, they can be local to a particular activity or be globally visible in the model. The default scope is determined by the first usage of the variable. The designer can change the visibility or scope of a variable in the data-dictionary in order to e.g. share a data-items among different activities. If not explicitly assigned a new value within an action, normal variables keep their values. Especially activation or deactivation of activities or states has no influence on once assigned values of variables.

In contrast to normal variables, events have a different semantics: Events are visible only for one step in the execution of a model - events can be consumed only in the step after being issued, in the step thereafter events are no longer remembered.

Expression Language

Besides user-defined events, the STATEMATE action-language provides a set of special events - associated with the different kinds of objects, such as '*en(s)*' which is issued when state s is entered, or '*fs(c)*' which is issued when condition c becomes false. For each variable v the implicit events '*wr(v)*', '*rd(v)*' and ' *ch(v)*' indicate that v has been written or read, respectively, or has changed its value. Some of these implicit events apply also to more complex expressions, such as e.g. '*fs(<expr>)*' issued when boolean expression *expr* becomes false.

Part of the expression language are also special events for scheduling of activities, such as '*st!(A)*', starting - or activating - an activity A or '*sp!(A)*' stopping - or de-activating - A. Activities can alternatively be only suspended and resumed later by scheduling events ('*sd!(A)*' and '*rs!(A)*', respectively). The actual activation status of an activity can be referred to using a set of dedicated events and conditions e.g. '*st(A)*' (started), '*hg(A)*' which means hanging, i.e. activity A being suspended but not resumed yet, '*ac(A)*' meaning active, i.e. not stopped yet, e.t.c. These scheduling events play an important role for modeling a reactive system. Using these events, STATEMATE enables the developer to realize arbitrary scheduling - activating and de-activating parts of the model according to the requirements of whatever protocol. Table 3.1 gives an overview of the events and conditions related to the scheduling of activities.

The expression language also provides constructs to refer to time. Since the model itself contains no encoding of time, *time* refers to simulation time which is controlled by the simulator when executing the model. We will discuss the execution of models and the interpretation of time in

	action	comment	status	comment
activation	*st!(A)*	start *A*	*st(A)*	event emitted if *A* started
de-activation	*sp!(A)*	stop *A*	*sp(A)*	event emitted if *A* stopped
status			*ac(A)*	condition, true if *A* is active
status			*hg(A)*	condition, true if *A* is active but suspended
suspension	*sd!(A)*	suspend *A*		
resumption	*rs!(A)*	resume *A*		

Table 3.1: Events and Conditions related to Scheduling

detail in the following section 3.2. References to simulation time are supported in two different ways:

timeout events explicit timing information can be introduced in a model by using timeout events. The general form of a timeout event is '*tm(e,T)*', where *e* is an event starting a *timer*, and *T* is an expression determining the amount of abstract time units after which a timeout is indicated to the model. The timer is reset each time *e* occurs.

scheduled actions While timeout events can be used in transition triggers in order to enable a transition a particular time after a certain event was issued, scheduled actions permit to time-trigger actions. The general form of scheduled actions is '*sc!(a,T)*', where *a* is the action to be performed after *T* time-units of simulation time will have expired. Once scheduled nothing can prevent *a* from being executed. This is a notable difference to the interpretation of timeout events, where the timer will be reset if its triggering event is observed again, before issuing the timeout.

Combination of Statecharts and Activity-charts

Statecharts are a comprehensive formalism for modeling behavior and are due to hierarchical structuring often considerably more compact than equally expressive traditional input/output automata. While being adequate for describing a behavioral entity, statecharts in no means support separation of functions. On the other hand, the graphical language of activity-charts is aimed at separating functional entities of the design. Although mini-specs can be used to describe behavioral aspects of an activity-chart, they lack the intuitiveness of using statecharts. With the combination of the closely related functional and behavioral aspects of activity-charts and statecharts STATEMATE offers adequate formalisms for building conceptual models:

"The backbone of the system model is an *activity-chart*, which is a hierarchical data-flow diagram, and in which the functional capabilities of the system are captured by *activities* and the data elements and signals that can flow between them" [HN96]. Each of these activities can contain at most one statechart per decomposition level.

Statechart implementations of activities - basic activities - are called *control activities* if they dynamically control activation or de-activation - or suspension and resumption - of activity-charts which are placed on the same hierarchical level. Dynamic activation and scheduling of activity-charts using scheduling event expressions forms the connection between the functional and the behavioral view. If an activity contains no control-activity (in its graphical decomposition), all sub-activities

are activated whenever the activity itself is activated. If it contains a control-activity, only the control-activity is activated. This control-activity is responsible for starting, stopping, suspending, and resuming all other activities of the same decomposition-level. If a control-activity is terminated or terminates itself, all controlled activities are terminated.

According to their usage within a system model, for controlled activities three different termination modes are distinguished:

controlled-termination The activity can only be terminated by the controlling control-activity.

self-termination The activity can terminate itself, the controlling control-activity needs to either poll for the status of this activity or to react on its termination event.

procedure-like The activity performs its actions when activated (within one step) and terminates afterwards.

So far, we have considered STATEMATE merely from a constructive and syntactical point of view, even though execution aspects could not be entirely disregarded. Of course, this section can not focus on all interesting aspects of STATEMATE models, nor is it the scope of this document to provide a description of all STATEMATE modeling features. A detailed description of modeling features as well as a discussion of their concepts can be found in [HLN+90, HN96].

All modeling techniques as well as the expression language constructs listed above - far from being complete - are aimed at a dynamic execution of models making use of these constructs.

3.2 Execution of Statemate Models

The simulation capabilities of STATEMATE play a central role in its prominence as development tool for embedded systems. Simulation can be performed at each level of decomposition. Single activity-charts or statecharts can be executed in isolation as well as the entire model can be executed at once. Arbitrary sub-models, each possibly consisting of various levels of activity-charts and statecharts can be selected for simulation. The STATEMATE simulator has been built fault-tolerant regarding incompleteness of models[7] and can be invoked in every phase of development - after applying a static check ensuring correctness w.r.t. declaration and usage of data-items and proper usage of graphical constructs. This check - called *check model* - is not mandatory, but strictly recommended in order to avoid waste of time with simulations resulting in execution errors.

The capability of simulating a sub-systems in the context of the entire system model - or in isolation - even in early phases of the development contributes to the idea of virtual integration in a model based development process.

The simulator can be used interactively, such that the user can drive the simulation either with user-designed panels or by directly influencing data-items, providing events and injecting all desired changes to the model. In interactive simulation, the user has a high degree of control over the simulated model and the progress of simulation time. In addition to interactive simulation, the simulator can also be controlled using so-called *simulation control programs* (SCP). SCPs can be used to drive the model according to pre-recorded stimuli. The entire simulation can be controlled using this procedural interface, without requiring user interaction. In the verification environment

[7] sometimes issuing warnings or error messages if incompleteness leads to execution problems.

SCPs are extensively used to provide simulations according to witnesses obtained from verification tasks.

The STATEMATE simulator can execute models using two different interpretations, a *synchronous* and a so-called *asynchronous* simulation semantics. The most important differences between these interpretations lie in the interaction with the environment and in the concept of time. It must be emphasized that *time* in this context always does not refer to to physical time but only to the virtual time of the simulator, which is controlled by the simulator according to the chosen execution semantics.

Synchronous Simulation Semantics In the synchronous simulation semantics, the system executes a single step per virtual time unit and then accepts new external stimuli. As result of performing action parts of transitions and static reactions, new events are generated and variables are assigned new values. The environment can provide new input stimuli at each step. Execution of a step performs infinitely fast in zero simulation time. Simulation time advances only between consecutive steps.

Asynchronous Simulation Semantics In contrast to the synchronous interpretation, the system continues computing steps unless no further transition can be taken. Only if no further reaction to external or internal changes has to be performed, the model becomes *stable*. The environment can stimulate the system with new inputs only if such a *stable status* is reached. A maximal sequence of single steps after taking the input stimuli until reaching again a stable state is called a *super-step*. Similar to the synchronous interpretation, the execution of a super-step is assumed to consume no time. Simulation time advances only between consecutive super-steps.

In [HN96], Harel emphasizes that the time interval between the execution of two consecutive steps or super-steps, respectively, is not part of the semantics. The interval "depends on the execution environment and the time model, over which users of the tool have significant degree of control". Hence, it depends on the user - in interactive simulation - how much simulation time passes between successive steps or super-steps.

> "The time calculated in dealing with the explicit time expressions appearing in timeout events and scheduled actions is measured in terms of some abstract time unit common to an entire statechart. Different statecharts can have different time units, in which case the relation between them must be specified prior to model execution"[HP98].

[HN96] provides a detailed informal description of the STATEMATE simulation semantics of statecharts. Modeling reactive systems using the full bandwidth of capabilities of STATEMATE is explained in [HP98] by example.

The execution semantics of STATEMATE models is defined in a constructive way by defining a *basic step algorithm*. We only will give a sketch of the basic step algorithm here, for a detailed description see [HN96].

Basic Principles of Execution

At the beginning of each step, the environment supplies the model with external stimuli. These stimuli trigger transitions and static reactions of the model, together with changes that occurred

in the model during the previous step or due to advancing simulation time. As result, the model changes its status. Transitions are taken, some states are exited and some other are entered. Values of conditions and data-items are modified, activities are activated or de-activated, and new events are generated.

All these changes take place respecting some basic principles :

δ-delay Reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step. Calculations of one step are based on the situation at the beginning of the step. The values of conditions and data-items regarded in expressions during a step computation are the values from the beginning of the step. *Actions performed during a step do never affect other actions performed within the same step*[8]. A δ-delay can not be measured in terms of simulation time. Regarding simulation time, δ is infinitely small - it just *preserves the temporal order of cause and effect.*

momentariness *Events "live" for the duration of one step only*, the one following that, in which they are generated. Events are not "remembered" in subsequent steps.

greediness *Always a maximal subset of transitions and static reactions is executed.*

synchronism *The execution of a step itself is assumed to take no time.* The time advancing between the execution of two consecutive steps is not part of the semantics, but depends on the execution environment. Hence, several steps may be executed at the same point in simulation time.

The concept of a δ-delay copes with a serious problem of synchronous languages. The *perfect synchrony hypothesis* [BB91] - which has been widely accepted by the reactive systems community - postulates that reactions to external stimuli and internal communications take zero time. It is asserted that response to external events takes place immediately. Hence, in synchronous languages, inputs and outputs for a computation of a model are observed at the same moment. This assertion leads to the so-called *causality paradoxon* of synchronous languages, which can be illustrated by the simple example of a transition labeled with `'not(E)/E'`, where `E` is an event. The action performed if `E` is not observed, prohibits its own cause. By respecting a δ-delay, STATEMATE avoids this causality problem, since the effect of transitions can have no impact on triggers within the same step. Self-triggering, i.e.: events that are executed based on their own occurrence - or non-occurrence - and not caused by external events - is impossible within one step.In order to realize the concept of a δ-delay for modification of the status, the basic step algorithm performs assignments in two

[8] The term *δ-delay* is borrowed from VHDL: "VHDL has a two-stage model of time. This two-stage model is referred to as the simulation cycle. (...) During the first stage of the simulation cycle, values are propagated through the data pathways (signals). This stage is complete when all data pathways which are scheduled to obtain new values at the current simulation time are updated. During the second stage, those active elements (processes) which receive information on their sensitivity channels are exercised until they suspend (via the execution of a wait statement). This stage is completed when all active processes are suspended. At the completion of the simulation cycle, the simulation clock is set to the next simulation time at which a transaction is to occur and the cycle is started again. (...) The above model means that there is always some delay between the time a process puts a value on a data pathway and the time at which the data pathway reflects that value. In particular, if no delay is given in the assignment of a value to the data pathway a delta delay is used. This delay does not update the time of the simulation clock but does require the passing of a new simulation cycle. (...) When a value is assigned in the pathway it is not immediately available to processes which read the value from the pathway. There is a delay between assigning and updating a signal value" [LSU95].

phases. In the first phase, the effects of assignments are projected using a list of pairs, each one of the form <*element, new-value*>. Only after all assignments to be performed have been considered for the old values of the elements, the second phase really assigns the new values to the affected elements.

The algorithm computes the set of enabled compound transitions w.r.t. the priority rules of the hierarchy. The set of transitions is split into non-conflicting subsets. For each of the sets of compound transitions the enabled static reactions are computed which are defined in states that are currently active and are not exited by a compound transition of the set. In case of conflicting subsets of compound transitions the simulator offers the user the selection of one of the nondeterministic alternatives[9]

Regarding conflicting write accesses to variables, the simulator follows a different strategy: Here only a warning is issued to the user; the written value is chosen according to an internal resolution strategy without prompting for user-interaction.

It is important to distinguish between the 'basic step algorithm' defining the execution of STATEMATE models and the various features of the simulator.

In the synchronous time model, simulation time is automatically increased by one time unit between two successive steps. For the asynchronous time model the situation is different: Since the execution of steps does not advance the simulation time, the simulator's operator has to do so explicitly. Therefore the simulator offers several different GO commands that let the user control the advance of time during simulation.

In addition to the different GO commands the simulator offers a 'continuous' simulation in which the model continuously computes steps and time advances automatically by one time unit per super step. The user can interactively provide the system with inputs, for example using a control panel.

Stepwise simulation enables easier debugging of the model. In a step-by-step simulation mode the simulator allows the user to nearly arbitrarily influence values of local variables and conditions of a model as well as to provide the model with events. Hence, the user can also assigning values to local variables which could never be assigned by the model itself in any normal simulation.

Perfect Synchrony Hypothesis

Both variants of STATEMATE's execution semantics are based on the assumption that internal computations take zero time. Internal communication among activities is synchronous in the synchronous as well as in the asynchronous semantics. It is thus interesting to consider both variants regarding their conformance with the *perfect synchrony hypothesis:*

For STATEMATE's synchronous execution semantics it is obvious that the perfect synchrony hypothesis does not hold: the model reacts to external events with a chain-reaction of generating internal events, each of them triggering reactions in the subsequent steps. The time it takes to react depends on the number of steps needed to complete the reaction of the system.

Regarding simulation time, the synchrony hypothesis holds for the asynchronous execution semantics. Reaction to external events with a series of steps, with one event triggering another, until reaching a stable status consumes no simulation time. The entire super-step takes zero simulation time. Time may only advance after the reaction to external events has been completed and the model has reached a stable status. But also the asynchronous execution semantics does not fully comply with the perfect synchrony hypothesis: the model can only emit events during a super-step.

[9] In contrast to the simulator, the STATEMATE code-generator selects one of the alternatives arbitrarily.

The model only becomes stable if no reaction to external or internal changes took place in the previous step, and thus no event has been emitted. In contrast, events provided by the environment are only sensed in stable states. Hence, w.r.t. event communication, the underlying step execution has to be regarded and reactions of the model are only observable *between* consecutive stable states.

Compositionality

A drawback of the asynchronous interpretation by the simulator is being not compositional. STATEMATE supports the simulation of activities of arbitrary design-levels. When simulating a system model using the asynchronous interpretation, the environment can provide the model with new stimuli only in stable states - the communication with the environment is asynchronous. In contrast, local communication among activities of the model is synchronous, i.e. changes can take effect every step. This becomes different, when simulating an internal activity in isolation - without the enclosing design-level. Sibling internal activities of the model do not belong to the scope of this simulation, but belong to the environment. Communication with these - now external - activities is treated as communication with the environment according to the asynchronous semantics. Thus, the interpretation of activities depends on the chosen scope for the simulation, because parallel composition of activities is always step-based. Only the communication of the composed model with its environment is interpreted according to the asynchronous execution semantics. STATEMATE does not support an *asynchronous* communication among activities of a parallel composition.

Another issue of compositionality for both the synchronous and the asynchronous semantics is the treatment of shared variables: Data-items can be shared among different activities. Simulation regards data-items only w.r.t. the simulation scope. When simulating an internal activity in isolation, effects of data accesses from external activities are not regarded. Although a data-item might be accessed outside the simulation scope, it is treated to be local to the simulated activity.

In [DJHP97] Damm, Josko, Hungar and Pnueli have presented a formal compositional semantics for the synchronous and asynchronous semantics of STATEMATE designs. We will discuss the representation of STATEMATE models according to this semantics in section 5.

3.3 Case Study: Radio-based Signaling System

In the previous sections, STATEMATE's modeling and simulation capabilities have been briefly explained. In this section we will present an example application of STATEMATE. This application will be used as reference example throughout this document.

The model has been developed within the DFG (Deutsche Forschungsgemeinschaft/German Research Foundation) focus area program *Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen* (Integration of Software Specification Techniques for Engineering Applications). The application is one of two reference case studies of the focus area program which were provided in order to be able to compare the results of the individual projects. Modeling has been carried out by the projects USE and FORMOSA. A detailed description of the model can be found in [KT00] and in [Klo03].

The STATEMATE application models the radio based interaction between trains and level crossings. Train and crossing react autonomously as far as possible. The communication medium establishing communication between train and crossing is also part of the model.

Conventional railway crossings are controlled by wayside hardware, for example sensors which announce an approaching train to a crossing, signals indicating the status of the crossing, etc.

This hardware is permanently installed and must be able to handle trains with various differing properties, like different length, speed, and so on. A high effort is necessary to keep the hardware operational and to control its correct behavior. Thereby, national standards inhibit transnational traffic. Currently, a locomotive has either to be able to adapt to another national standard or has to be exchanged when trains cross frontiers of different countries.

European railway companies currently investigate better and more flexible solutions for the control of railway traffic. Among these solutions radio-based communication plays an important role, since permanently installed wayside elements can be avoided. Protocols can be adapted to varying situations. For example, a fast train could announce itself earlier to a crossing than a slow one. A crossing could react according to priorities. Since the components required for the communication are located directly at the train and the crossing, less maintenance effort is necessary compared to conventional solutions.

The train sends its requests and messages to the communication medium which propagates them to the crossing and vice versa. Hence, at this top-level view the system consists of three communicating core activities: `TRAIN`, `COMMUNICATION` , and `CROSSING`. Train driver, 'physical' components, such as barrier and lights with their sensors, and the crossing control center are modeled as environment. This environment interacts with the signaling system by indicating, for example, a lights defect or sensing the state of the barrier. The core system has to behave appropriately under these environmental circumstances.

In order to describe a generic situation and to keep the model as simple as possible, the focus of modeling has been the communication between one train and one crossing. Hence, an interleaving of two or more securing procedures is not covered by the case study. Also, two or more trains approaching the same crossing are not regarded. Additionally, only the essentials of the involved ingredients are regarded in the model, e.g. only these parts of the train which are relevant to the communication protocol were modeled.

It has been a general decision in the model development to keep some details abstract. Some of the functions belonging to parts of the model are realized in a virtual manner. The environment provides the results of these abstract functions as input to the model. This has been justified by the wish to provide all individual projects of the focus area program with a uniform basis for their particular approaches, permitting differing interpretations concerning details which are not directly relevant to the protocol. On the other hand, a basic idea of the model based development is illustrated by the application: abstraction from details which are not relevant to safety or mission critical properties of the system under construction.

Figure 3.2 shows the top-level of the radio-based signaling system. `TRAIN`, `CROSSING` and `COMMUNICATION` are modeled as internal activities of top-level activity `SYSTEM`. `DRIVER`, `SENSOR` , `LIGHTS`, `BARRIER`, and `OPERATION_CENTER` are external activities contributing to the communication protocol between the internal activities only indirectly. In reality, physical components like `SENSOR` , `LIGHTS` and `BARRIER` can fail due to physical malfunctions. This is taken into account by providing inputs for these components indicating defects.

A train approaching a crossing reaches a specific distance, which is called activation point. This activation point is the latest position at which a train can initiate the securing of a crossing in order to pass without braking. In the radio-based protocol this point is determined dynamically. In order to determine the exact position of the activation point, the delay for communication setup, message transmission, the time required to secure the crossing, and speed and position of the train have to be taken into account.

In order to determine the activation point, the train possesses a *track chart* which keeps relevant

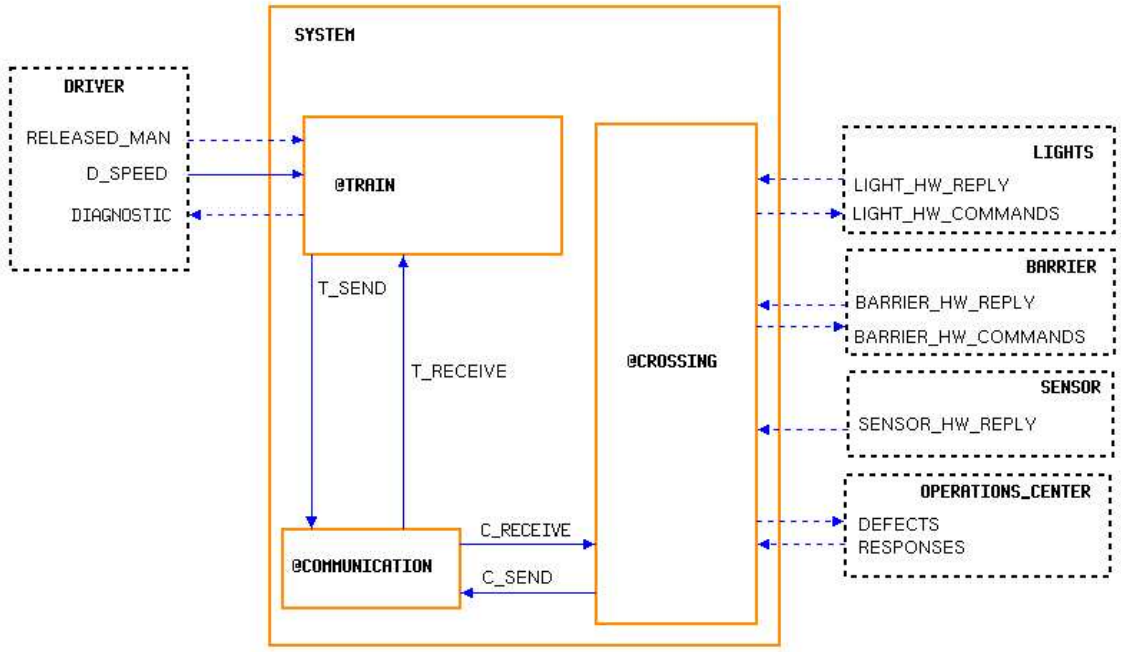


Figure 3.2: Top-level of Radio-based Signaling System

information about the track, like maximal speed, positions of crossings, stations and so on. According to this track chart, the train places *control points* at all positions to be specifically regarded. With each of these control points a target speed is associated, which steadily has to be compared to the actual speed. The train accelerates or reduces speed according to this comparison. In the model version presented here, the track chart has been left empty, an input provides the train with the position of the crossing instead.

After an amount of time has elapsed, which under normal circumstances is required to secure the crossing, the train requests a status report of the crossing - this is modeled in statechart `WF_CROSSING_SAFE` (cf. figure 3.7) using a timeout event with constant `CCT` (Crossing Closing Time). If the crossing is safe, the crossing issues a *safe*-report, otherwise the crossing does not respond at all. At this point the hardware failure detection comes into play: if the lights or the barriers indicate a malfunction, the crossing will not indicate a secured status.

A sensor detects the passing and sends a signal indicating that the train has passed to the crossing. After receipt of this notification the crossing returns to its normal state, i.e. raises the barriers and turns off the lights.

The model only adheres to this protocol, if communication could be established. The communication component indicates successful setup by issuing an event. If communication could not be established or crossing has not send a *safe*-report in time, the train has to be stopped in a secure distance before the crossing.

TRAIN

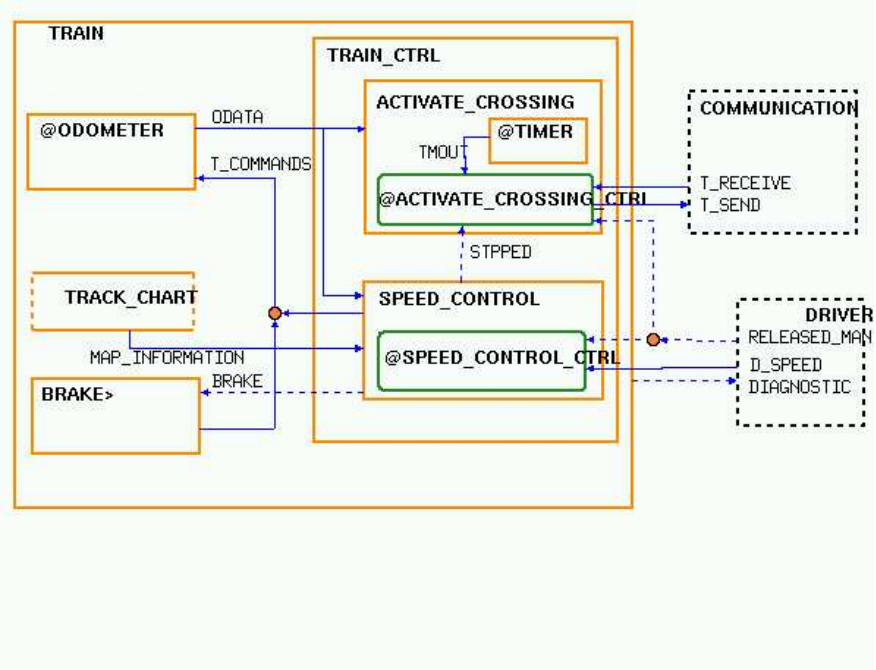


Figure 3.3: Activity `TRAIN`

Activity `TRAIN` consists of several sub-components (figure 3.3), where sub-activity `TRAIN_CTRL` forms the core functionality of the train : Communication with a crossing is performed by activity `ACTIVATE_CROSSING`, which is mainly modeled by statechart `ACTIVATE_CROSSING_CTRL` (figure 3.6) , while controlling the speed of the train is modeled by activity `SPEED_CONTROL`, which is implemented by statechart `SPEED_CONTROL_CTRL` (figure 3.5).

Mini-spec `BRAKE` reacts to the event `BRAKE` which is issued by activity `SPEED_CONTROL` if the train is driving too fast.

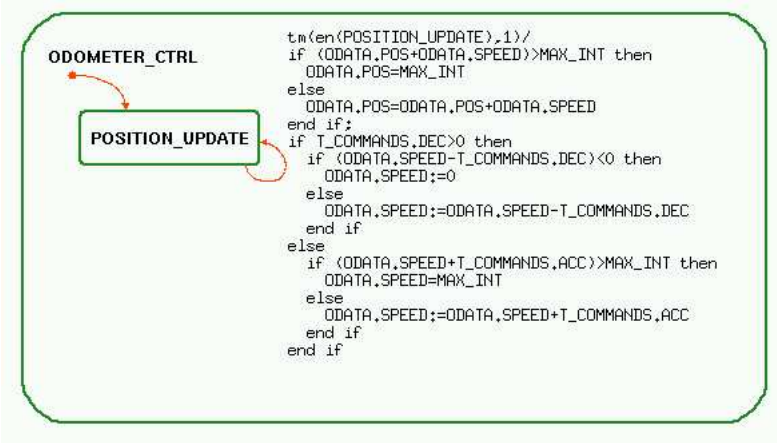


Figure 3.4: Statechart `ODOMETER_CTRL`

Statechart `ODOMETER_CTRL` (figure 3.4) computes time-triggered - once per time-unit - the actual speed and position according to acceleration- and deceleration-commands. Record `ODATA` with its components `SPEED` and `POS` is used in `SPEED_CONTROL_CTRL` to determine the distance from crossing and to compute the maximal allowed speed in order not to overrun an unsecured crossing. Both statecharts `ODOMETER_CTRL` and `SPEED_CONTROL_CTRL` closely interact by performing computations mutually based on the results of computations of the other statechart.

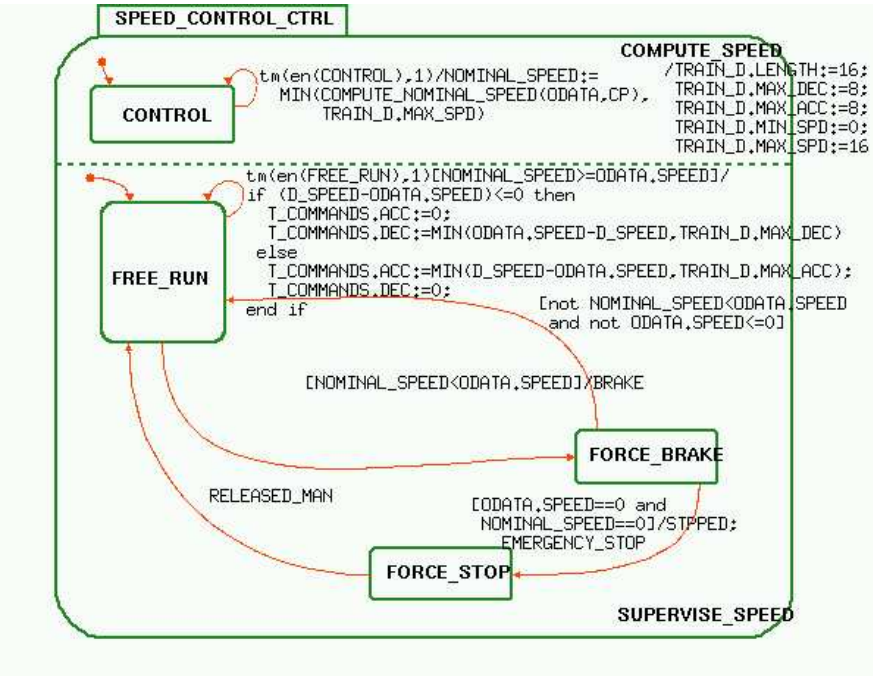


Figure 3.5: Statechart `SPEED_CONTROL_CTRL`

Statechart SPEED_CONTROL_CTRL (figure 3.5) consists of two parallel sub-states; the upper one computes the maximally allowed speed (NOMINAL_SPEED), whereas the lower one checks if the train speed is below the maximum. The actual computation is deferred to the function COMPUTE_NOMINAL_SPEED which takes the dynamic train data (ODATA) and the next control point (CP) as parameters.

A control point is set to a location in front of a crossing and represents an unsecured crossing. Control points must not be passed by a train unless the crossing is secured, i.e. the target speed at reaching a control point has to be zero until the crossing is secured. Control points can be set, i.e. the target speed at its position is set to zero, or deleted, i.e. the target speed at its position is set to the maximal value for this track segment. All control points in a track segment are set when the segment is assigned to a train.

If the driver's preselected speed (D_SPEED) is greater than the train's actual speed (ODATA.SPEED), train is accelerated, otherwise it is decelerated. When the train speed is greater than the permitted speed (NOMINAL_SPEED), state FREE_RUN is exited, FORCE_BRAKE is entered and the brake is activated. Once the train speed is less than the permitted maximal speed, again state FREE_RUN is re-entered. If permitted and actual train speed are both zero, the train has to be stopped, state FORCE_STOP is entered and SPEED_CONTROL_CTRL emits the event STPPED to ACTIVATE_CROSSING_CTRL. Only those stops are indicated by emitting STPPED which result from NOMINAL_SPEED dropping to zero. If the train stops for other reasons, no STPPED event is issued; the train may resume its course on its own without manual release (RELEASED_MAN) by the driver.

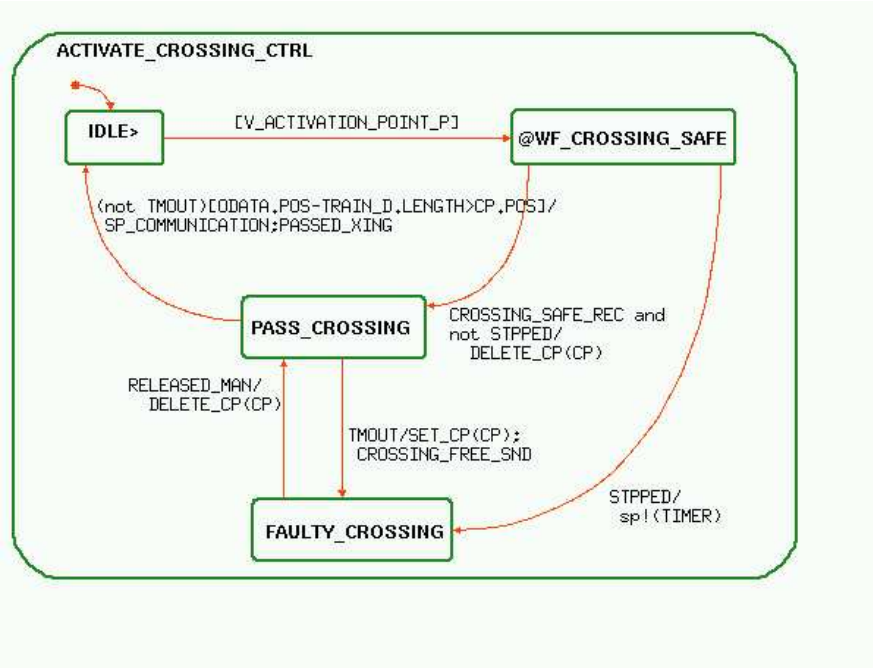


Figure 3.6: Statechart ACTIVATE_CROSSING_CTRL

Statechart ACTIVATE_CROSSING_CTRL (figure 3.6) handles the communication between train and crossing. An important detail is not visible in figure 3.6: State IDLE contains an action which is performed on entering the state. Since store TRACK_CHART is empty in this version of the model, this

entering-action sets the crossing position CP.POS according to an input abstracting from a concrete track chart. Additionally, another input provides the model with the information whether this crossing shall be treated as already regarded. Treating a crossing as already regarded disables the entire protocol between TRAIN and CROSSING and involving COMMUNICATION. In this case, TRAIN can also pass an unsecured CROSSING. We will discuss the problematic effect of this input in section 8.1.

Once the train reaches an activation point – indicated by the condition V_ACTIVATION_POINT_P – state IDLE is exited and WF_CROSSING_SAFE is entered. In hierarchical state WF_CROSSING_SAFE the protocol between train and communication is realized (figure 3.7).

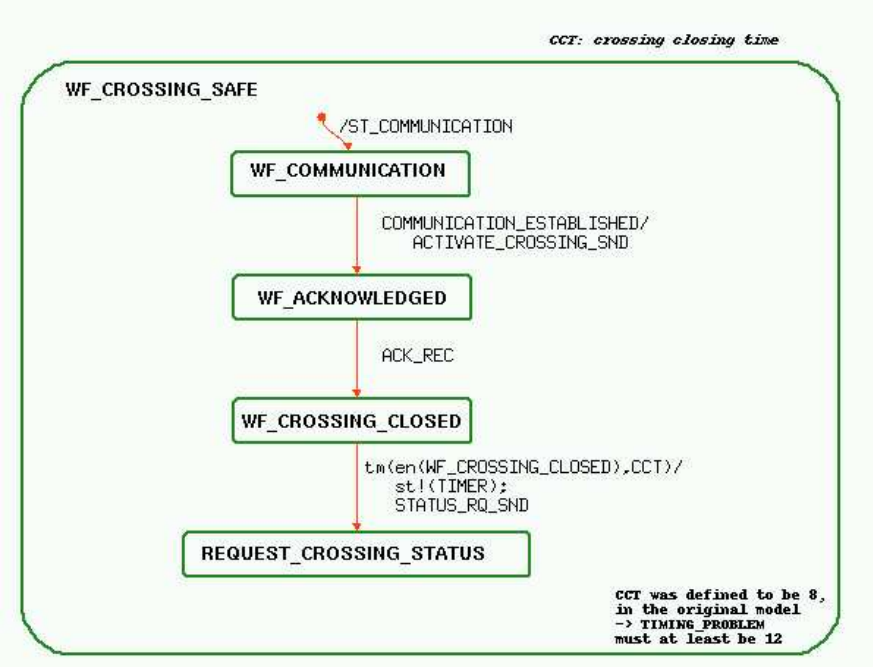


Figure 3.7: Statechart WF_CROSSING_SAFE

On entering WF_CROSSING_SAFE communication is started. Once connection has been established, the crossing is requested to secure itself using event ACTIVATE_CROSSING_SND. On receiving an acknowledgment from the crossing, the train sends a status request after waiting an amount of time (CCT=Crossing Closing Time) which corresponds to the time needed for the crossing to carry out the securing procedure. At this point an important difference exists between asynchronous execution and synchronous execution of the model: When executing the model in the asynchronous execution semantics, CCT=8 is an adequate delay before sending the status request, because CROSSING can under some circumstances complete its securing in this interval. In the synchronous execution semantics, for CCT<12 the crossing will never be able report itself secured. This will be discussed in more detail in section 8.1.

Simultaneously with sending the status request a TIMER is started which supervises TRAIN reaching CROSSING in time. Being in state REQUEST_CROSSING_STATUS of WF_CROSSING_SAFE, TRAIN waits for a status message of CROSSING. The hierarchical state WF_CROSSING_SAFE is exited by either one of two inter-level transitions: Either a safe status report (CROSSING_SAFE_REC) of the crossing is

received or the the train has been stopped due to statechart SPEED_CONTROL_CTRL (STPPED). In case of TRAIN not being stopped by SPEED_CONTROL_CTRL due to a pending status report, TRAIN may pass the crossing - as long as this remains possible before receiving a TMOUT event from TIMER. Hence, a delayed or missing status report forces TRAIN to treat CROSSING to be faulty. Once having entered state FAULTY_CROSSING, TRAIN may pass CROSSING only after manual release (RELEASED_MAN) by the driver.

COMMUNICATION

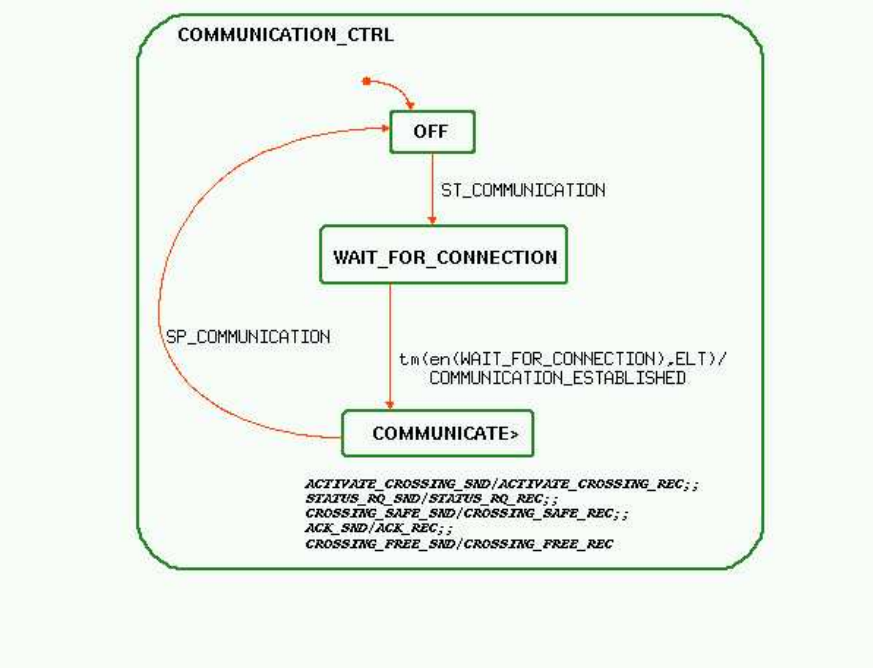


Figure 3.8: Statechart COMMUNICATION_CTRL

Communication between TRAIN and CROSSING is established by a radio link. In top-level activity SYSTEM the information exchange is grouped by information flows T_SEND and T_RECEIVE between TRAIN and COMMUNICATION, respectively, C_SEND and C_RECIEVE between CROSSING and COMMUNICATION. Sent signals (in T_SEND/C_SEND) are forwarded by COMMUNICATION via the appropriate receive channel (C_RECEIVE/T_RECEIVE). Communication has to be explicitly enabled and disabled, which is initiated by the train. Therefore T_SEND also contains the events ST_COMMUNICATION and SP_COMMUNICATION to start and stop the communication. Confirmation of the communication establishment (COMMUNICATION_ESTABLISHED) is contained in T_RECEIVE.

Activity COMMUNICATION is realized by statechart COMMUNICATION_CTRL (cf. figure 3.8).

All interactions except setting up and terminating the connection take place in a static reaction of state COMMUNICATE, which has been added as comment below state COMMUNICATE in figure 3.8. As long as COMMUNICATE is the active state, receive-events (suffix '_REC') are issued for their associated send events (suffix '_SND').

CROSSING

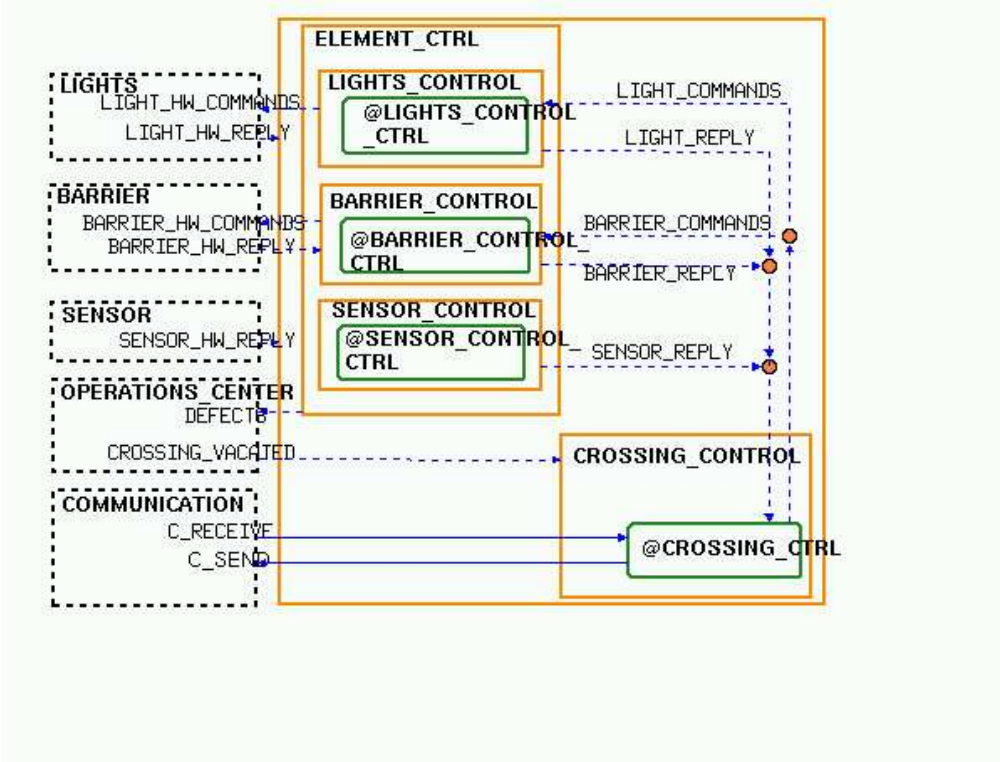


Figure 3.9: Activity `CROSSING`

Activity `CROSSING` (see figure 3.9) contains the overall software control of the crossing which is responsible for coordination of the whole securing process. The component software control (located in activity `CROSSING_CTRL`) interacts with the involved hardware control which is located in activity `ELEMENT_CTRL`. The controlled 'physical' hardware is provided by the external activities `LIGHTS`, `BARRIER`, and `SENSOR`.

The statecharts of the four activities `CROSSING_CONTROL`, `BARRIER_CONTROL_CTRL`, `LIGHTS_CONTROL_CTRL` and `SENSOR_CONTROL_CTRL` interact very closely.

`CROSSING_CTRL`

`CROSSING_CTRL` (figure 3.10) is provided with events and conditions from the three other statecharts of `CROSSING` reporting their respective mode of operation and itself provides the other statecharts with commands.

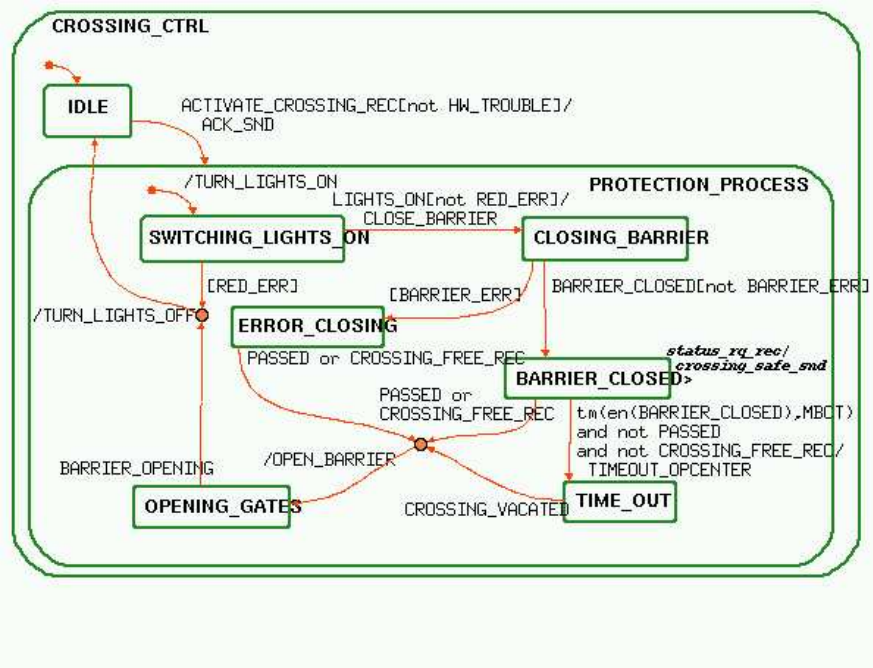


Figure 3.10: Statechart CROSSING_CTRL

The initial state IDLE remains active until CROSSING_CTRL receives the event ACTIVATE_CROSSING_REC. If the system is not in an error condition, state PROTECTION_PROCESS will be entered which encapsulates all further states. A hardware error is present if at least one of the components red light, barrier or sensor operates incorrectly. Condition HW_TROUBLE is defined in CROSSING_CTRL as disjunction of the free input conditions RED_ERR, SENSOR_ERR, and the condition BARRIER_ERR which is provided by BARRIER_CONTROL_CTRL (figure 3.11) and reflects the fact that:

(1) free input condition CLOSED has not become true 'maximal closing time'(MCT) time units after the command CLOSE_BARRIER, or

(2) free input condition CLOSED has not become false again 'maximal opening time'(MOT) time units after the command OPEN_BARRIER

Upon entering state PROTECTION_PROCESS, sub-state SWITCHING_LIGHTS_ON is activated and event TURN_LIGHTS_ON is issued. Statechart LIGHTS_CONTROL_CTRL (figure 3.12) reacts on receiving this event by triggering first the YELLOW light and then the RED light - sent to external activity LIGHTS. Control remains in state SWITCHING_LIGHTS_ON until either receipt of event LIGHTS_ON or detection of a red light malfunction (RED_ERR), with both events originating from statechart LIGHTS_CONTROL_CTRL. Using events TURN_LIGHTS_ON, TURN_LIGHTS_OFF (issued by CROSSING_CTRL) and LIGHTS_ON (issued by LIGHTS_CONTROL_CTRL), a fragile hand-shake protocol with mutual dependences is defined. Moreover, this protocol depends on the free inputs RED_ERR and YELLOW_ERR..

After having successfully secured the crossing by traffic lights, event CLOSE_BARRIER is sent to BARRIER_CONTROL. The system remains in the state CLOSING_BARRIER until either the barrier closing process is completed (indicated by BARRIER_CLOSED) or a hardware error of the physical barriers is reported (BARRIER_ERR). Under normal conditions the control stays in the state BARRIER_CLOSED until

(1) the train has passed (indicated by the event PASSED issued by the activity SENSOR_CTRL), or
(2) the message CROSSING_FREE_REC arrives (originating from TRAIN and indicating that the train was not able to reach the crossing in a reasonable time) , or
(3) the 'maximum barrier closed time' (MBCT) has expired.

On entering BARRIER_CLOSED condition IN_SAFE is set to true indicating a secured state of CROSSING observable at system-level. IN_SAFE is set to false again on exiting BARRIER_CLOSED. Condition IN_SAFE will be referred to in the application examples of Symbolic Timing Diagram verification as the basic observation regarding CROSSING's securing status.

Only as long as state BARRIER_CLOSED is the active state, a status message requested by means of STATUS_RQ_REC is answered with CROSSING_SAFE_SND. In case of the crossing being not able to secure itself - BARRIER_CLOSED is not the active state - , CROSSING simply ignores the status request. An explicit error message is not transmitted. This is a failsafe implementation: the train may pass the crossing only after receipt of CROSSING_SAFE_REC, which is the event associated with CROSSING_SAFE_SND via COMMUNICATION.

BARRIER_CONTROL_CTRL

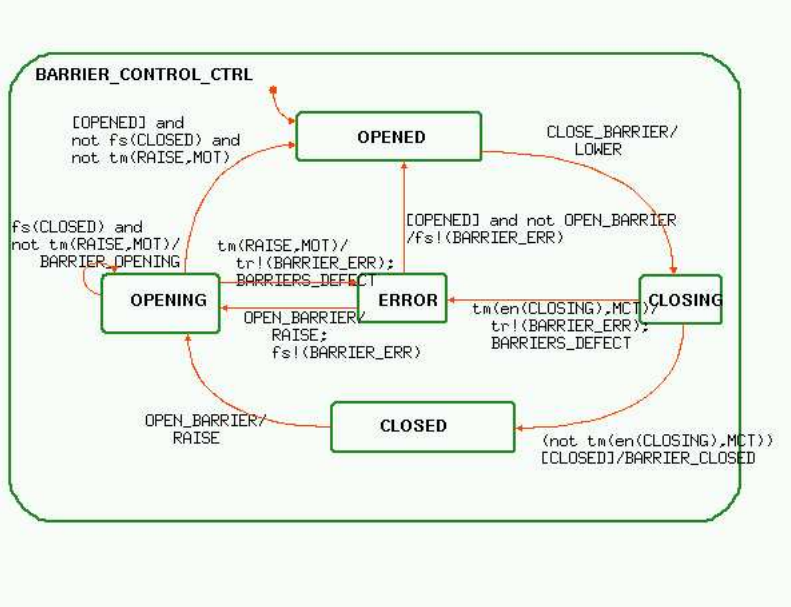


Figure 3.11: Statechart BARRIER_CONTROL_CTRL

Statechart BARRIER_CONTROL_CTRL is aimed at controlling the correct responses of the physical barriers to the issued commands. Free input condition CLOSED indicates the actual status of the barriers. Only if OPENED is the active state of BARRIER_CONTROL_CTRL the demanding event CLOSE_BARRIER from CROSSING_CTRL is translated into an LOWER event triggering the barriers. Accordingly, only if CLOSED is the active state, then the demand OPEN_BARRIER effects RAISE to be sent to the barriers. If free input condition CLOSED is not true 'maximal closing time' (MCT) after LOWER, or if the barriers remain CLOSED 'maximal opening time' (MOT) after RAISE, BARRIER_CONTROL_CTRL enters its state ERROR and indicates a barrier error using condition BARRIER_ERR to CROSSING_CTRL. CROSSING_CTRL

only knows of the actual state of the physical barriers via the condition BARRIER_ERR and the events BARRIER_CLOSED and BARRIER_OPENING controlled by BARRIER_CONTROL_CTRL.

LIGHTS_CONTROL_CTRL

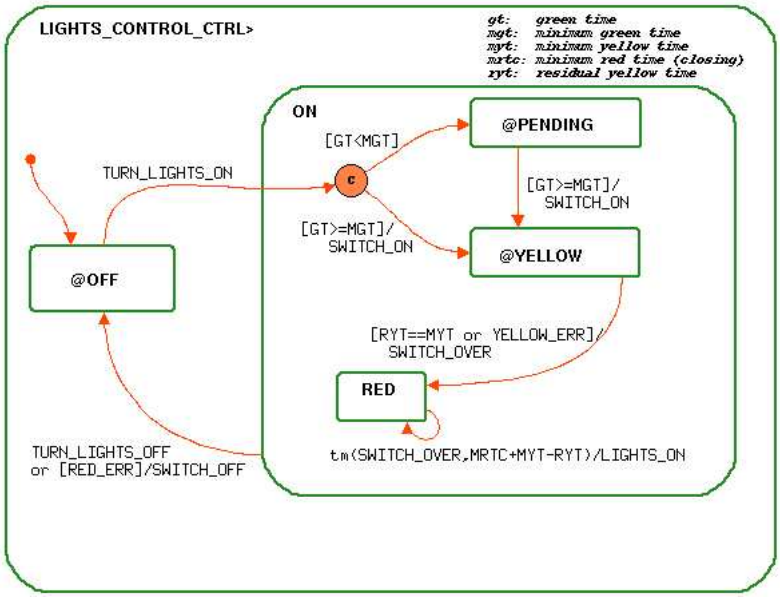


Figure 3.12: Statechart LIGHTS_CONTROL_CTRL

LIGHTS_CONTROL_CTRL controls the physical lights according to the commands TURN_LIGHTS_ON and TURN_LIGHTS_OFF issued by CROSSING_CTRL. Thereby, the intervals between changes of the lights have to adhere to some minimal time limits: MGT - minimum green time - guarantees that cars can cross the crossing between to closures. Local variable GT capturing the green time is increased - by timeout triggers - when staying in states OFF and PENDING, respectively, and reset when entering OFF either initially or from state ON. MYT - minimum yellow time - permits car-driver reaction delays between green and red phase of the lights. Local variable RYT capturing the residual yellow time is reset when entering YELLOW from either OFF or PENDING and increased - by timeout triggers - when staying in state YELLOW. Finally, MRTC - minimum red time (closing) - guarantees a minimal delay between switching the red lights on and indicating LIGHTS_ON, which triggers CROSSING_CTRL to provide BARRIER_CONTROL_CTRL with a CLOSE_BARRIER command.

SENSOR_CONTROL_CTRL

For reasons of completeness, figure 3.13 shows statechart SENSOR_CONTROL_CTRL. This small statechart is responsible for issuing a PASSED event after free input SENSOR_ON indicated a passing train, by first becoming true and then false again. Since PASSED is consumed in CROSSING_CTRL and is the only perception of CROSSING of a train having passed the crossing, SENSOR_CONTROL_CTRL plays a central role in the protocols between CROSSING_CTRL, LIGHTS_CONTROL_CTRL and BARRIER_CONTROL_CTRL.

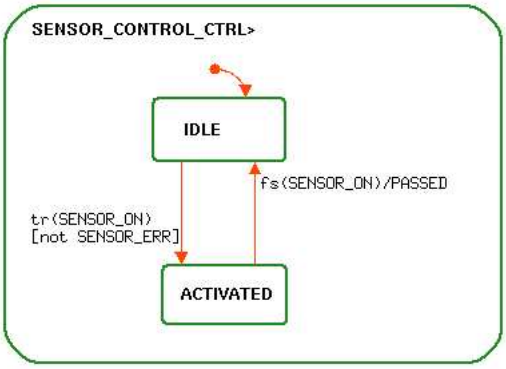


Figure 3.13: Statechart SENSOR_CONTROL_CTRL

Timer Constants of the Case-Study

Tabular 3.2 lists all timer constants of the model together with their value and a short description. As they are used in timeout-events, the listed timer constants determine the real-time behavior of the case-study.

ELT=1	Establishing Lag Time	setup time for communication chanel, delay in COMMUNICATION
MCT=2	Maximum Closing Time	tolerated closing duration, before reporting error
MOT=2	Maximum Opening Time	tolerated opening duration, before reporting error
MBCT=40	Maximum Barrier Closed Time	Maximal duration of closed barrier, before assuming train to be stopped
MRTC=4	Minimum Red Time (Closing)	Red Light must be on for a certain amount of time before lowering barrier
MYT=2	Minimum Yellow Time	duration of yellow light before changes are permitted
MGT=4	Minimum Green Time	minimal duration of green light, before changes are permitted
CCT=8/12	Crossing Closing Time	TRAIN guesses how much time crossing needs in order to close barrier

Table 3.2: Timer Constants in the Case Study

4 Model Checking

Although conceptual modeling can help to circumvent particular problems in the development of embedded systems - like inaccurate, contradictory or erroneous specifications - a conceptual model may itself contain modeling flaws. When used as reference model, correctness of the model under all circumstances is an important issue. Since implementations in later development phases will adhere to the reference model, the quality of the development process depends on the quality of the models. The best evidence for a model conforming to the requirements can be produced by formal proofs that the model behaves as expected under all circumstances. Formal verification techniques can be applied to formally prove, that a model never violates its requirements.

Two well-established approaches to formal verification are *theorem proving* and *model checking*. Theorem-proving is in general an interactive verification approach, where a user is supported in manually performing a proof task. The theorem prover offers and applies rules to the proof obligation. Although modern theorem provers like PVS[1] [OS19] or Coq[2] are endued with also automated strategies, theorem proving remains an interactive verification technique, requiring much expert knowledge about the model and its requirements as well as about the technique itself.

Model checking, on the other hand, is an automatic technique for verifying finite state systems. Given a finite state representation of the system and a specification, the model checking algorithm determines whether the system fulfills the specification. Temporal logic model checking, is a technique developed independently in the 1980s by Clarke, Emerson and Sistla [CES83] and by Queille and Sifakis [QS81]. In this approach specifications are expressed in a temporal logic and systems are modeled as finite state transition systems. An efficient search procedure is used to check if a given finite state transition system fulfills a formal specification.

The semantical model representation, on which model checking procedures are based, are explained in sections 4.1 and 4.2. In section 4.3, we will explain temporal model checking for the branching time temporal logic CTL. Section 4.4 introduces model checking with fairness constraints. A symbolic representation of the model using Binary Decision Diagrams (BDD) is described in section 4.5.

In a variant of the model checking procedure, the specification for verification of the system model is - like the system - represented by an automaton. This technique is applied for linear temporal logic, for which first a so-called *tableau* is generated, which is a formal automaton that captures all runs satisfying the negation of the formula. Model checking is then performed by checking the parallel composition of model and tableau for language emptiness. Model checking linear temporal logic will be briefly described in section 4.6.

Section 4.7 explains, how model checking of invariants can be realized using a modified reachabilty computation.

This simplified model checking technique can also be exploited for verification using so-called

[1] http://pvs.csl.sri.com/
[2] http://coq.inria.fr/

synchronous observers, where the parallel composition of a model with a specification automaton is examined. Here, the specification automaton observes the model and complains if a run of the model violates the specification that is encoded by the observer. We will briefly discuss this approach in section 4.8.

Another simplified model checking technique - bounded model checking using satisfiability checking - is considered in section 4.9. Finally, as a further possibility to tackle verification complexity, section 4.10 briefly discusses abstraction techniques.

4.1 Synchronous Transition Systems

The model checking algorithm requires an adequate system representation, capturing the status and status changes over time according to the possible computations of the reactive system.

A reactive system starts working with an initial valuation of its variables. Periodically, the systems reads some inputs from its environment. Depending on the inputs and the values of its variables, the system changes the values of its variables or leaves them unchanged. Hence, a particular status of a reactive system can be described by the values of the system's variables. Reactions and computations of the system can be described by the changes of these values according to the current status and external stimuli.

Intuitively, state transition systems are a well suited representation of reactive systems. A state transition system with the possible valuations of the system's variables as its states and transitions describing the possible changes of the valuations can be used to formalize this intuition. The execution steps performed by the system can be characterized by the valuations of all its inputs and variables and their changes over time.

We assume that each variable v of the system ranges over a finite set of values Dom_v, called the *domain* of v.

Definition 4.1 (Valuation)

For a given system, let $V = \{v_1, \ldots, v_n\}$ be the set of system variables.

A valuation for V is a function which associates each variable $v \in V$ with a value in the domain Dom_v of variable v:

$\sigma : V \rightarrow Dom_V$, where $Dom_V = \bigcup_{v \in V} Dom_v$,

Let $\Sigma(V) = \{\sigma | \sigma : V \rightarrow Dom_V\}$ denote the set of all possible valuations of V.

If V is obvious from the context we will use the notation Σ instead of $\Sigma(V)$.

Given a subset $V' \subseteq V$, the restriction of a valuation $\sigma \in \Sigma(V)$ to V', denoted by $\sigma \downarrow_{V'}$, is a valuation of V', given by $\sigma \downarrow_{V'}: V' \rightarrow Dom_{V'}$.

■

A snapshot of the valuations at a particular instant of time is interpreted as a state of the model. If all variables range over finite domains then the set of states is a finite set. In general, variables can not change their values arbitrarily. Thus, depending on a state of the model only a specific set of variables can be changed to certain values, i.e. only particular states are reachable from the current state. This relationship between states and their possible successor states determines a transition relation [PS97].

Definition 4.2 (Synchronous Transition System (STS))

A Synchronous Transition System $S = (V, \Theta, \rho)$ consists of the following components:

- V is a set of typed variables.
- Θ is the initial condition. It is a satisfiable assertion characterizing the initial state.
- $\rho \subseteq \Sigma(V) \times \Sigma(V)$ is the transition relation. $\rho(\sigma, \sigma')$ relates a state $\sigma \in \Sigma$ to its possible successors $\sigma' \in \Sigma$ by referring to both unprimed and primed versions of the variables. The unprimed version of a variable refers to its value in σ and the primed version of the same variable refers to its value in the possible successor state σ'. If $\rho(\sigma, \sigma') = true$, we say that state σ' is a S-successor of state σ in STS S.
  ρ is totally defined, i.e. $\forall\sigma \in \Sigma \exists\sigma' \in \Sigma : \rho(\sigma, \sigma') = true$.

■

A *computation* or *run* of STS S is an infinite sequence of valuations $\sigma \in \Sigma(V)$ where each valuation is obtained from the previous by a transition according to transition relation ρ. Hence, a run π of a model can be characterized by an infinite sequence of valuations of the set of model variables:

Definition 4.3 (Run of a System)
Given system S with system variables set V:
A run $\pi : \sigma_0, \sigma_1, \sigma_2, ...$ of STS S is a finite or infinite sequence of valuations $\sigma_i \in \Sigma$ of V, $i \in \mathbb{N}_0$, s.t. the following conditions hold:

- σ_0 satisfies the initial condition ($\sigma \models \Theta$)
- State σ_{i+1} is a S-successor of σ_i, for each $i = 0, 1, 2, ...$

An infinite run is also called a *computation*.

■

Definition 4.4 (Consistency and Viability)
We denote by $Comp(S)$ the set of all computations of STS S. S is called

- *consistent,* if $Comp(S) \neq \emptyset$, i.e. S has at least one computation.
- *viable,* if every finite run π of S can be extended to a computation

■

Temporal logics proved to be useful for the specification of reactive models. The order of events and their relation over time can often be described without referring to time explicitly. Temporal logics extend boolean logics by introducing temporal operators such as "$\mathtt{G}p$" which states that formula p holds *globally* or "$p\mathtt{U}q$" which states that formula p is valid *until* formula q becomes valid. Formulas are interpreted over the set of runs of a finite state system. For example, a run π fulfills $\mathtt{G}p$ if formula p holds for every valuation occurring in π.

In the implementation of the model checking algorithm as presented in [CES83] states and transitions of the model are represented explicitly. For models with small numbers of states this approach works fairly well. For industrial sized real world models the number of states is often too large to be handled explicitly. 1987 Burch, Clarke, Dill and McMillan introduced an algorithm using a symbolic representation of the state transition graphs [McM93, BCM+90]. Since the symbolic approach captures the regularity of the state space of models, it is possible to verify systems with orders of magnitude larger numbers of states than could be handled with the explicit-state algorithms.

4.2 Kripke-Structures

A model checker interprets a temporal logic formula w.r.t. runs of a state transition representation of a model. In order to be suitable for the algorithm, the model representation should capture those properties that must be considered by the algorithm. On the other hand the representation must be easy to handle and abstract from those details which do not affect the verification result.

The state transition graph which optimally fits the needs of model checking algorithms is a *Kripke structure*. Informally, a Kripke structure is a symbolic transition system extended with a labeling function that labels each state with the set of atomic propositions which are true in that state.

Definition 4.5 (Kripke-Structure)
A Kripke structure is a tuple
$M = (AP, S, S_0, R, L)$, where

- AP is a set of atomic propositions
- S is a finite set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is a transition relation. R must be totally defined, that is $\forall s \in S : \exists s' \in S : (s, s') \in R$.
- $L : S \to 2^{AP}$is a function that labels each state with a set of atomic propositions which are true in that state.

Let in the following the set of states S be given by the possible valuations of the system variables, i.e. $S{:=}\Sigma(V)$ for the set V of variables of the system for which Kripke structure M has been build.

A *path* in M is an infinite sequence of states $\Pi = s_0 s_1 s_2 ....$ such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in R$ for $i \geq 0$.

Let $paths(M, s_0)$ denote the set of all paths in M starting with state s_0.

■

Though the transition relation R is defined totally, not every state $s \in S$ needs to be reachable from the set S_0 of initial states by applying the transition relation. The subset of *reachable states* of S can be computed from S_0 and R by starting with S_0 and iteratedly applying R.

We already emphasized the similarity of STS and Kripke structures. In the remainder of this work, we will denote the Kripke structure M obtained from a STS S by $\mathcal{K}(S)$:

Definition 4.6 (Kripke Structure of STS)
Given a synchronous transition system $Sys = (V, \Theta, \rho)$. Let $\mathcal{K} : STS \to' Kripke\,Structures'$ be a function, which constructs a Kripke structure $\mathcal{K}(S){:=}M$ from Sys, where $M = (AP, S, S_0, R, L)$ with

- AP are atomic propositions regarding valuations of the variables in V.
- $S = \Sigma(V)$.
- S_0 are the initial states according to Θ.

- $R = \rho$.
- $L : S \rightarrow 2^{AP}$is a function that labels each state $\sigma \in S$ with a set of atomic propositions which are true in that state.

■

4.3 CTL Model Checking

The specification language for which Clarke, Emerson and Sistla [CES83] presented their model checking algorithm was the propositional branching tree temporal logic CTL (Computation Tree Logic).

In CTL temporal operators are only permitted in combination with universal path quantifiers `A` or existential path quantifier `E`.

Definition 4.7 (CTL) The syntax of CTL is defined by:

1. Every atomic proposition $p \in AP$ is a CTL formula
2. if f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $\texttt{EX}f_1$, $\texttt{EG}f_1$, $\texttt{E}[f_1 \,\texttt{U}\, f_2]$

■

The semantics of a CTL formula is defined with respect to Kripke structures. The propositional connectives $\neg$ (negation) and $\vee$ (or) have their usual meaning. Temporal operators of CTL are

`X` next state - $\texttt{EX}f_1$ holds in a state s if there exists at least one direct successor state of s for which $f_1$holds. $\texttt{AX}f_1 \equiv \neg\texttt{EX}\neg f_1$ holds in a state s if f_1 holds for all successor states of s reachable within one step.

`U` until - is a binary operator. $\texttt{E}[f_1\texttt{U}f_2]$ holds in a state s if $f_1$holds in s and there exists a path along which f_1 holds for all states until eventually f_2 holds for some state s'. $\texttt{A}[f_1\texttt{U}f_2] \equiv \neg\texttt{E}[\neg f_2 \,\texttt{U}\, (\neg f_1 \wedge \neg f_2)] \wedge \neg\texttt{EG}\neg f_2$ holds in a state if f_1 holds in s and f_1 holds for all states on all paths starting in s until eventually f_2 holds for some state on the path.

`F` finally - is an abbreviated notation for $true\,\texttt{U}\,f_1$.

`G` globally - is the dual operator of `F` - i.e. $\texttt{AG}f_1 \equiv \neg\texttt{EF}\neg f_1$. $\texttt{EG}f_1$ holds in a state s if f_1 holds in s and there exists at least one path such that f_1 holds for all states on the path. $\texttt{AG}f_1$ holds in a state s if f_1 holds in s and for all states on all paths starting in s.

A state s of a model M satisfies a CTL formula f denoted by $M, s \models f$, if

$$
\begin{array}{lll}
M, s_o \models p & \text{iff} & s_0 \models p, \text{ with } p \in AP \\
M, s_0 \models \neg f & \text{iff} & s_0 \not\models f \\
M, s_0 \models f_1 \vee f_2 & \text{iff} & s_0 \models f_1 \vee s_0 \models f_2 \\
M, s_0 \models \texttt{EX}f_1 & \text{iff} & \exists s_1 : (s_0, s_1) \in R_M \wedge s_1 \models f_1 \\
M, s_0 \models \texttt{EG}f_1 & \text{iff} & \exists \Pi = (s_0, s_1, ...) \in paths(M, s_0) : \forall i \geq 0 : s_i \models f_1 \\
M, s_0 \models \texttt{E}f_1 \,\texttt{U}\, f_2 & \text{iff} & \exists \Pi = (s_0, s_1, ...) \in paths(M, s_0) : \exists k \geq 0 : M, s_k \models f_2 \wedge \\
 & & \forall 0 \leq j \leq k : M, s_j \models f_1
\end{array}
$$

As abbreviation $M \models f$ is written if $M, s_i \models f$ for all initial states $s_i \in S_o$.

Due to the duality of existential (E) and universal quantification (A) each CTL-formula can be transformed to a formula where none of the temporal operators is negated and negations occur only at the level of propositions. Based on this *negation normal form*, it is sometimes useful to define sub-classes of CTL. The sub-class of negation normal CTL-formulas in which only universal quantification is used is often referred to as ACTL and in contrast ECTL is the sub-class of negation normal formulas which make only use of the existential quantification.

All CTL formulae can be expressed in terms of $\neg f$, $f_1 \vee f_2$, $\mathtt{EX}f$, $\mathtt{E}[f_1 \,\mathtt{U}\, f_2]$ and $\mathtt{EG}f$, due to the duality of existential and universal quantification, e.g. $\mathtt{AF}f = \neg\mathtt{EG}\neg f$. Hence, algorithms for only the cases $\mathtt{EX}f$, $\mathtt{EG}f$ and $\mathtt{E}[f_1 \mathtt{U} f_2]$ are required.

The model checking algorithm determines whether a given Kripke structure M satisfies a CTL formula f for all initial states $s_o \in S_0$. For this purpose, the algorithm first computes the set of states satisfying the CTL formula.

If all initial states of the Kripke structure are elements of the computed set, there can not exist any path through the structure which violates the specification. On the other hand, if there are initial states not satisfying the CTL formula, i.e. which are not in the computed set, then the specification is violated along some path through the structure.

In a first phase the model check algorithm starts to label each state of the Kripke structure with the sub-formulas of f of length 1 which are valid in that state. Successively the states are labeled with valid sub-formulas of increasing length up to the length of the formula. The length of the formula is determined by the total number of operators and operands. If f contains no temporal operators it consists entirely of atomic propositions. For formulas of the form $\neg f$ the states are labeled that are not already labeled with f . For formulas of the form $f_1 \vee f_2$ states are labeled which are already labeled with f_1 or f_2.

- The set of states satisfying $\mathtt{EX}f$ can be determined by searching the states which are labeled with f. All predecessors of these states are labeled with $\mathtt{EX}f$.
- $\mathtt{E}[f_1 \,\mathtt{U}\, f_2]$ is handled by searching those states which are labeled with f_2. From this set of states the algorithm works backwards using the reverse transition relation to find all states which can be reached by a path in which each state is labeled with f_1. The states along this paths are labeled with $\mathtt{E}[f_1 \,\mathtt{U}\, f_2]$.
- In order to label the set of states satisfying $\mathtt{EG}f$, the algorithm searches for a path along which each state is labeled with f.

The labeling algorithm for $\mathtt{EG}f$ is based on the decomposition of the Kripke structure into *strongly connected components*.

Definition 4.8 (Strongly Connected Component)
A strongly connected component C is a maximal substructure such that every state in C is reachable from every other state in C along a path entirely contained in C. C is *nontrivial* if it either consists of more than one state or if it consists of one state with a self-loop.

■

The algorithm determines strongly connected components by the following construction rule of a labeled transition graph :

Rule 4.1 *Derive* $M' = (AP, S', R', L')$ *from* $M = (AP, S, S_0, R, L)$ *where* $S' = \{s \in S | M, s \models f_1\}$, $R' = R|_{S' \times S'}$, *and* $L' = L|_{S'}$. *Note that* R' *may not be total.*

$M, s \models \mathtt{EG} f_1$ *iff*

(1) $s \in S'$ *and*

(2) There exists a path in M' *that leads from* s *to some node* t *in a nontrivial strongly connected component* C *of* M' *[CGP99].*

■

Arbitrary nested CTL formulas f are handled by successively applying the state labeling algorithm to the sub-formulas of f up to the length of f.

After termination of the labeling algorithm for all states s and for all sub-formulas f_i of f holds:

$$M, s \models f_i \;\Leftrightarrow\; s \,\text{is labeled with}\, f_i.$$

4.4 Fairness

In many cases it is desired to do verification with respect to certain properties not expressible in CTL. For example, when verifying a system, one may be interested in only these computations in which a certain requested resource is not blocked forever (because the resource is always already allocated by a concurrent process). There is a need to express constraints on computations, requiring recurring conditions - so-called *fairness* constraints. In [CES83], a modification of the model check algorithm has been presented, which permits the specification of fairness constraints. Fairness constraints can be defined using a set of CTL formulas, representing sets of states of the Kripke structure.

The extension of the model checking algorithm to CTL with fairness constraints makes use of strongly connected components. A strongly connected component is fair, if at least one of the states denoted by the fairness constraints is a state of the strongly connected component. A new proposition Q is introduced and all states are labeled with Q from which there exists a path to a fair strongly connected component. Checking $\mathtt{EX} f$ under fairness constraints is then performed by checking $\mathtt{EX}(f \wedge Q)$. Computing $\mathtt{E}[f_1 \,\mathtt{U}\, f_2]$ under fairness constraints is performed by computing $\mathtt{E}[f_1 \,\mathtt{U}\, (f_2 \wedge Q)]$ instead. $\mathtt{EG} f$ is computed under fairness constraints the same as above (cf. section 4.3) with the only difference that strongly connected component C is also required to be fair.

The semantics of CTL with fairness constraints implemented by the modified model checking algorithm is referred to as *fair semantics.*

There exist a variety of slightly different definitions of fairness constraints. Sometimes fairness constraints are also treated as model properties. For this purpose the definition of Kripke structures is modified to capture fairness as well.

Definition 4.9 (Fair Kripke Structure)

A fair Kripke structure is a 6-tuple $M = (AP, S, S_0, R, L, F)$, where

- AP, S, S_0, R, and L are defined as in definition 4.5, and
- $F \subseteq 2^S$ is a set of fairness constraints.

If $\Pi = s_0 s_1 ....$ is a path of M, then $inf(\Pi) = \{s \,|\, s = s_i \text{ for infinitely many } i\}$ denotes the set of states occurring infinitely often in Π. Π is a fair path if and only if $inf(\Pi) \cap F_i \neq \emptyset$.

■

4.5 Symbolic Model Checking

So far we have considered explicit-state Kripke structures to ease the explanation of the algorithm. The explicit-state representation contains redundant information. The size of the state space for explicit-state models grows exponentially in the number of variables. Algorithms employing explicit-state representations are reasonably applicable only to systems with a very restricted number of variables.

The size of models model checking algorithms can cope with increased dramatically with the symbolic approach introduced by [BCM+90]. The basic idea of the symbolic model representation is that both the set of states of a Kripke structure and its transition relation can be represented by their characteristic functions.

Definition 4.10 (Symbolic Representation)
Since the states S of a Kripke structure $M = (AP, S, S_0, R, L)$ are the possible valuations $\sigma \in \Sigma_V$ of the set of system variables V, the characteristic functions for the initial state set and the transition relation can be expressed as boolean predicates ranging over the valuations of the set of variables V:

Let $\chi_{S_0} : \Sigma_V \rightarrow \mathbb{B}$ be the characteristic function of the initial states $S_0 \subseteq S$, s.t $\chi_{S_0}(\sigma) = true$ iff $\sigma \in S_0$.

Let $\tau_R : \Sigma_V \times \Sigma_V \rightarrow \mathbb{B}$ be the characteristic function of transition relation R, s.t. $\tau_R(\sigma_1, \sigma_2) = true$ if $(\sigma_1, \sigma_2) \in R$.

■

The representation of the transition relation determines how the predicate representing a set of states is transformed to characterize the successor set of states. For these predicate transformers, i.e. predicates denoting predicate transformations, primed variables denote the the value of the variable after application of the transformation.

Using a symbolic model representation permits a more concise definition of the model checking algorithm than in the explicit case: reachability computation as well as the decision procedures for temporal operators can be based on fix-point operations:

Definition 4.11 (Reachability Computation) Let v, u denote valuations of V. The set C_n of states reachable within n steps can inductively be computed:

$C_0 = \{u | \chi_{S_0}(u)\}$

$C_{n+1} = \{u | \chi_{S_0}(u) \vee \exists v \in C_n : \tau_R(v, u)\}$

Due to the finiteness of the model characterized by χ_{S_0} and τ_R, there exist an upper bound, i.e. a fix-point for this iteration:

$\exists n : \forall m \geq n : C_m = C_n$.

■

- The set of states satisfying $\mathtt{EX} f_1$ can be computed by $\exists v : f(v) \wedge \tau_R(u, v)$.
- Fix-point computations similar to the reachability computation are applied to determine the sets of states satisfying $\mathtt{EG} f_1$. The set of states satisfying $\mathtt{EG} f_1$ is computed using the fix-point iteration scheme $f_1 \wedge \mathtt{EX}(\mathtt{EG} f_1)$.
- The iteration scheme used for computing the set of states satisfying $\mathtt{E}[f_1 \,\mathtt{U}\, f_2]$ is $f_2 \vee (f_1 \wedge \mathtt{EX}(\mathtt{E}[f_1 \,\mathtt{U}\, f_2])$ [JEKD90].

Representing sets of states as well as the transition relation by boolean predicates would have been not that successful without a combination with a data-structure called *reduced ordered binary decision diagrams*(ROBDDs) [R.E86, Bry92].

Informally a ROBDD is a rooted, directed acyclic graph with

- one or two terminal nodes of out-degree zero are labeled 0 or 1 respectively
- a set of non-terminal nodes of out-degree two with one outgoing edge labeled 0 and the other labeled 1
- a variable name attached to each non-terminal node
- a linear variable order such that for all paths from the root node to the terminal nodes the order is respected
- no two distinct nodes have the same variable and the same 0 and 1 successors
- no non-terminal node has identical 0 and 1 successors

Besides being an efficient data-structure for boolean formulas, binary decision diagrams permit very efficient manipulations of the represented formulas [Bry92]. Due to maximal sharing of nodes and their linear variable order ROBDDs are a canonical description of a boolean formula - w.r.t. the variable order. As one of the most important consequences, checking for semantical equivalence of two formulas can be done in constant time by checking for isomorphism of their ROBDDs.

The same as for the explicit-state model checking algorithms, there exist modifications of the symbolic model checking algorithms to deal with fairness constraints. Symbolically checking a CTL formula under fairness constraints requires another fix-point computation. This fix-point computation determines for each state in the set of states satisfying the CTL formula whether there exists a fair path containing the state. Only states which satisfy the CTL formula and are located on fair paths of the model are regarded by this symbolic model checking algorithm.

Detailed descriptions of symbolic model checking with and without fairness constraints can be found in, for example, [CGP99].

An important issue of model checking not mentioned so far, is the ability of most available model checkers to provide counterexamples for violated specifications [EOKX95]. Counterexamples are essential in localizing subtle errors in complex designs. Unfortunately, the quality of a counterexamples for a CTL formulas depends on the structure of the formula. A counterexample for an existential specification would require the enumeration of all paths from the initial states showing that there exists no path satisfying the specification. For universal branching time specifications (=ACTL) the counterexample may be a tree[3]. Fortunately, for a large class of specifications the counterexamples are simple paths. While paths are intuitively understandable to the user, the inspection of a tree is much more complicated.

Although the model checking algorithm for CTL is quite efficient, a temporal logic with linear time interpretation offers a more natural semantics. Linear interpretation has the advantage of being interpreted w.r.t. computation sequences. In contrast to CTL, counterexamples for violations of linear time temporal logic formulas are always paths.

[3] E.g. $\mathtt{AX}(p) \vee \mathtt{AX}(q)$ might be violated for a state s having one successor state $s_1$for which $p \wedge \neg q$ holds and another successor state s_2 for which $\neg p \wedge q$ holds. In this case, $\mathtt{AX}(p) \vee \mathtt{AX}(q)$ is violated by a computation tree.

4.6 LTL Model Checking

Temporal operators in CTL have to be used always with either universal or existential quantification. Thus, CTL formulas are always state formulas, specifying computation trees rooted in the state the formula refers to. In contrast to CTL the temporal logic PTL (propositional linear temporal logic) often referred to as LTL (linear temporal logic) is aimed at the specification of paths instead of trees.

Definition 4.12 (LTL) The syntax of LTL formulas is defined by:

1. Every atomic proposition $p \in AP$ is a LTL formula
2. if f_1 and f_2 are LTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $\mathtt{X} f_1$, $\mathtt{G} f_1$, $f_1 \,\mathtt{U}\, f_2$

■

The semantics of $\mathtt{X} f_1$, $\mathtt{G} f_1$, and $f_1 \,\mathtt{U}\, f_2$ is defined with respect to paths Π of a Kripke structure M:

$$
\begin{array}{lll}
M, \Pi \models p & \text{iff} & \Pi_0 \models p, \text{ with } p \in AP \\
M, \Pi \models \neg f & \text{iff} & \Pi \not\models f \\
M, \Pi \models f_1 \vee f_2 & \text{iff} & \Pi \models f_1 \vee \Pi \models f_2 \\
M, \Pi \models \mathtt{X} f_1 & \text{iff} & \Pi_1 \models f_1 \\
M, \Pi \models \mathtt{G} f_1 & \text{iff} & \forall i \geq 0 : \Pi_i \models f_1 \\
M, \Pi \models f_1 \,\mathtt{U}\, f_2 & \text{iff} & \exists k \geq 0 : M, \Pi_k \models f_2 \wedge \forall 0 \leq j \leq k : M, \Pi_j \models f_1,
\end{array}
$$

where Π_i denotes the i-th state in path Π. As in the definition of CTL, $\mathtt{F} f_1$ is used as abbreviation of $true \,\mathtt{U}\, f_1$.

If a LTL formula f holds for all possible paths of a Kripke structure M, often $M \models f$ is written instead of $\forall \Pi : M, \Pi \models f$.

In contrast to CTL the algorithm for model checking propositional linear time logic requires an additional construct, a so-called tableau [CGH97, BCM+90]. A tableau T_f of a LTL formula f is the most general model satisfying f, i.e. each possible path in T_f is a path satisfying f. Consequently, let $T_{\neg f}$ be the most general model of the negation of f contains all paths violating f. Model checking whether a model M satisfies a given LTL specification f can be performed by checking that $M || T_{\neg f}$ has no possible computations. Hence, in order to check satisfaction of a LTL formula f by model M, the product structure of tableau $T_{\neg f}$ and the Kripke structure of M has to be built.

CTL model checking can be applied to this product structure in order to compute the set of states satisfying $\mathtt{EG}\, true$. If there are states satisfying $\mathtt{EG}\, true$, then the product automaton is not empty - i.e. there exists a path in the model for which the negation of LTL formula f is a valid specification. Thus, model M can not satisfy LTL formula f. On the other hand, if the product automaton is empty, then the negation of the LTL formula f does not specify any run of the model - i.e. model M satisfies f.

The tableau associated with a propositional LTL formula f is a fair Kripke structure $T = (AP_f, S_T, S_{T_0}, R_T, L_T, F)$ with AP_f as its set of atomic propositions. Each state of T is labeled

with a set of *elementary* formulas $el(f)$ obtained from f using the following rules[4]:

$$\begin{aligned} el(p) &:= \{p | p \in AP_f\} \\ el(\neg f_1) &:= el(f_1) \\ el(f_1 \vee f_2) &:= el(f_1) \cup el(f_2) \\ el(\mathtt{X} f_1) &:= \{f_1\} \cup el(f_1) \\ el(f_1 \mathtt{U} f_2) &:= \{\mathtt{X}(f_1 \mathtt{U} f_2)\} \cup el(f_1) \cup el(f2) \end{aligned}$$

The set S_T of T is $2^{el(f)}$ and the labeling function is defined by $L_T : S_T \rightarrow 2^{AP_f}$. In order to define the transition relation R_T an additional function $sat(f_1)$ is required which associates for every elementary formula $f_1 \in el(f)$ a set of states in S_T in which f_1 is satisfied:

$$\begin{aligned} sat(f_1) &:= \{s \mid f_1 \in s\}, \text{ where } f_1 \in el(f) \\ sat(\neg f_1) &:= \{s \mid s \notin sat(f_1)\} \\ sat(f_1 \vee f_2) &:= sat(f_1) \cup sat(f_2) \\ sat(f_1 \, U \, f_2) &:= sat(f_2) \cup (sat(f_1) \cap sat(\mathtt{X}(f_1 \, U \, f_2)) \end{aligned}$$

The transition relation R_T of the tableau is built w.r.t. elementary formulas of the form $\mathtt{X} f_i$. R_T is built in a way, s.t. if some state s is labeled by a formula of form $\mathtt{X} f_i$, then the successor states of s w.r.t. R_T are labeled by the elementary formulas $el(f_i)$. Hence, if $\mathtt{X} f_i$ is true at a state, all successors w.r.t.. R_T satisfy f_i . On the other hand, if $\neg \mathtt{X} f_i$ is true in a state, no successor state satisfies f_i.

Using the definition of sat, R_T is formally defined by :

$R_T(s, s') = \bigwedge_{\mathtt{X} f_i \in el(f)} s \in sat(\mathtt{X} f_i) \Leftrightarrow s' \in sat(f_i)$

The set of initial states S_0 of T is the set of states satisfying the LTL specification f. Thus , $S_0 = sat(f)$.

Starting in the initial states, a tableau contains all paths satisfying the LTL formula f for which it was generated according to the construction rule above, but up to now also paths are permitted which do not adhere to fairness requirements. Without restrictions through fairness constraints, a tableau for $f_1 \mathtt{U} f_2$ contains also paths on which $f_1 \wedge \neg f_2$ holds forever, which is not according to the definition of $\mathtt{U}$.

For a tableau T_f of LTL formula f, the fairness constraints F restricting the valid paths of T_f are given by:

$\{sat(\neg(f_1 \, \mathtt{U} \, f_2)) \cup sat(f_2) \mid f_1 \, \mathtt{U} \, f_2 \text{ occurs if } f\}$

In order to check whether the system satisfies a LTL formula f, the product structure $P = (AP_f, S, S_0, R, L, F)$ of the tableau $T_{\neg f} = (AP_f, S_T, S_{T_0}, R_T, L, F_T)$ for $\neg f$ with the Kripke structure $M = (AP_f, S_T, S_{T_0}, R_T, L, F_T)$ of the system has to be built, where:

- $S = \{(s, s') \mid s \in S_M, s' \in S_T \text{ and } L_M(s)|_{AP_f} = L_T(s')\}$. $L_M|_{AP_f}$ is the restriction of the labeling function of M to the atomic propositions of the tableau $T_{\neg f}$. Only the states fulfilling the same atomic propositions of AP_f are regarded. All the other states of M and $T_{\neg f}$ are disregarded in the construction for P.

[4]The algorithm described in the following as well as the notation has been taken from [CGP99].

- $S_0 = \{(s_0, s_0') \mid s_0 \in S_{M_0}, s_0' \in S_{T_0} \text{ and } L_M(s_0)|_{AP_f} = L_T(s_0')\}$. Initial states of the product P are the initial states from M and from $T_{\neg f}$ which fulfill the same atomic propositions of AP_f
- $R((s, s'), (t, t'))$ iff $R_M(s, t) \wedge R_T(s', t')$. Only these transitions exist in P for which a corresponding transition exists in M and in $T_{\neg f}$. R is not necessarily total by construction. If R is not a total relation, all states which do not have successors are removed from S and R is restricted to the remaining states.
- $L((s, s')) = L_T(s')$. The labeling only labels states with 2^{AP_f}
- $F = \{(s, s') \mid s' \in F_T\}$. The product structure inherits the fairness constraints of the tableau. For this construction it is assumed that the model structure itself has no fairness constraints.

Using the product-structure $P = M \cap T_{\neg f}$, which is built from the tableau $T_{\neg f}$ for the negation of the formula f to be checked and from the Kripke structure M of the model, now $M \models f$ can be checked by checking $M \cap T_{\neg f}$ for emptiness. If there exist runs of M, satisfying LTL-formula f, then these runs can not be paths of $T_{\neg f}$. Hence, $M \cap T_{\neg f} = \emptyset$ if and only if all runs of M satisfy f and thus are no runs of $T_{\neg f}$.

4.7 Invariance Checking

In practice some important properties can be specified by invariants, i.e. formulas of the form: $\mathtt{AG}(\phi)$, where ϕ is a boolean formula containing no temporal operators. Provided that no fairness constraints have to be fulfilled, checking whether a boolean formula is an invariant of a model does not require the full effort of temporal logic model checking. An invariant must hold on all possible paths of the model, including the initial states. In order to find a violation of $\mathtt{AG}(\phi)$ it suffices to find a path from an initial state of a model to a state in which ϕ is violated. Since deciding the truth of a boolean formula for a particular valuation of its variables is trivial using ROBDDs, verification of invariant properties can be performed by symbolic model checking very efficiently using a *modified reachability computation* algorithm *[VIS96a, VIS96b]*.

Let C_i be the set of states reachable with the i-th application of the transition relation to the initial states as in definition 4.11. For each of the sets C_i of the fix-point computation, the validity of boolean formula ϕ is checked. If ϕ is not true for some C_i, the formula can not be an invariant of the model. On the other hand, if ϕ holds for all C_i - up to the set of all reachable states - , then $\mathtt{AG}(\phi)$ is a valid invariant of the model [CGP99].

Only for valid invariants the model checker has to perform the entire reachability computation, violations of specifications are detected before completion. Hence, the worst case complexity for invariance checking is that of computing the set of reachable states of the model.

Fairness constraints can not be regarded by this algorithm, since these specify paths through the set of reachable states, which needs not to be computed in advance for invariance checking.

4.8 Verification using Synchronous Observers

When no fairness constraints have to be regarded in order to check a desired property of a model, *invariance checking* can also be applied to more complicated specifications by encoding them as *observers*. Basically, an observer is an automaton which is aimed at following and assessing the

runs of a model. Running in parallel to a model the observer then indicates whether an observed run of the model conforms to the specification, for which the particular observer has been built.

The parallel composition of model and observer is built in such a way that the observer only observes the behavior of the model without restricting it in any way. Verification is then performed by "checking that the parallel composition of the program and its observer never causes the observer to complain" [HLR93].

Ideally, an observer should be designed in such a way that its single output changes its value immediately whenever a computation of the model violates the encoded specification.

The verification approach using synchronous observers has - to our knowledge - first been presented by Halbwachs, Lagnier and Raymond [HLR93]. There, the approach is explicitly restricted to safety properties, which can be checked without considering fairness constraints: "... the desired properties of a program can be easily and modularly expressed by means of an observer, i.e., another program which observes the behavior of the first one and decides whether it is correct. Thus, the same language is used to write the program and its desired properties. The verification then consists in checking that the parallel composition of the program and its observer never causes the observer to complain. This verification can often be performed by traversing the finite control automaton built by the compiler" [HLR93].

In general, the critical properties of a reactive system are often only required to hold, provided that the environment also behaves correctly, that is, under some assumptions about the environment.

Observers can also be used to express required properties of the model environment. Using observers in combination with invariance checking, assumptions can be expressed by referring to the observer outputs on the left hand side of an implication which has to hold invariantly.

Since observers can be implemented using the same description language as the one used for the model itself all facilities of the description language can be used to capture the specification. Hence, it is possible for example to realize counters in order to count particular observations. In contrast, counting is impossible using CTL or LTL.

Even though, especially the combination of invariance checking and observers shows very promising complexity results even for complicated specifications, application of observers is obviously not a-priori restricted to invariance checking. Encoding a specification into an observer using the same language as is used to describe the model to be verified - and performing verification on the basis of the parallel composition of model and observer - can of course also be used in combination with temporal logic model checking.

In chapter 6 two areas of application will be discussed in detail: verification (1) using predefined observer patterns provided by a library, and (2) using observers automatically generated from graphical specifications provided by Symbolic Timing Diagrams.

4.9 Bounded Model Checking using Satisfiability Checking (BMC)

State-of-the-art symbolic model checkers are able to handle models with hundreds of state variables. Symbolic model checking is based on a boolean encoding of models, using ROBDDs (or short BDD). Although binary decision diagrams permit very efficient manipulations of the represented formulas, BDDs generated during model checking often become too large for currently available computers when verifying large systems. In addition, finding an optimal ordering of variables plays an important role since the size of ROBDDS depends on the variable order. Reordering of ROBDDs after application of boolean manipulations can drastically decrease the size of the representation.

Unfortunately, reordering of ROBDDs is time consuming and only based on heuristics. For some examples no space efficient ordering exists.

Procedures deciding satisfaction of propositional boolean expressions in non-canonical form avoid the potential state explosion problems of ROBDDs.

Biere,Clarke, Cimatti, Fujita and Zhu proposed to apply satisfiability checking for falsification [BCC+99]:

> The basic idea is to consider counterexamples of a particular length k and generate a propositional formula that is satisfiable iff such a counterexample exists.

Satisfiability checking can be used to check for reachability of a particular variable valuation within the first k steps of a system. By applying the transition relation k-times to the initial states, a model of the k-th step of the system can be derived. Hence, the conjunction of the first k applications of the transition relation to the initial states yields a model of the first k steps. Using this *unrolled* model, satisfiability of a particular valuation is checked w.r.t. bound k:

Essentially, there are two steps in bounded model checking. In the first step, the sequential behavior of a transition system over a finite interval is encoded as a propositional formula. In the second step, that formula is given to a propositional decision procedure, i.e. a satisfiability solver, to either obtain a satisfying assignment or to prove there is none. Each satisfying assignment that is found can be decoded into a state sequence, which reaches states of interest. In bounded model checking only finite length sequences are explored [CBRZ01].

Formally, bounded model checking is performed as follows:

Definition 4.13 (Bounded Model Checking)
Given :

- a Kripke structure $M = (AP, S_0, S, R, L)$, with AP ranging over the variables $V = \{v_1, \ldots, v_n\}$
- an ECTL formula ϕ and
- a user supplied bound k

Let $V_i = \{v_{1_i}, ..., v_{n_i}\}$ be an instance of the set V of variables that is built by substituting each variable $v_j \in V$ with a new variable v_{j_i}, i.e. $V_i{:=}\{v_1, ..., v_n\}[v_{1_i}/v_1, ..., v_{n_i}/v_n]$. Let T_R be a boolean predicate ranging over the atomic propositions for two instances V_i, V_{i+1} of V, s.t. $[\![T_R(V_i, V_{i+1})]\!](\sigma_i, \sigma_{i+1}) = true$ iff $\tau_R(\sigma_i, \sigma_{i+1}) = true$ (cf. definition 4.10 on page 50). The *unrolled transition system* $[M]_k$ is then characterized by the predicate:

$$[M]_k{:=}\begin{cases} X_{S_0}(V_0) & \text{if } k = 0 \\ X_{S_0}(V_0) \wedge \bigwedge_{i=0}^{k-1} T_R(V_i, V_{i+1}) & \text{if } k > 0 \end{cases}, \text{ where}$$

$X_{S_0}(V_0)$ is a predicate representing the characteristic function χ_{S_0} of the initial states, i.e. $[\![X_R(V_0)]\!](\sigma) = true$ iff $\chi_{S_0}(\sigma) = true$.

From ϕ a bounded predicate $[\phi]_k$ is formed regarding the instances $V_0, ..., V_k$ of V, which is satisfiable if and only if ϕ is true along a path of length k in $[M]_k$.

The predicate $[M,\phi]_k{:=}[M]_k \wedge [\phi]_k$ can be checked for satisfiability:

$$[M,\phi]_k \text{ is satisfiable, iff } \exists(\sigma_0,...,\sigma_k) : [\![[M,\phi]_k]\!](\sigma_0,...,\sigma_k) = true, \text{ i.e.}$$

$$\exists(\sigma_0,...,\sigma_k) : \left([\![X_{S_0'}(V_0)]\!](\sigma_0) \wedge \bigwedge_{i=0}^{k-1} [\![T_{R'}(V_i,V_{i+1})]\!](\sigma_i,\sigma_{i+1}) \right) = true, \text{ with}$$

$\sigma_i : V \rightarrow Dom_V$, where $X_{S_0'}(V_0)$ is a predicate representing the characteristic function $\chi_{S_0'}$ of the initial states S_0' of $[M,\phi]_k$, and $T_{R'}(V_i,V_{i+1})$ is a predicate representing the characteristic function $\tau_{R'}$ of the transition relation R' of $[M,\phi]_k$w.r.t. two instances V_i, V_{i+1} of V.

■

In order to verify validity of an ACTL formula ϕ for all paths Π_k of length k, it is sufficient to show that $[M,\neg\phi]_k$ is not satisfiable. If, on the other hand, bounded model checking finds a sequence of valuations $[\sigma]_k$ for which $[M,\neg\phi]_k$ is satisfied, also M violates ϕ within the first k steps.

Since only paths of length k are considered, bounded model checking under-approximates the system under consideration:

- Bounded model checking finds counterexamples very fast. For some use-cases, finding counterexamples is arguably the most important feature of model checking.
- Counterexamples showing violations of $[\phi]_k$ are of minimal length. This feature helps the user to understand a counterexample more easily.
- Since bounded model checking uses much less space than BDD based approaches, it can handle much larger systems than symbolic model checking . Unlike BDD based approaches, bounded model checking does not depend on variable ordering, and hence requires no time consuming dynamic reordering.

The advantages of bounded model checking over symbolic model checking are very promising for all use-cases where violation of formulae is highly probable. On the other hand, applying under-approximation is restricted to only these cases, where a ultimative evidence of positive validity is not required.

4.10 Abstraction

Often an interesting way to attack the verification complexity is the application of abstractions. The principle idea is to consider a more abstract but less complex model than the concrete one, in order to obtain a desired verification result. Abstractions can be achieved by restricting the possible runs of the model, e.g. by setting inputs to constants or restricting domains of variables of the model. On the other hand, abstractions can be obtained by permitting more possible runs of the model, e.g. by weakening conditions of the model. Also, simple replacement of e.g. large domains of variables with enumerated domains is an abstraction - capturing all possible values of the variable but requiring less bits for its representation than the original domain.

Abstractions should be applied to the model w.r.t. the verification context, i.e. the expected result. Only abstractions should be applied which permit a conclusion from the result of the abstract verification task to the un-abstracted model. Given a model M and an abstraction Abs, where $M_A{:=}Abs(M)$, we distinguish:

over-approximation M_A has a less restricted behavior than M, or in other words $M_A$has strictly more possible runs than M. For an ACTL formula ϕ holds: $(M_A \models \phi) \Rightarrow (M \models \phi)$. From $M_A \not\models \phi$ no conclusion about $M$and ϕ is possible. For ECTL formula ϕ, we have: $(M_A \not\models \phi) \Rightarrow (M \not\models \phi)$, but $(M_A \models \phi) \not\Rightarrow (M \models \phi)$.

exact approximation M_A shows an indistinguishable behavior to M in terms of observations. (bisimulation argument)
As one of the simplest examples we might be only interested in the sign of an integer variable. While the domain of the variable could be chosen to be unbounded for the original model M, it is sufficient to distinguish the cases *negative,zero,positive* for the value of the variable for M_A. Using an appropriate abstraction mapping, a large domain can be represented by a much smaller domain containing at least enough values to distinguish all relevant cases.
For an exact approximation and for a CTL formula ϕ holds: $(M_A \models \phi) \Leftrightarrow (M \models \phi)$.

under-approximation M_A has a more restricted behavior, i.e. has strictly less possible runs than M. For an ACTL formula ϕ holds: $(M_A \not\models \phi) \Rightarrow (M \not\models \phi)$. For an ECTL formula ϕ holds: $(M_A \models \phi) \Rightarrow (M \models \phi)$.

We will consider some abstraction techniques integrated with the STVE in section 7.2.

4.11 Verification Tools integrated with STVE

The STVE is integrated with the model checker VIS [VIS96a, VIS96b], which itself uses the RoBDD-package CUDD by Fabio Somenzi [Som98]. VIS is a state-of-the-art symbolic model checker integrating various different formal verification procedures, such as : invariance checking, CTL model checking, LTL model checking, language emptiness checking as well as combinational and sequential equivalence checking. For the context of this work, mostly invariance checking is the preferred technique for robustness analyses, for formal debugging, pattern verification as well as for specification verification using STDx .

Also integrated with the STVE is a bounded model checker, which is based on the SAT solver Prover Plug-In, trademark of Prover Technologies AB in Sweden, the United States and other countries. As already stated above, application of bounded model checking is in particular useful, when a violation of the verified property is the expected outcome of a verification activity. This is the case in all drive-to-checks as well as for robustness analyses in early phases of development (cf. sections 6.1, 8.1).

The user can choose either VIS or the bounded model checker as verification engine for robustness analyses, formal debugging and also for pattern verification. Specification verification using STDx is restricted to usage of the VIS model checker (cf. sections 6.5, 6.3, and 7.4), because only a complete verification procedure is appropriate for verification of fulfillment of requirement specifications.

5 System Representation for Formal Verification

This chapter is concerned with a compositional representation of STATEMATE models for verification. Section 5.1 gives an overview of a compositional semantics for STATEMATE models and brings key aspects of this semantics regarding compositional verification into focus. In section 5.2 compositional synchronous transition systems are introduced, which provide the basis of the formal semantics definition. Real time aspects regarding the verification of STATEMATE models are considered in section 5.3. Finally, section 5.4 explains the languages System Modeling Interface (SMI) that is used for the representation of behavior and System Structuring Language (SSL) which serves as language for representing the structure of system models. In this section also the integration of Symbolic Timing Diagram specifications with the structural representation of STATEMATE models is explained. This integration forms the basis of compositional verification, which will be discussed in chapter 7.

5.1 A Compositional Semantics for Statemate Models

In STATEMATE, simulation can be carried out for each level of the activity hierarchy. The simulation depends on the selected scope and is in this respect not compositional, i.e. execution of internal activities in isolation is not consistent with the simulation of the entire model: When using the asynchronous time model for simulation, the simulator provides the simulated activity with input stimuli in stable states only.[1]In contrast, when simulating an *internal activity in the context of the entire system*, events generated by sibling activities - i.e. other internal activities within the system but not belonging to the chosen simulation scope - are sensed between stable states at a granularity of single steps.

Thus, the immaculate and unaltered behavior of an internal activity can be observed only when simulating the activity within the scope of the entire system, i.e. in its *concrete* environment.

Since simulation is a key issue for the role of models as executable reference specifications in a model based development process, the definition of a formal semantics has to conform to the execution of models as supported by the simulation tool.

STATEMATE's simulation is based upon the 'basic step algorithm', as presented informally by Harel and Namaad in [HN96](cf. section 3.2). The algorithm defines how a new status is computed based upon the actual status and external stimuli. The actual status is determined by the states, events, conditions and variables of the model.

Thereby, the assumed flow-directions of events, conditions and data-items - local, input, output - are of great importance for the execution of a model:

[1]In interactive simulation the user can inject external changes also between steps of a super-step. This feature is out of scope of a formal semantics and is not supported in the verification tool set.

- When using the asynchronous time model for the simulation of a model, external events and changes of values are sensed only at stable states, while local events and value changes are sensed at every step. Hence, an internal activity of a system, in general behaves differently when simulated in isolation compared to embedded simulation with the system containing the activity. Variables and events of the system - which are not controlled by the considered but by sibling activities - are treated as *external* when simulating the activity in isolation, although they are, at all, *local to the system*.

- Reaching a stable state in the asynchronous interpretation is a dynamic property of the *chosen simulation scope*. Since changes of data-items or events which are controlled by the simulation environment are sensed only at stable states, an activity may become stable when simulated in isolation. The interaction with sibling activities belonging to the system might cause *super-step divergence* when simulating the same activity as part of the system.
  The simulation tool classifies events into 'internal' and 'external' depending on the chosen simulation scope. An asynchronous system model will reach a stable status only if no internal activity has to process pending internal events. Otherwise the actual super-step is continued - possibly forever - unless no more internal events are to be processed. Hence, the choice of a simulation scope has a significant impact on dynamic stabilization.

- *Sharing variables* with other activities of the system, in general leads to different behavior of an activity when simulated in isolation compared to its behavior when simulated as part of the enclosing system. When a shared variable is changed by another sharing activity, it changes its value nondeterministically from the point of view of the considered activity.

For the verification of systems modeled with STATEMATE, a formal semantics is required which is compositional w.r.t. the decomposition of systems into subsystems. It must be ensured, that considering a subsystem in isolation yields results which are consistent with the behavior of the entire system.

Compositionality has been a key topic for the definition of the reference semantics presented by Damm, Josko, Hungar and Pnueli in [DJHP97]. Their compositional semantics forms the basis of a model representation which is rich enough to model the STATEMATE parallel composition by intersection of the infinite traces generated by the involved components. The compositional semantics has been implemented in the model representation realized and presented by Brockmeyer [Bro99].

The compositional semantics is based on the definition of distributed protocols to which the compositional models of internal activities contribute. This way, all dynamic properties of the entire system are consistent with the properties of internal activities considered in isolation. In particular, distributed protocols for accesses to shared variables, for scheduling of controlled activities as well as for dynamic stabilization w.r.t. the asynchronous semantics are defined by the compositional semantics. The possible interactions with sibling activities within the system boundaries are taken into account even when considering an activity in isolation.

"Roughly, compositional models have to provide room for padding arbitrary (but still "legal") environment interactions into computations of a component. Alternatively, the construction of compositional models can be phrased as a requirement on the model to support a sufficiently rich class of *observables* for assumption-commitment style reasoning to be complete" [DJHP97].

While the informal semantics definition of Harel and Namaad lacks a concept of activity interfaces, one of the key concepts of the semantics definition is that of a *formal interface* for each activity

w.r.t. the system boundaries, which can only be obtained from a data- and control-flow analysis w.r.t. the entire system.

For an activity A, all events - including implicit events -, conditions and data-items, as well as auxiliary events and conditions for controlling the activation status of A, are treated as *variables* of A. These variables are classified regarding their usage inside and outside of A. Variables not used outside of A are *local* variables of A. Otherwise, if a variable is used inside and outside of A, it contributes to the observable behavior of A and hence has to be classified as *observable* of A. For each observable of A the interface specifies its direction, i.e. whether this observable is an *input* or an *output* of A. According to the compositional semantics, bidirectionally used objects ('*inout*') are split into directed copies. Shared variables are modeled using additional components maintaining the concurrent accesses to the variable. Consequently, the direction of each interface variable is either '*input*' or '*output*'.

In the synchronous semantics, inputs - regardless if driven from outside the system-model or from other components of the system - are read at the beginning of the step and all components perform a step in parallel.

Regarding the asynchronous execution of STATEMATE models, observables driven from outside the considered activity - but within the system - must be treated differently from observables that are controlled by the environment of the system. Therefore, inputs *from sibling activities* within the system are marked as *fast* inputs of the interface, while inputs *controlled by the environment of the entire system* are *slow* inputs.

Fast inputs can change at every step of the execution, while slow inputs can only change in stable states of the system. Indeed the distinction between externally and locally generated events and value changes of variables is paramount for the definition of a super-step: it terminates, if no further steps can be taken on the basis of events or changed values generated locally in the system. When considering an internal activity in isolation, events from other activities or variables written by other activities of the system have to be read from fast inputs - according to the formal interface.

Since asynchronous stabilization is a dynamic property of the entire system, becoming stable is modeled by the compositional semantics using an explicit distributed protocol between the activities of the system. Enabling the activities to take part in this protocol, the activity representations and their interfaces are extended with observables indicating local stability and sensing the willingness of their neighboring activities to become stable. The contribution of one component to this distributed protocol is described on page 65.

As example, consider figure 3.2 on page 31. Activity `SYSTEM` consists of three sub-activities : `TRAIN, COMMUNICATION` and `CROSSING.` From the perspective of internal activity `TRAIN` the environment consists partly of the environment of top-level activity `SYSTEM` on the one hand, since `TRAIN` is triggered by *external* stimuli provided by the environment. On the other hand, also `COMMUNICATION` and `CROSSING` are part of the environment of `TRAIN.` Inputs to `TRAIN` provided by the environment of top-level activity `SYSTEM` have to be represented by *slow* inputs. In contrast, variables written by activity `COMMUNICATION` and consumed by `TRAIN` have to be modeled by *fast* inputs in the interface of `TRAIN`..

Slow inputs which can change only at Super-step-boundaries are modeled by introducing a `local` buffer for each slow input. These buffers have to be explicitly updated with real inputs in stable states.

Slow inputs are then accessed by the model only by referring to these buffers. In contrast, communication between internal activities of the model is modeled as fast communication in the compositional semantics.

We can only give an informal description of activity interfaces here. We refer to [DJHP97] for the complete and formal definition.

Definition 5.1 (Interface of an Activity)
The definition of $intf(A)$ takes the data- and control-flow of the entire system containing A into account. Thereby, the interface is partitioned into three parts:

Explicit Part: the explicit part consists of user-defined events, conditions, and data items which are used in graphical or textual declarations of A:

- $intf(A)$ contains all data-items, conditions and events as inputs which are used read-only in A.
- Events, conditions and data-items which are only written in A (but not read) are outputs of $intf(A)$.
  If conditions or data-items can be written also by another activity of the system, they may be affected by a write/write conflict. In this case, they are split into an adequate input and output portion, s.t. conflicting accesses are resolvable. Also the model representation of activity A for verification has to take this split into account. $intf(A)$ contains an input as well as an output for this *shared* element. In order to distinguish the input from the output, both have to be named differently[2].
- Events, conditions and data-items which are read and written in A can be either locals of A - if they are not used in other activities - or are shared with other activities of the system. If an element is read by other activities, it must at least be an output in $intf(A)$. If it can also be written by another activity, an input must be provided which can prevail over the local assignment to the element. A shared condition or data-item must be represented in $intf(A)$ again by an input as well as by an output, such that the effects of conflicting assignments are preserved.

Implicit Part: the implicit part consists of the scheduling primitives used in A, plus the implicit events associated with interface elements of the explicit part:

- For all *scheduling events* supported by STATEMATE - e.g starting an activity, stopping an activity, e.t.c. - adequate in- and output-events are part of $intf(A)$ according to the control flow of the system (cf. page 63 et seq.).
- For each condition or data-item x of $intf(A)$, the implicit events $written(x)$, $read(x)$, $changed(x)$ with same direction as x itself are part of $intf(A)$ - indicating the respective accesses to x according to the simulation semantics.

Auxiliary Part: the auxiliary part of the interface comprises

- If A is an activity of an asynchronous model, then also the set of events and conditions required for the stabilization protocol is part of $intf(A)$ (cf. page 65 et seq.).

[2] Accesses to shared data-items are controlled by so-called monitors, which are separate components that explicitly resolve access conflicts

- Since A can refer to simulation time in order to schedule actions and trigger transitions, also advance of simulation time has to be regarded by A. For simplicity, it is assumed throughout this work that time advances in commensurate units. For the synchronous execution semantics it is assumed, that time increases by one time unit with every step. In contrast, for the asynchronous execution semantics time is assumed to increase by one time unit whenever the model becomes stable in terms of the asynchronous stabilization protocol. For asynchronous models a dedicated output `SUPER_SYNC` - indicating stability - is part of $intf(A)$. Hence, `SUPER_SYNC` can be interpreted as *tick of a clock capturing the simulation time*. The impact of stabilization on the interpretation of time will be discussed in section 5.3.

- W.r.t. the chosen execution semantics, each input of the interface is either marked as *slow* or *fast*, according to the location of its driver - i.e. the controlling sources in the embedding system or in the environment of the system.

■

With respect to their fundamental importance for compositional verification, we will in the following focus on the distributed realization of the activity scheduling protocol as well as of the asynchronous stabilization protocol.

In particular, scheduling of controlled activities has always to be regarded in compositional verification when controlled activities are involved in the considered properties. When verifying a controlled activity in isolation, adequate assumptions have to be provided in order to consider the activity in the correct activation state.

Regarding the asynchronous semantics, stabilization has a great impact on verification, because time advances only in stable states. Even though time might not be explicitly considered, super-step divergence of a system is at least a highly undesired behavior. Since the system accepts inputs from the environment only in stable states, super-step divergence has in general to be treated to be a severe failure of the system.

Scheduling of Controlled Activities Using a Distributed Protocol

The mechanisms for scheduling of controlled activities have been discussed in section 3.1 (cf. tabular 3.1). According to the compositional semantics, the component model of a controlled activity has to be capable of being in all possible activation states. W.r.t. compositionality, the model of a controlled activity has to assure, that e.g. reactions on inputs can only take place, if the activity has been activated and has neither been stopped nor suspended by its controlling activity. The scheduling-primitives with which an internal activity is controlled by a controlling activity have to be provided as fast inputs of the controlled activity's interface.

Continuing the example above, as activity `TIMER` (cf. figure 3.3 on page 32) can be started, stopped, suspended and resumed by its controlling activity `ACTIVATE_CROSSING_CTRL`, `TIMER` it is a controlled activity[3] in activity `TRAIN` . Events and conditions indicating the state of activation have to be signaled back to `ACTIVATE_CROSSING_CTRL`.

The events and conditions involved in this protocol - and therefore provided in the interface for `TIMER` - are:

[3] The controlling statechart `ACTIVATE_CROSSING_CTRL`, which itself is not controlled by another activity, is automatically started with the initial activation of `TRAIN`.

START, STOP, SUSPEND, RESUME	input	event
STARTED, STOPPED	output	event
HANGING, ACTIVE	output	condition
ACTION	local	event

Figure 5.1 illustrates the activation states of a controlled activity in a statechart-like manner.

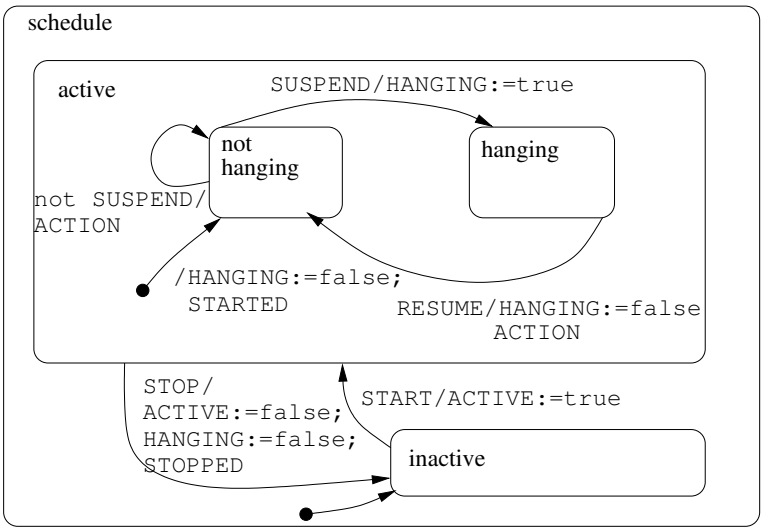


Figure 5.1: Activation States of Controlled Activity

Listing 5.1 depicts algorithmically, how the local contribution of controlled activity TIMER[4] to the distributed protocol for scheduling control is realized according to the compositional semantics. The listing describes the reaction of the model to scheduling primitives for a single step. In order to distinguish between the actual value of a variable and its future value, we assume the variable to exist in a primed and in an unprimed variant. The value of a variable at the beginning of the step is referred to by the unprimed variant, whereas the new value of the variable is referred to by the primed variant.

Notice that scheduling communication between control-activity and controlled activities is the same for asynchronous and synchronous execution. Since the control of controlled activities is always performed by an internal activity w.r.t. the entire system[5], the distributed scheduling protocol of the compositional semantics is always based on fast communication.

---

do step
begin
set primed version of local and output events to false

[4] We have chosen TIMER for this example, because the contribution of a controlled activity always consists of all primitives. Since, for example TRAIN is not controlled by any control activity, starting and stopping of TRAIN is not (and needs not to be) modeled in the compositional representation of TRAIN.

[5] In Statemate, implementation of controlled activities depends on the implementation of a controller. Hence, a top-level activity can never be a controlled activity because the controlling activity has to be part of the same design.

```
    if (ACTIVE) then
       if (STOP) then
          ACTIVE':=false
          HANGING':=false
          STOPPED':=true
          inactivate active states
       else
          if (not HANGING) then
             if (SUSPEND) then
                HANGING':=true
             else
                ACTION':=true
             end
          else
             if (RESUME) then
                HANGING':=false
                ACTION':=true
             end
          end
       end
    else
       if (START) then
          STARTED':=true
          HANGING':=false
          ACTIVE':=true
          -- for asynchronous models:
              LOCALLY_INSTABLE':=true
          activate initial states
       end
    end
    if (ACTION') then
       execute one step of model
       -- for asynchronous models:
             for any assignment or state change : LOCALLY_INSTABLE':=true
    end
       -- for asynchronous models:
             Asynchronous synchronization block (cf. listing 5.2 on page 66)
end
```

Listing 5.1: Explicit Scheduling Protocol Program

Stabilization in the Asynchronous Execution Semantics

Like scheduling of activities, stabilization has to be modeled using a distributed protocol to which all activity-models contribute.

In order to establish stability as a distributed property of all components, each component generates a local event `LOCALLY_INSTABLE` for any assignment or state change in order to indicate that no stable status has been achieved within the current step.

Based upon the distributed determination of being locally instable or being able to stabilize, asynchronous stabilization is modeled with a distributed protocol.

Listing 5.2 shows the contribution of one component to this distributed protocol. Events involved in this protocol are :

`STABLE_ENV`	input	event
`STABLE, SUPER_SYNC`	output	event
`PRESTABLE, LOCALLY_INSTABLE`	local	event

if ((not(ACTIVE) and START) or
("any state change or assignment has been performed in the current step"))
then LOCALLY_INSTABLE':=true *end*

```
if (LOCALLY_INSTABLE') then
   PRESTABLE':=false
else
  if (not PRESTABLE) then
     PRESTABLE':=true
   else
     STABLE':=true
     PRESTABLE':=false
     if (STABLE_ENV) then
        SUPER_SYNC':=true
        update input-buffers with primary inputs
     end
   end
end
```

Listing 5.2: Asynchronous Synchronization Block

Whenever an assignment or state change has been performed in the actual step, `LOCALLY_INSTABLE` is generated. Only if `LOCALLY_INSTABLE` has not been generated during the actual step, the component model becomes able to synchronize with other components in order to complete the actual super-step. This pre-stability is indicated by the local event `PRESTABLE`. If the model does not become locally instable again in the next step, a `STABLE` event is generated to indicate the will to synchronize with the other activities. External to the activity-models[6], the `STABLE` events of all components are collected by a dedicated *monitor*. If all components emit `STABLE` events at the same step, the monitor emits an event `STABLE_ENV` which is delivered to all components. Thus, `STABLE_ENV` indicates the will of all other components to synchronize. Depending on the stability of their respective environment, each activity-model generates its `SUPER_SYNC` event in synchrony to the others activities. This distributed protocol triggers each activity-model to update its input buffers. Hence, new inputs are accepted only in stable system states.

5.2 Compositional Synchronous Transition Systems

In [DJHP97] the semantics of Statemate models is defined in terms of *compositional synchronous transition systems* (CSTS). CSTS are basically STS (cf. definition 4.2), enhanced with the definition of an interface:

[6] This collection and logical combination of events is performed by so-called monitors, which are discussed in detail in [DJHP97].

Definition 5.2 (Compositional Synchronous Transition System)
Given a STS S describing the behavior of activity A. By marking a subset E of the variables V as externally observable, S results in a compositional synchronous transition system CSTS $\mathcal{C} = (V, \Theta, \rho, E)$, where V, Θ and ρ are defined the same way as for STS and

- E is a subset of V , the set of externally observable variables.

E itself consists of the disjoint sets:

- $E_{in} \subseteq V$ of externally visible variables which are only read by A , and
- $E_{out} \subseteq V$ of externally visible variables which can be modified by A.

ρ does not constrain E_{in}:

$$\forall \sigma_1', \sigma_2' \in \Sigma(E_{in}) \exists \sigma_1, \sigma_2 \in \Sigma(V) : ((\sigma_1' = \sigma_1 \downarrow_{E_{in}}) \wedge (\sigma_2' = \sigma_2 \downarrow_{E_{in}}) \wedge (\sigma_1, \sigma_2) \in \rho)$$

■

With respect to this definition, we now can define observation sequences as the restriction of runs of synchronous transition systems to their externally observable interfaces. In contrast to the definition of runs of a system, which does not take observability into account (cf. definition 4.3), an observation sequence hides away the internals of a model.

Definition 5.3 (Observation Sequence)
Given a CSTS $\mathcal{C} = (V, \theta, \rho, E)$.
Let $\pi{:=}\sigma_0\sigma_1\sigma_2...$ be a computation of $\mathcal{C}$, with valuations σ_i of V, $i \in \mathbb{N}_0$.
Let $\pi_{obs}{:=}\pi \downarrow_E$ be the restriction of π to E, such that π_{obs} is a sequence of valuations of the externally observable variables E.
We call the restriction π_{obs} to E an *observation sequence.*

■

Definition 5.4 (Parallel Composition of CSTS)
Given two CSTS $\mathcal{C}_1 = (V_1, \Theta_1, \rho_1, E_1)$ and $\mathcal{C}_2 = (V_2, \Theta_2, \rho_2, E_2)$ with $(V_1 \backslash E_1) \cap V_2 = \emptyset$ and $(V_2 \backslash E_2) \cap V_1 = \emptyset$. The parallel composition of $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined by $\mathcal{C} = (V, \Theta, \rho, E)$, where

- $V{:=}V_1 \cup V_2$
- $E \subseteq E_1 \cup E_2$:
  $E{:=}\quad E_{1_{in}} \backslash (E_{1_{in}} \cap E_{2_{out}})$
  $\cup\quad E_{1_{out}} \backslash (E_{1_{out}} \cap E_{2_{in}})$
  $\cup\quad E_{2_{in}} \backslash (E_{1_{out}} \cap E_{2_{in}})$
  $\cup\quad E_{2_{out}} \backslash (E_{1_{in}} \cap E_{2_{out}})$
- $\Theta{:=}\Theta_1 \wedge \Theta_2$. Θ thus characterizes the set of initial valuations $\{\sigma \in \Sigma(V) \mid \sigma \downarrow_{V_1} \in \Theta_1$ and $\sigma \downarrow_{V_2} \in \Theta_2\}$
- $\rho \subseteq \Sigma(V) \times \Sigma(V)$ is given by
  $(\sigma, \sigma') \in \rho$ iff $(\sigma \downarrow_{V_1}, \sigma' \downarrow_{V_1}) \in \rho_1$ and $(\sigma \downarrow_{V_2}, \sigma' \downarrow_{V_2}) \in \rho_2$

Let the parallel composition of $\mathcal{C}_1$ and $\mathcal{C}_2$ be denoted by $\mathcal{C}_1||\mathcal{C}_2$

■

If $(V_1 \backslash E_1) \cap V_2$ or $(V_2 \backslash E_2) \cap V_1$ are not empty, the variables in $(V_1 \backslash E_1)$ or $(V_2 \backslash E_2)$ need to be renamed before applying the composition.

This composition does in general not preserve viability and consistency. It may be the case that both, $\mathcal{C}_1$ and $\mathcal{C}_2$ are viable (consistent) but the composition is not. Observe that e.g. Θ may characterize an empty set even if both $\Theta_1$and Θ_2 are satisfiable. Or, one transition system may require that a valuation with $v = 1$ has to be followed by a valuation with $v = 2$ and the other system may demand that a valuation with $v = 1$ is followed by a valuation with $v = 3$. Hence a system state with $v = 1$ reached in the composed system will have no successor.

It is an important fact for this entire work that the representation of STATEMATE models according to the compositional semantics guarantees viability and consistency of composed STATEMATE designs:

"In the modeling of statecharts we will not have such contradictory requirements in components. Our semantics will not introduce deadlocks in a composed system when there is no corresponding deadlock in one component. To achieve this, the semantics of one component will contain all observable behavior of its environment. Semantical models satisfying this property are called compositional" [DJHP97].

Fact 5.1 (Parallel Composition of CSTS for Statemate)

Given two CSTS $\mathcal{C}_1 = (V_1, \Theta_1, \rho_1, E_1)$ *and* $\mathcal{C}_2 = (V_2, \Theta_2, \rho_2, E_2)$ *, s.t.* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *represent two sibling activities* A_1 *and* A_2 *within the same parent-activity* A *(where w.l.o.g. let* $\mathcal{C}_1$ *be the representation of* A_1 *and* $\mathcal{C}_2$ *be the representation of* A_2*) .*

Then $\mathcal{C}:=\mathcal{C}_1||\mathcal{C}_2$ *is built from* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *such that :*

- $\mathcal{C}$ *is consistent*
- $\mathcal{C}$ *is viable*

In particular, the representation guarantees that $E_{1_{out}} \cap E_{2_{out}} = \emptyset$ *by introducing* monitors *for shared variables. A monitor observes all actions of the components which write on* v*, collects the values and broadcasts a (nondeterministically) selected value to the environment. Furthermore it reads the given value from the environment and transports appropriate values to the components. It does not only manage the value of* v *but also the related read, written, and changed events.*

Moreover, $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are* input enabled *, i.e.:*

$$\forall \sigma_1', \sigma_2' \in \Sigma(E_{1_{in}}) \exists \sigma_1, \sigma_2 \in \Sigma(V_1) : ((\sigma_1' = \sigma_1 \downarrow_{E_{1_{in}}}) \wedge (\sigma_2' = \sigma_2 \downarrow_{E_{1_{in}}}) \wedge (\sigma_1, \sigma_2) \in \rho_1)$$

and likewise for $\mathcal{C}_2$*.*

■

This fact can be seen as an axiom, because we can not provide a formal proof without citing the entire work of Brockmeyer [Bro99].

We simply extend the definition 4.6 (Kripke structure of an STS) to CSTS:

Definition 5.5 (Kripke Structure of CSTS)

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$. Let $\mathcal{K}(\mathcal{C})$ denote the Kripke structure according to definition 4.6, which is constructed from the STS $S = (V, \Theta, \rho)$ (by ignoring E).

■

5.3 Real-Time Aspects for the Verification of Statemate Models

Open embedded system continuously interact with their environments. A system reads inputs from its environment, performs internal computations depending on these inputs, and finally delivers results of its computations back to the environment. Often, these reactions have to take place within particular time limits. Hence, not only functional correctness but also adherence to these time limits is an important issue in the development of such a system. A reference model in a model based development process should incorporate essential timing requirements at least in an abstract manner.

In order to take reaction deadlines into consideration, the modeling formalism needs to be capable of specifying not only qualitative temporal but also *real-time* aspects of the behavior of a system. Although STATEMATE permits explicit references to time, time itself is not part of the execution semantics. "The execution of a step takes zero time. The interval between the execution of two consecutive steps is not part of the step semantics. Rather, it depends on the execution environment and the time model, over which users have significant degree of control" [HN96].

In contrast to simulation, automatic verification clearly cannot allow interactions by user-commands to prescribe how time is advanced[7].

Leaving the treatment of time entirely to the simulator is a flexible concept and may be satisfactory for simulation purposes. Regarding real-time properties, this concept may lead to interpretation ambiguities of the model. When considering real-time properties, simulator settings and user interaction - such as progress of time, the size of the chosen time units, and the chosen execution semantics become essential for the behavior of a model. For reasoning about real-time aspects, timing information must be associated with the runs of the model. This means either to augment a model with explicit time information or at least an interpretation of its observable behavior w.r.t. an implicit model of time.

In general, there are two options to model real-time: either time is modeled with a *discrete* time domain, like natural numbers, or time is modeled with a *dense* time domain, like real or rational numbers. In order to choose the appropriate time domain for verification it is important to distinguish between physical and conceptual requirements. A dense time domain might be unavoidable when considering implementations or real hardware w.r.t. physical time. In the model based development process discussed in this work, STATEMATE is employed to build conceptual rather than physical models. At the level of conceptual modeling, time should be treated in a more abstract way. The chosen representation of time should permit quantitative temporal specifications but abstract from physical implementation details. Hence, a discrete time domain is appropriate, since the systems considered here are modeled using statecharts which perform their computations step by step. Observations can change only at a granularity of steps. Thus, an adequate perception of time could be an association of one step with one time unit. For the synchronous execution of STATEMATE models this interpretation matches perfectly.

Regarding asynchronous execution of models, the interpretation of time is more sophisticated.

[7] Also in STATEMATE the interpretation of time varies among the different tools of the tool set:
"While the basic algorithm for a step ... is implemented in STATEMATE's simulation and dynamic test tools, and in its various code-generators, each of these executes a step under somewhat different circumstances, and the way the two models of time are reflected in the execution differs slightly among them" [HN96].
"The software code generators generate one style of code; however, two different schedulers are provided that support different time models. One of them uses the CPU time. That has the effect that steps, and therefore super steps also, take more than zero time" [HN96].

On the one hand, the virtual simulation time elapses only between stable states. Thus, an appropriate perception of time associates a super-step with one unit of time. On the other hand, the asynchronous semantics is defined upon sequences of steps. A global stabilization of the model takes place if in the last step of a sequence of steps no action can be performed without reading new inputs or increasing the virtual simulation time. Although steps are assumed to take zero time, they are separated by δ-delays.

All activities of the model are executed in parallel; asynchronous stabilization is derived from the *synchronous* behavior of the parallel parts of the model. Unfortunately, there is no fixed relation between a super-step and the sequence of steps required to reach stabilization. In general, the compositional model representations of internal activity have mixed interfaces, containing both fast and slow inputs. Also scheduling of controlled internal activities by control-activities is represented by fast communication. A control-activity can start, stop, suspend and resume its controlled activities. The effect of this scheduling takes place immediately - regardless of the chosen simulation semantics. According to the compositional semantics, super-step synchronization is represented by a distributed explicit protocol. Only if all activities of a system are willing to become stable, the system reaches a stable status. Thus, the activities have to communicate their local status to each other in every step, in order to conjointly become stable. Becoming stable is then distributedly computed by each activity based upon local stability and the status of the other activities.

It is an important issue of the asynchronous semantics that models need not always necessarily become stable. Super-step stabilization (divergence-freedom) is itself a property that must be verified for a system. The appropriate measure for verification of stabilization is an upper bound for the length of all super-steps. Such a check - verifying if a user-guessed upper bound holds for all super-steps - is offered by the STVE.

The important contribution of fast communication to the asynchronous interpretation requires the ability of capturing also timing w.r.t. steps in specifications for verification of internal activities. In order to specify synchronization and scheduling properties, the dedicated primitives of the model representation must be observable for requirement specifications. Considering super-steps without regarding internal step communication may be appropriate only for a black-box view of the *entire system*. When specifying requirements for internal activities of a system, capturing fast communication is unavoidable. Thus, in requirement specifications for verification it is often necessary to *quantitatively refer to time in terms of super-steps as well as in terms of δ-delays.*

In order to refer to observations with respect to time, the runs of a system are associated with timing information.

STATEMATE's asynchronous simulation semantics assumes time only to advance between consecutive super-steps, while intermediate steps are executed sequentially without consuming time (except for δ-delays). This concept of time is captured by the following definition:

Definition 5.6 (Timed Observation Sequence)
For a given CSTS $\mathcal{C} = (V, \theta, \rho, E)$, we define:

- Let Σ_E be the set of possible valuations of the observable variables E.
- Let $\pi = \sigma_0\sigma_1\sigma_2...$ be an observation sequence for E, with valuations $\sigma_i \in \Sigma_E$, $i \in \mathbb{N}_0$.
- Let $\tau = \tau_0\tau_1\tau_2...$ be an infinite sequence of times $\tau_i \in \mathbb{N}_0$, $i \geq 0$ such that τ is monotonic increasing: $\forall i \in \mathbb{N}_0 : \tau_i \leq \tau_{i+1}$. We call τ a *time sequence.*

A *timed observation sequence* $ts = (\pi, \tau)$ is a pair consisting of an observation sequence π (cf. definition 5.3) and a time sequence τ such that σ_i is the valuation of E at time τ_i and in step i. Let further $TComps(\mathcal{C})$ denote the set of all timed observation sequences of $\mathcal{C}$.

■

The chosen time model for verification of asynchronous models associates a super-step of the model with an instant of time. Time is mapped to natural numbers and elapses in commensurate time units. Throughout this work, for simplicity the value of a time unit is chosen to be 1. According to this time model, a timed observation sequence can be obtained from a run of the model by assuming a global clock which is increased with every super-step. For the asynchronous semantics, observations at the granularity of single steps are associated with δ-delays, because the synchronous communication underlying the asynchronous semantics is based on sequentiality of chain reactions to internal and external changes sensed within a super-step. A quantitative treatment of δ-delays and steps, respectively, can be achieved by interpreting the distances between positions in the observation sequence. In contrast, a quantitative treatment of super-steps has to take dynamic stabilization into account.

Definition 5.7 (Timed Observation Sequences and Stabilization)
Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$ describing the behavior of activity A for the asynchronous execution semantics, where E is the set of externally observable model variables of CSTS $\mathcal{C}$ (according to the compositional semantics). In particular, part of the interface according to the definition of $intf(A)$ is the dedicated output `SUPER_SYNC`, which indicates super-step stabilization and hence increase of time. The time sequence portion τ of a timed observation sequence can be obtained from the sequence $\pi{:=}\sigma_0\sigma_1...$ of valuations $\sigma_i \in \Sigma_E$,$i \geq 0$ by counting truth valuations of `SUPER_SYNC`:

$$\tau_i{:=}\begin{cases} 0 & i = 0 \\ \tau_{i-1} + 1 & [\![\texttt{SUPER\_SYNC}]\!](\sigma_i) = true \\ \tau_{i-1} & \text{otherwise} \end{cases} \tag{5.1}$$

Consequently, π determines the timed observation sequence $ts = (\pi, \tau)$, where τ is derived from π according to the above equation (5.1).

■

For the synchronous semantics, we disallow multiple steps in the same instant of time. Time is not only monotonic, but strictly monotonic increasing. The following definition captures the notion of time for the synchronous semantics.

Definition 5.8 (Simplified Timed Observation Sequence)

- Let $ts = (\pi, \tau)$ be a timed observation sequence, where τ be a monotonically increasing time sequence.
  In contrast to definition 5.6, we require τ to be *strictly* monotonic increasing, i.e.: $\forall i \in \mathbb{N}_0 : \tau_i < \tau_{i+1}$.
  Moreover, we require $\forall i \geq 0 : \tau_{i+1} - \tau_i = 1$.

We then call ts a *simplified timed observation sequence*.

■

Simplified timed observation sequences are the perfect perceptions of systems executing according to the synchronous semantics. Since time is strictly monotonic increasing, "next step" means the same as "next time". A simplified timed observation sequence can hence be obtained from an observation sequence by simply referring to the positions in the sequence.

5.4 System Representation for Formal Verification

A high-level design tool such as STATEMATE provides a broad range of modeling concepts and offers the developer an intuitive, application oriented way of specifying a systems functional structure and behavior.

Syntactical diversity aggravates implementation of efficient algorithms for analysis and verification. Verification means performing computations on a semantical representation of a system. The simpler this representation can be kept, the simpler and more efficient the verification tools can be designed. Moreover, tailoring algorithms to a particular modeling tool - such as Statemate - disables the reuse of the verification tools in another verification context. Thus, an intermediate representation of STATEMATE system designs is used on the path down to the verification tool set. The translation into the intermediate format maps high-level constructs of the modeling languages to more basic - but semantical equivalent - constructs. In addition, the translation introduces explicitly all constructs into the model representation, which are required according to the compositional semantics - such as e.g. the distributed stabilization protocol for the asynchronous semantics, the interface declarations, etc.

In [Bro99], Brockmeyer presented an automatic and efficient translation of STATEMATE models into an intermediate representation, putting the compositional semantics faithfully into practice. Serving as basis for the verification tool set, the languages SMI (System Modeling Interface) and SSL (System Structuring Language) have been developed in the context of [Bro99] and [Wit99]. SMI is particularly suitable for representing model behavior, whereas SSL has been designed for representing the structure of a system and the communication flow among its activities.

SSL provides information about the structureof a system in a more coarse grained and in a fine grained way, which will be explained later in this section. We will refer to the structure representation and in particular to the mappings of the fine grained view in chapter 7, where compositional verification and proof-management will be presented.

System Modeling Interface (SMI)

SMI has been designed to describe the behavior - not the structure - of embedded systems in an imperative programming language style. SMI programs describe synchronous transition systems in a step-by-step manner. Each run through the code corresponds to one step of the transition system. The environment provides the program with new inputs, the program computes new values for its variables and then waits for new inputs again. Depending on its actual status, the program controls which variables are assigned with which new values and hence determines the next status of the system that will become visible to the environment at the end of the step. This way, a SMI program describes all possible steps which the transition system can perform.

Program statements and control structures of SMI are:

- Sequential composition - in general, all program statements are composed sequentially according to the order of their appearance,

- Parallel composition : `PAR` S_1 `||` ... `||` S_n `RAP` - the parallelly composed code blocks $S_1, \ldots, S_n$ are executed independently and without any defined order.

- If-then-else : `if b then` S_1 `else` S_2 `end` - if boolean expression `b` evaluates to true, then code block S_1 is executed, otherwise code block S_2 is executed.

- Deterministic choices : `DCASE []` b_1`:`S ... `[]` b_n : S_n `DESAC` - each of the choices is guarded by a boolean condition. Since, at most one of the guards is expected to be true at the same moment, the first choice whose guard evaluates to true is chosen.

- Non-deterministic choices : `NCASE []` b_1`:` S_1 ... `[]` b_n`:` S_n `NDESAC` - each of the choices is guarded by a boolean condition, of which more than one can be true at the same moment. In this case one of the choices for which the guard evaluates to true is nondeterministically chosen.

- While loops: `WHILE b DO S OD` - if the boolean condition `b` evaluates to true, code block `S` is iteratively executed unless b evaluates to false.

- assignments to variables - only local and output variables can be assigned with values. Assignments to inputs and to constants are not permitted.

- `SKIP` - does nothing and leaves the state-space of the program unchanged.

In order to distinguish between the value of variables at the beginning of one step and the newly assigned values, variables exist in a primed and an unprimed version. Primed variables denote the actual values, while unprimed variables denote the "old values", i.e. the value at the beginning of the step. Unprimed variables keep their value until the end of the actual step. The variables are updated before performing the next step, i.e. the values of the primed variables are copied to the unprimed variables. Inputs only exist in an unprimed version, since they are controlled by the environment and do not change their value during the computation of a step. Assignments of values are only permitted to primed variables.

Regular variables are externally observable only in their unprimed version. Hence, in general, reactions to inputs can only be observed in the next step; but sometimes it is necessary to model instantaneous reactions to inputs, i.e. within the same step. Therefore, SMI offers also a concept of instantaneous variables. Instantaneous variables are *observable* only in their primed version. However, also instantaneous variables keep their value for the next step.

Sometimes, it is not necessary to store temporary results of computations for the next step - although shared among different assignments, intermediate values are sometimes of interest only in the actual step. Therefore, SMI supports so-called auxiliary variables, which exist only in a primed version, i.e. do not memorize their value for the next step. Since auxiliary variables are used only for intermediate results, they are not observable from outside the program.

For the observation of particular internal values of a program, SMI supports a concept of memoryless observers, which are outputs of the program making observable internal values of interest. In contrast to auxiliary variables, these outputs are observable from outside the program. Like auxiliary variables they also do not store their value for the next step.

The variables of a SMI program are collected and defined in a symbol table associated with the program. Each variable is defined in terms of the following attributes:

- Name - the name of the data-object. All names are unique in a SMI program.
- Type - SMI supports a rich set of of types: *integer, bit*, *enumeration*, *condition*, *string*, *bit-array* and *real* are basic types. Based upon these basic types *arrays*, *unions,* and *records* can be recursively constructed. Integer, reals, bit-arrays and strings are in general unbounded types. In order to keep the state space of the program finite, variables can be restricted to subranges of these types.
  Arrays, unions, records and subranges are defined using named type-declarations in the symbol-table.
- Mode - SMI distinguishes a variety of modes as well as some combinations of these modes: *input*, *output*, *instantaneous_ output, local*, *auxiliary*, *observer* and *constant*.
  - Inputs are used read only by the SMI program, they are controlled by the environment. Note that inputs can always change at every step. In order to represent *slow* inputs according to the compositional semantics of asynchronous models, buffers associated with these inputs must be provided by an SMI program representing a super-step model. These buffers are assigned with input-values only in stable states. Instead of referring to a real input all accesses to a slow input are replaced by accesses to the associated buffer. Thus, a slow input is modeled using the basic constructs of an input and a guarded assignment, where the guard is the indication of a stable status.
  - *Local* or *output* variables are controlled by the SMI program. The distinction into *local* and *output* is used to denote the intended usage of the data-object. A *local* data-object is read and written by the code and not expected to be read outside the code, while output data-objects are intended to be read by the environment. In contrast to regular *output* variables, mode *instantaneous_ output* denotes output variables, for which the primed version is observable from outside the program instead of the unprimed version.
  - Modes *auxiliary* and *observer* are useful modes for variables which do not need to keep their values for the next step. Since variables with mode *auxiliary* or *observer* variables have a defined value only during a step, they do not enlarge the state space of the SMI program.
  - *Constants* can never change their value - in particular no primed version is required for their representation.
- Initial value - For constants the initial value defines the value of the constant, otherwise - for local and output variables the initial value defines the value at initialization. If no initial value is specified, the variable will be initialized to a data-type specific default. For local and outputs variables also initialization to a non-deterministically chosen value of their domain can be specified.
  Since inputs are controlled by the environment, they can not be initialized with a particular value.

The semantics of SMI has been formally defined using Synchronous Transition Systems (STS, cf. definition 4.2 and [PS97]) . By taking formally defined model interfaces into account, the semantical definition has been lifted to Compositional Synchronous Transition Systems (CSTS, cf. definition 5.2 and [DJHP97]).

A detailed definition of SMI can be found and its semantics can be found in [BBEH99, BBEW98].

System Structuring Language (SSL)

SMI has been designed to describe the behavior of models. The symbol table associated with a SMI program defines a set of variables and their directions according to their usage in the SMI program. This way, all necessary information is provided for technical purposes. Variables added to the SMI program for only technical reasons, as for example resolution of non-determninism, are not distinguished from user defined variables in the symbol table. This '*technical interface*', as provided by the symbol table, represents a *glass-box* view to a model, while the '*logical interface*' for requirement specifications has to provide a *black-box view* to the model, hiding away all internals of the model, such as local variables and auxiliary inputs or observers:

- The translation from Statemate to SMI annotates the SMI program with additional observers. For example, for each graphical transition an observer is added indicating the transition being taken. Aimed at supporting specific analysis techniques - such as a drive to transition analysis -, these observers do not belong to the '*logical interface*' of the represented activity.
- Auxiliary inputs are used for the resolution of non-determinism: SMI supports non-deterministic choices of code blocks, for which the guards need not be exclusive. By introducing sufficiently many fresh inputs, the choice of which code block will be executed can be left to the environment, thus making the non-deterministic choice internally deterministic.
- SMI programs representing activities according to the asynchronous semantics model slow inputs using primary inputs and buffer-variables as described above. These buffers are local variables from a technical point of view, which is correctly represented in the symbol table. From the '*logical interface*' point of view the driving primary inputs are not observable, whereas the buffers have to be considered as inputs.

For the definition of logical interfaces and their inter-connections w.r.t. information flow in the represented system, SMI is embedded in SSL [SAC97], which borrows its structuring operators and type concept from the hardware description language VHDL. The supported set of basic types and declaration rules for composed types is rich enough to entirely comprise the type concept of SMI.

All type, sub-type and constant declarations which are required to define typed interfaces and communication structure of a model are collected in so-called packages. The description of a system's structure refers to these packages containing the required type declarations in order to define interfaces and the communication structure of the model. We will not consider packages im more detail here, but refer to [SAC97] instead.

Besides packages, the units of a SSL structure descriptions are *entities*, *architectures* and *configurations*.

Definition 5.9 (SSL design description)
Formally a SSL design description is a quadruple

$$Des = (packages, Entities, Architectures, Configurations), \text{ where}$$

- *packages* is a set of packages containing type, sub-type and constant declarations,
- *Entities* is a set of entity declarations, each providing a typed communication interface for a portion of the design,

- *Architectures* is a set of architecture bodies, each of them referring to an entity of *Entities*,
- $Configurations$ is a set of configurations, each of them referring to an entity and an architecture.

■

In an *entity,* the observables of components are listed together with their data-flow directions 'in' or 'out' and their data-types. This way, an entity provides a unique declaration of a component's interface. Each component (activity) of a system is associated with exactly one entity.

Definition 5.10 (Interface)
A SSL-interface $IDecl$ is a set of communication signals, which are also called *ports.* For each of the ports $p \in IDecl$, the following attributes are defined:

- $name(p)$ is a unique identifier with respect to $IDecl$, i.e. all ports in an interface have different names.
- $mode(p) \in \{in, out\}$ defines the direction of p, and
- $type(p)$ defines the data domain of p by referring to a type declaration in the scope of $IDecl$.

For an entity E, we will write $Intf(E)$ to denote the interface of entity E.

■

Architectures are used to bind views to entities. They either provide structural descriptions or serve as containers which incorporate views in view specific representations.

- A structural architecture is a description of a design decomposition level. Structural descriptions are constructed via the declaration of components, their instantiation and definition of their inter-connections. Component declarations are kind of *local entity declarations.* In fact, they are very similar to entity declarations but in contrast to entities, they are no independent units of the structure. Components are defined only in the context of the surrounding architecture. From its component declarations, a structural architecture defines component instances, to which entities with compatible interfaces can be bound.
  Communication between components is represented by connecting their interfaces via signals, i.e local communication channels. Signals are declared as typed objects locally to the structural architecture. In order to avoid multiple writer conflicts, each signal can be connected to at most one output of a component, but to arbitrary many inputs of other components. Hence, the direction of a signal is determined by the mapping to component interfaces.
- Simple architectures are used as containers for different views to be associated with an entity. Simple architectures are placeholders; *variables* which can contain any kind of interface related information. In the context of this work, they are used to associate entities with behavior descriptions represented by SMI code, or with a set of STDx requirement specifications. By integrating SMI code or STDx requirement specifications into the structure description of a model, architectures play a central role in the management of the decomposed model representation for verification. Architectures enable the STVE to navigate a structural design description and to access the different views to the sub-components within an uniform data-base format.

Definition 5.11 (Classification of Architectures)
Let *Architectures* be the set of architectures of a SSL design description. Let *Akind* be a function indicating the kind of an architecture:

$$Akind : Architectures \rightarrow \{behaviour, specification, structural\}$$

Then, for $A \in Architectures$:

- $Akind(A)$:=*behavior*, if A is a simple architecture containing a SMI program and a symbol table
- $Akind(A)$:=*specification*, if A is a simple architecture containing a set of STDx requirement specifications
- $Akind(A)$:=*structural*, if A instantiates at least one component instance.

Furthermore, let $Contents(A)$ obtain a set of the contents of A, s.t.

$$Contents(A) := \begin{cases} \emptyset & \text{if } Akind(A) = structural \\ \{spec | spec \text{ is a set of STDx specifications } \} & \text{if } Akind(A) = specification \\ \{SMI - program, symbol - table\} & \text{if } Akind(A) = behaviour \end{cases}$$

■

The semantics of $Contents(A)$ for $Akind(A) = specification$ is discussed in section 6.5 and in section 7.3, respectively. According to definition 5.2 and [DJHP97, Bro99], $[\![Contents(A)]\!]$ is given by a CSTS, if $Akind(A) = behavior$. Of course, the definition of $Contents(A)$ is a bit informal, but sufficient for the intention of this work, because we only need a notion of an access-function to the contents of architecture A and their meanings.

The semantics of structural architectures is considered in the remainder of this section. In practice, the definition of kind and contents of architectures can easily be extended also to, for example, Live Sequence Charts [Klo03], as it has been done for the cited work.

Each entity can be associated with different architectures: for example, a structural description, another architecture containing a SMI program and a specification architecture containing a set of STDx requirement specifications may refer to an entity. Configurations uniquely denote a pair of one entity and one of its architectures.

Neither architectures nor configurations are independent units of a design structure, but always refer to an entity.

Definition 5.12 (Relations among Entities, Architectures and Configurations)
Given a SSL description of a design, let

$$ent : Architectures \cup Configurations \rightarrow Entities$$

denote the referenced entity for each architecture or configuration.
Furthermore, for each configuration let

$$arch : Configurations \rightarrow Architectures$$

denote the configured architecture.
Then, for a configuration and a configured architecture it must hold:

$$ent(arch(C)) = ent(C).$$

While the interface of each component of a system is represented by exactly one entity, the different views to a component can be represented by arbitrary many architectures. Let $Archs : Entities \rightarrow Architectures$ and $Configs : Entities \rightarrow Configurations$ denote the sets of architectures and configurations respectively, which refer to a particular entity, s.t.:

- $Archs(E)$ denotes the set $\{A \in Architectures | ent(A) = E\}$, and
- $Configs(E)$ denotes the set $\{C \in Configurations | ent(C) = E\}$.

■

Structural architectures instantiate components of the declarations and assign names to these instances. Like a *local variable*, a component instance is a typed placeholder for the instantiation of an appropriate sub-design within a structural architecture. Although component instances have a meaning only w.r.t. the instantiating structure, they rigorously define the interface to a sub-design which can be instantiated within the structure. Later on, configurations bind concrete entities and architectures to these local interfaces.

A structural architecture can best be compared to a hardware board. The board provides sockets to plug in circuits with an appropriate pin-out. The layout of the board wires the pins of the sockets, without implementing further behavior. Then, plugging the right circuits into the sockets of the board is a matter of configuration. While the interface of the board does not change, the configuration selects appropriate circuits to be plugged into the sockets.

In order to keep the formal treatment of component instances as simple as possible, we assume that all component instances are assigned unique names[8]

Let $CompInsts$ be the set of component instances of all structural architectures in a design. Then, we write $comps(A)$ to denote the set of component instances of a particular structural architecture A, where $comps$ is given by

$$comps : Architectures \rightarrow 2^{CompInsts}.$$

The mapping of inter-connections between component declarations and the mapping of component instances to entities consists of two parts. The structural architecture defines local component interfaces and local inter-connection channels. The component interface objects are mapped to either the local inter-connection channels or to interface objects of the entity to which the architecture is bound by a configuration. The configuration of a structural architecture not only binds configurations - each identifying an entity-architecture pair - to the component declarations, but also maps the component interface objects to interface objects of the instantiated entities.

By recursively associating configurations to components of a structural architecture, pairs of entities with associated architectures are associated with the components. This way, entire design hierarchies can be identified with their respective top-level configuration.

[8] In the technical realization, component instances are uniquely accessible using the information about instantiating architecture and the entity referred to by the activity.

Definition 5.13 (Recursive Configurations)
A configuration C of a structural architecture A associates the component instances of A with configurations. Let

$$Assocs : Configurations \rightarrow 2^{CompInsts \times Configurations}, \text{ s.t.}$$

$Assocs(C)$ denotes the set of associations for component instances of a structural architecture $arch(C)$ with configurations defined by configuration $C \in Configurations$:

$$\{(comp\_inst_i, C_i) | comp\_inst_i \in comps(arch(C)) \wedge C_i \in Configurations\}$$

- For $ca \in Assocs(C)$
  - let $Inst(ca)$ denote the component instance of association ca, and
  - let $Conf(ca)$ denote the configuration of association ca.

■

Using definitions 5.11 and 5.13, the configurations of a design description can be classified regarding the contents of the configured architectures.

Definition 5.14 (Classification of Configurations)
The classification of configurations w.r.t. the contents of the configured architectures is given by:

$$Ckind : Configurations \rightarrow \{behavior, specification, mixed\}, \text{ s.t.}$$

.
for a given configuration C:

- $Ckind(C)$:=$behavior$, if either
  1. $Akind(arch(C)) = behavior$, or
  2. $Akind(arch(C)) = structural \bigwedge \forall ca \in Assocs(C) : Ckind(Conf(ca)) = behavior$
- $Ckind(C)$:=$specification$, if either
  1. $Akind(arch(C)) = specification$, or
  2. $Akind(arch(C)) = structural \bigwedge \forall ca \in Assocs(C) : Ckind(Conf(ca)) = specification$
- $Ckind(C)$:=$mixed$, if $(\not\forall ca \in Assocs(C) : Ckind(Conf(ca)) = behavior) \wedge (\not\forall ca \in Assocs(C) : Ckind(Conf(ca)) = specification)$

We further need a measure for the depth of a configuration. Let $depth : Configurations \rightarrow \mathbb{N}$ be this measure, where

$$depth(C) := \begin{cases} 0 & \text{if } AKind(arch(C)) = specification \\ & \vee AKind(arch(C)) = behavior \\ max\{depth(Conf(ca)) + 1 | & \text{otherwise} \\ \quad \forall ca \in Assocs(C)\} & \end{cases}$$

■

Fact 5.2 (Depth of Specification Configuration)
Without further proof, we state the following fact: Integrated with the STVE is a tool that initiates the architectures and configurations forming the specification configuration hierarchy for an existing SSL design description. This tool guarantees that:

$$\forall C : (Ckind(C) = specification) \Rightarrow (depth(C) \leq 1)$$

■

Using SSL and SMI, STATEMATE models are translated into a modular intermediate representation suiting the compositional semantics. Hierarchy and communication relations of the components are described by SSL, while the behavior of the components is described by SMI code assigned to architectures of the structure description. Semantically, a SSL structure with behavioral architectures is interpreted as the synchronous parallel composition of its components.

It depends on the complexity of a considered system component, if its SMI representation is directly manageable by the verification tools or if the verification task has to be decomposed to smaller sub-components. The translation of STATEMATE models to SSL and SMI produces equivalent structural and behavioral representations for every activity, which is not a leaf of the hierarchy. Statecharts can not be further decomposed. Hence, they are always represented by leaves of the SSL hierarchy.

Fact 5.3 (Equivalent Behavioral Configuration for Structural Configurations)
The model representation using SMI and SSL according to [Bro99] *guarantees that for each structural configuration C_{struc} with $Ckind(C_{struc}) = behavior$ and $depth(C_{struc}) > 0$ there exists a configuration C_{beh} with $depth(C_{beh}) = 0$ and $Ckind(C_{beh}) = behavior$, s.t. C_{beh} is behaviorally equivalent to C_{struc}.*

More formally:

Let $\mathfrak{C}(C)$ denote CSTS $\mathcal{C}$, which represents the behavior that is associated with $ent(C)$ by C. Let $\mathfrak{C}(Conf(ci))$ be the set of CSTS $\mathcal{C}_{ci}$, which represent $ci \in Conf(ci)$ according to $ent(Conf(ci))$ for all $ci \in Assocs(C_{struc})$. Then:

$$\forall C_{struc}, depth(C_{struc}) > 0 : \exists C_{beh} : depth(C_{beh}) = 0 :$$
$$\forall ts \in TComps \left(\underset{ci \in Assocs(C_{struc})}{||} \mathfrak{C}(Conf(ci)) \right) \Leftrightarrow ts \in TComps(\mathfrak{C}(C_{beh}))$$

■

We have to take fact 5.3 as given without further proof for the context of this work. We refer the reader to [Bro99] for the foundation of this fact.

As an example we consider the top-level activity of the radio-based signaling system as shown in figure 3.2 .

Example

Activity chart `SYSTEM` communicates with its environment via sensors and actuators both represented by external activities (dashed boxes in figure 3.2). Flowlines denote which data-items are communicated from the sensors to `SYSTEM` and which data-items and events are communicated from

SYSTEM to the actuators. From the flowlines and the usage of events, conditions and data-items inside SYSTEM an entity declaration is derived which captures the communication interface of SYSTEM. In addition, all scheduling primitives required according to the compositional semantics are added to the interface. The resulting entity declaration is depicted as triangle in figure 5.2.

An architecture (represented by the filled rectangle in 5.2) is created as container for the behavior description of the entire model, i.e. SMI code and symbol table describing the behavior of activity chart SYSTEM. This behavior description instantiated by the architecture refers to the observable interface as defined by the entity. A configuration (depicted as circle) binds the architecture to the entity. Hence, configuration SYSTEM_SMI_CONFIG uniquely denotes this pair of entity SYS_ENT and architecture SYS_BB.

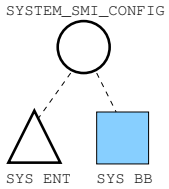


Figure 5.2: Basic SSL description

Activity chart SYSTEM consists of three sub-activities , TRAIN , CROSSING , and COMMUNICATION. Like SYSTEM, triples of entity, architecture and configuration represent interfaces and associated behavior for the sub-activities.

As discussed above, an entity can be associated with different views. A structural view of activity chart SYSTEM is represented by binding a structural architecture SYS_SB (depicted by the empty rectangle in figure 5.3) to entity SYS_ENT.

SYS_SB provides declarations of interfaces for sub-components together with mappings between these interfaces using local inter-connection channels. The sub-component interfaces fit to the entities TRAIN_ENT, CROSS_ENT and COMM_ENT - modulo renaming - of the interface objects.

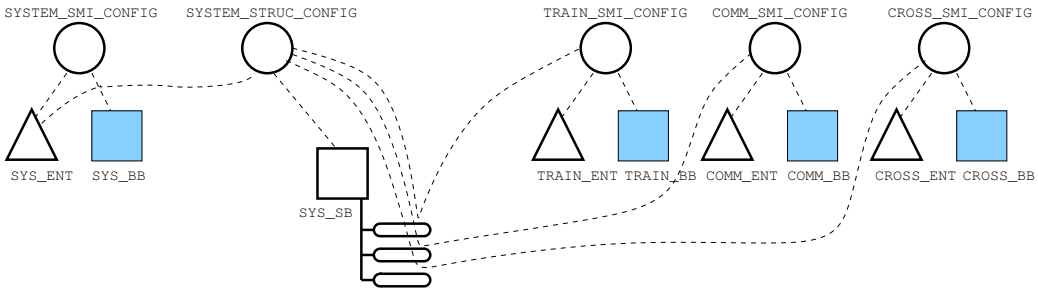


Figure 5.3: SSL Hierarchy

Figure 5.3 shows configuration SYSTEM_STRUC_CONFIG binding the structural architecture SYS_SB to entity SYS_ENT and also binding the configurations TRAIN_SMI_CONFIG as well as COMM_SMI_CONFIG and CROSS_SMI_CONFIG to component declarations of the structural architecture SYS_SB.

Fine Grained View

Besides defining coarse grained associations of which entity-activity pair is instantiated at which component instance, SSL also defines the concrete mapping of interfaces to interfaces in a fine grained manner.

Since interfaces and interface-mappings have to strictly conform to type compatibility rules, entities as well as architectures and configurations have to refer to type declarations - as they are provided in packages. Therefore, each design unit of SSL - package, entity, architecture or configuration - has its own so-called *scope*. The scope of a design unit consists of all declarations visible to the particular design unit. The contents of a package can be made visible to a design unit by a so-called *use-clause*.

Definition 5.15 (Scope)

- The scope of an *entity* contains all contents of *used* packages as well as the interface objects declared by the entity.
- The scope of an *architecture* is inherited from the entity to which the architecture refers. Within the scope of an architecture are also all additionally used packages as well as the signal declarations contained in the architecture.
- The scope of a *configuration* is inherited from the configured architecture. In case of a *structural configuration*, the scope is extended also with the scopes of the configurations associated with the component instances.

■

For a structural architecture A and one of its component instances $comp\_inst$, let $CIntf(comp\_inst, A)$ denote the interface defined by $comp\_inst$ w.r.t. A. The only difference between this definition and definition 5.10 is that an entity has its own scope, while a component instance is defined only in the scope of structural architecture A. Thus $Cintf(comp\_inst, A)$ explicitely refers to A in order to take the scope of A into account, while for $Intf(E)$ for an entity E the scope is already determined by E. Note, that although ports of different component instance interfaces may have the same name, they are uniquely identified in the scope of the instantiating structural architecture by their access path.

The interface objects of component instances are mapped to either signals declared by the structural architecture or to interface objects of the entity to which the structural architecture refers. A direct mapping of a interface objects of one component instance to interface objects of another component instance is not permitted. Thus, for all inter-connections between component instances, appropriate signal declarations have to provide communication channels to connect the interfaces of the components.

Definition 5.16 (Signals of an Architecture)

The local signals $LocSig$ declared in a structural architecture A form a set, for whose elements $sig \in LocSig$ the following attributes are defined:

- $name(sig)$ is a unique identifier with respect to $LocSig$ and $Intf(ent(A))$.
- $mode(sig) = local$ specifies that the signal is local to architecture A

- $type(sig)$ defines the data domain of the signal by referring to a type declaration in the scope of A.

The signals in the scope of A are then given by

$$Sig(A) = LocSig(A) \cup Intf(ent(A)).$$

■

SSL requires type consistency for the mappings. Only ports and signals with compatible types can be mapped to each other.

Each communication channel must have a unique driver. One output of a component can be connected to one or more inputs of other components via signals, s.t. broadcasting can be realized through this concept. In particular, it is not permitted to connect different outputs of component declarations to one input of a component declaration via a signal - since this would induce write conflicts which need resolution. Especially, there is no direct mapping of shared-variable parallelism which is quite often used in STATEMATE models. Such access resolution problems have to be provided explicitly within a SSL structure, for instance by providing appropriate additional components, resolving write/write and read/write conflicts.

The inter-connections between component instances and the connections to the interface of the actual decomposition level are described by a mapping.

Definition 5.17 (Mapping of Structural Architecture)

The mapping of component instance interfaces to the scope of a structural architecture A is given by the mapping

$$mapa: \bigcup_{comp\_inst \in comps(A)} CIntf(comp\_inst, A) \rightarrow Sig(A), \text{ s.t.}$$

- $type(mapa(i)) = type(i)$, i.e. the mapping is type conform.
- $mode(i) = out \Rightarrow mode(mapa(i)) \neq in$, i.e. outputs of a component interface can be mapped to local signals of A, or to outputs of the entity $ent(A)$, to which A refers.
- $mode(i) = in \Rightarrow mode(mapa(i)) \neq out$, i.e. inputs of a component interface can be mapped to local signals of A, or to inputs of the entity $ent(A)$, to which A refers.
- $\forall sig \in Sig(A)$:

$$\begin{cases} mode(sig) = in \wedge \nexists i \in mapa^{-1}(sig) : mode(i) = out \\ mode(sig) = out \wedge \sharp(mapa^{-1}(sig)) = 1 \wedge \exists i \in mapa^{-1}(sig) : mode(i) = out \\ mode(sig) = local \wedge \exists i \in mapa^{-1}(sig) : mode(i) = out \wedge \forall j \in mapa^{-1}(sig) : \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad j \neq i \Rightarrow mode(j) = in \end{cases}$$

If sig is an input of $ent(A)$, then arbitrary many inputs of component interfaces can be mapped to sig.

If sig is an output of $ent(A)$, then exactly one output of one component interface must be mapped to sig.

If sig is a local signal of A, then there must be exactly one output from component interface be mapped to sig, while other component interface ports mapped to sig can only be inputs.

■

Definition 5.18 (Compatible Interfaces)
Two interfaces $IDecl_1$and $IDecl_2$ are called *compatible,* if for all interface objects of $IDecl_1$ there exists a interface object in $IDecl_2$ with same *mode* and same *type*, and vice versa.

■

Definition 5.19 (Wellformedness of Configurations)
A configuration C of a design is *wellformed* if the following conditions are fulfilled:

completeness $\forall comp\_inst \in comps(arch(C)) : \exists ci \in Assocs(C)$, s.t. $Inst(ci) = comp\_inst$, i.e. all component instances are associated with a configuration.

consistency $\forall ci \in Assocs(C) : Intf(ent(Conf(ci)))$ is *compatible* with $CIntf(Inst(ci), arch(C))$, i.e. for all associations of component instances with configurations, the interface of the entity bound to the component instance is compatible with the interface of the component instance.

pureness $Ckind(C) \neq mixed$, i.e. all component instances are associated with configurations of either kind behavior of of kind specification.

■

Definition 5.20 (Scope of a Structural Configuration)
The scope of a *wellformed* configuration configuring a structural architecture is given by:

- $SigCompConf : Configurations \rightarrow 2^{Configurations \times SigNames}$, where $SigCompConf(C)$ denotes the set:

$$\bigcup_{ci \in Assocs(C)} \{(config, sig) | \, config = Conf(ci) \wedge sig \in Intf(ent(Conf(ci)))\}$$

- $SigConf(C) = SigCompConf(C) \cup \{(C, sig) | \, sig \in Sig(arch(C))\}$

■

A structural configuration maps every port of an entity associated with a component instance to a port or signal in the scope of the structural architecture.

Definition 5.21 (Mapping of a Structural Configuration)
A *wellformed* configuration C configuring a structural architecture defines a mapping

$$mapc : SigCompConf(C) \rightarrow \bigcup_{comp\_inst \in comps(arch(C))} CIntf(comp\_inst, arch(C)).$$

Hence, in combination with the mapping of the configured structural architecture (cf. definition 5.17)

$$mapa : \bigcup_{comp\_inst \in comps(arch(C))} CIntf(comp\_inst, arch(C)) \rightarrow Sig(arch(C)),$$

the mapping of interfaces - of entities bound to the component instances - to the scope of the structural architecture is defined by:

$$mapa \circ mapc : SigCompConf(C) \rightarrow Sig(arch(C))$$

■

6 Requirement Capturing for Open Embedded System

In section 2.2, the different roles in the V-Model development process have been discussed. In each phase of the development process an incremental sub-process is performed (cf. figure 2.1). Within these sub-processes the particular product is developed by iterative refinements. The iterations start with the product developed in the previous phase plus the requirements to be met in the current phase. After completion of the refinement iterations, the product is submitted to quality assessment. Then, either quality assessment accepts the product or rejects its acceptance. In the latter case the incremental sub-process is re-iterated until the product is submitted to assessment again.

In the model based development process described in this work, products of the earlier phases are conceptual STATEMATE models. Formal verification can be applied not only in the assessment of submitted models, but can also support modeling activities during the refinement iterations of the development itself. Various quality criteria can formally be established already during the iteration cycle, thus increasing not only the quality of intermediate products but also the traceability of the development process itself. Since conceptual models produced in the early phases are aimed at being reference models for later phases of the development, clearness and accuracy of these models is a major concern of model based development processes. Dedicated formal analysis techniques support the developer in detecting and avoiding ambiguities and modeling flaws. Basic robustness criteria are, for example, absence of graphical transition non-determinism or absence of hazardous accesses to variables of the model. Also range violations in assignments to model variables are severe modeling flaws, which have to be detected and eliminated as early as possible in the development.

Another issue of interest already during the development is reachability of particular states or transitions. Formal proofs of reachability or unreachability of specific state configurations or variable valuations provide important hints regarding correctness of a model. Reachability checks can be applied in order to detect dead code as well as to produce specific simulation stimuli sequences.

Since model checkers can generate witness traces showing how a particular specification can be violated by a model, model checking can be utilized to systematically generate witnesses documenting computations of interest. This can be achieved by claiming a particular configuration of the model to be unreachable, and then using a model checker to falsify this claim. The resulting witness traces can be translated into simulation control programs, driving simulations of the model using the STATEMATE simulation tool. This way, model checking can be applied as a valuable and formal debugging aid.

Application of model checking for robustness analyses and reachability checking during the development needs to be integrated seamlessly with the development tools employed in the refinement iterations of the development. Since model analysis and formal debugging are to be applied by developers as part of their development activities, application of these techniques must not require too much expert knowledge. Checks to be performed can be defined by e.g. selection of graph-

ical states or transitions. From this selection a specification is generated, which claims that the selected transition respectively the selected combination of states is never reachable in the model. If this claim is violated, i.e. the situation specified this way is reachable in the model under construction, the developer is provided with a trace showing a computation which finally reaches the specified situation. Not only graphical states and transitions can be selected, but also arbitrary boolean expressions ranging over the variables and events of the model can be entered in order to define reachability analyses. Hence, model checking provides constructive - and definite - answers to questions a developer wants to ask about the model.

Proving functional and safety requirements is necessarily more complicated and requires higher effort than robustness checking and formal debugging. Requirements such as e.g. "a train must never pass an unsecured crossing" or "whenever a crossing is unable to report its status, the train will stop in a secure distance before the crossing" have to be formalized using an adequate specification formalism.

In order to ease the formalization of requirements and to require as little expert knowledge as possible, predefined temporal schemes - so-called *pattern* - are offered to the user of the STVE. Pattern are observer automata-templates encoding temporal relations between formal parameters. A concrete specification for verification can then be obtained by instantiating an appropriate pattern, with its formal parameters mapped to expressions referring to particular variables and states of the model.

Fortunately, many important requirements are of rather simple temporal nature - quite often referring to sequentiality or causality of not more than three observations[1].

For example, the requirement "a train must never pass an unsecured crossing" can be formalized using an instance of a `Q_only_after_P`-Pattern, where the formal parameter `P` is mapped to an expression indicating a secured crossing and `Q` mapped to an expression indicating a train passing the crossing.

Often assumptions about the environment need to be used in order to focus on specific situations in which requirements must be fulfilled. Therefore, all pattern offered by the STVE can be instantiated in assumption place as well as in commitment place, i.e. for formalization of system requirements.

It depends on the criticality of the requirement, whether conformance to particular temporal specifications needs to be repeatedly verified during the development iteration (after application of changes to the model). Alternatively, it may suffice to prove adherence to the requirement only at the final assessment at the end of the iteration. Definition of a proof-obligation once with the STVE, produces a proof script which can be re-executed whenever changes applied to the model make this desirable. Thus, requirement formalizations and generated proof scripts can be reused throughout the entire development phase.

Obviously, the usage of predefined pattern has its limits. In general, predefined pattern can not capture every requirement of interest. Although new custom-specific pattern can be provided by experts on demand - and have been added also during writing of this thesis, complicated user-defined specifications can better be formalized using the graphical specification formalism Symbolic Timing Diagrams (STDx), which is also integrated with the STVE. Symbolic Timing Diagrams provide an intuitive and easy comprehensible way of graphically specifying requirements in terms

[1] In section 2.2, we already cited [ESt97, 4-18], elaborating on the desired locality of important functionality. In particular, the more safety critical an embedded system is, the more clearly it should be structured. Safety critical functionality must not rely or depend on too much prerequisites and must not involve dependences on unreliable parts of the model. Following this guideline, the degree of inter-dependences is a quality measure for safety critical systems.

of interface observations of a model. Although their formally defined semantics aims at being intuitively understandable, some expert knowledge is required to capture requirements using STDx.

This chapter is organized as follows: In section 6.1, robustness analysis and formal debugging techniques - as offered by the STVE - are explained. Section 6.2 gives a comparative overview of the approaches to capturing user-defined requirements, as supported by the STVE through pattern and STDx. Pattern and STDx share the common semantical basis of Timed Symbolic Automata (TSA): Pattern are basically hand written instances of TSA observer modules , while STDx diagrams are unwound by an algorithm into TSA. From these TSA, synchronous observers are generated for application in verification.

The formalism and semantics of TSA is presented in section 6.3. Pattern are briefly presented in section 6.4. Section 6.5 concludes this chapter with the presentation of the graphical formalism STDx and the definition of a formal semantics semantics for STDx specifications that is based on an unwinding algorithm.

6.1 Robustness Analysis and Formal Debugging

As presented in section 3.1 STATEMATE offers powerful facilities for (graphical) development of models. A key issue of modeling using the STATEMATE tool set is executability of models in every state of concretization, in early phases of a development as well as for more matured models of later phases. Since models of earlier phases may be built in a possibly very abstract manner, STATEMATE permits non-determinism in various ways.

The STATEMATE simulator follows different resolution strategies in order to deal with such non-deterministic situations. A key feature of STATEMATE's simulation semantics is the parallel execution of independent parts of models. All activities are executed in parallel. Consequently, write accesses to data-items shared among activities may lead to multiple writer hazards, which are resolved by the simulator automatically by randomly choosing one of the possible results. Conflicts are only reported during simulation without prompting for user interaction.

In case of transition non-determinism the simulator follows a different strategy. For each simulation step the simulator determines which transitions are enabled, i.e. these transitions whose trigger expressions evaluate to true in the actual system state. From these transitions the simulator computes the maximal sets of non-conflicting transitions (cf. Basic Step Algorithm, section 3.2) . When the simulator detects more than one of such sets the user is prompted for interactive selection.

When using STATEMATE models as conceptual reference models for subsequent development phases, non-determinism contradicts the desired intuitiveness and unambiguouity. A model should be as unambiguous and robust as possible, in order to serve as reference model; non-determinism and hazards are sources of ambiguities and need to be detected and eliminated.

While the simulator detects and resolves non-deterministic situations when they occur during simulation, there exists no analysis within the STATEMATE tool-set detecting possible non-determinism or hazards in advance. In general, dynamical occurrence of non-determinism depends on certain valuations of variables or state configurations. For example, two transitions with triggers `'[i>=4]'` and `'[i<=4]'` for an integer variable `i` conflict only if the value of `i` is 4. For all other values of `i`, the triggers are mutually exclusive and the simulation behaves deterministically.

Another source of problems is specific to STATEMATE's asynchronous simulation semantics. As described in section 3.2, asynchronous stabilization is a dynamic property established upon synchronous execution of a series of steps. An asynchronous model becomes stable only if no internally

generated events or changes of data are to be processed any more. In general, there exists no upper bound to the length of step sequences to be performed unless becoming stable, but a model that does not stabilize under all circumstances must be treated as *divergent*. Unfortunately, the STATEMATE tool set itself offers no analysis for divergence detection, and also the simulator can not detect divergence.

Since time advances only in stable states of asynchronous models, sequences of steps collapse to a single point in time from the perspective of the asynchronous execution semantics. Thus, a class of possible hazards is related to asynchronous models: *sequential data hazards*. Read/Write or Write/Write accesses to data-items sequentially taking place within the same super-step of an execution must be treated as hazards from the perspective of time, although they are separated by delta delays. Again, STATEMATE's simulator reports such hazards when they occur during simulation, but like synchronous hazards, STATEMATE offers no analysis for detection of sequential hazards in advance.

Computing values and assigning them to data-items always may lead to range violation errors. A computed value may not fit into the range of a data-item, it might be less or greater than the lower or upper bound of the data-item's domain. Such range violations may lead to severe errors. Again, STATEMATE's simulator issues a warning only when a range violation occurs during simulation.

Analyses

Beyond syntactic checks, STATEMATE offers no systematic analyses for detecting possible modeling problems or errors in advance. Range violations, non determinism, data hazards and diverging super steps are detected only when they take place during simulation, otherwise they remain undetected. Hence, exhaustive simulation is the only strategy supported by STATEMATE to discover such erroneous situations. How many errors can be discovered by simulation depends on the coverage of the state space of a model. Systematic interactive simulation is time consuming, while there exists no warranty of detecting all errors in a model.

Formal verification can be used for a more sophisticated error detection. Because of the ability of a model checker to explore the entire state space of a model at once, model checking can systematically be utilized to analyze all possible runs of a model for erroneous situations.

The STVE offers specific and systematic analysis for particular modeling problems.:

- Stabilization Bound Check. From a technical point of view the bounded stabilization check differs from all other error analysis checks. This check aims at establishing a definite upper bound for the amount of steps an asynchronous model maximally performs until reaching a stable status. Hence, it must be verified that for all possible computations no sequence of steps is longer than an user defined upper bound without reaching a stable status. Informally, this can be achieved by extending the model with a counter, which is reset in stable states and incremented by one in each unstable step of the model. If the counter exceeds the upper bound for some computation, a witness showing this particular computation prefix is generated[2]. If otherwise the counter can never exceed the upper bound, freedom of super-step divergence under all circumstances has been formally proven for the model. As described in section 15, becoming stable is indicated by an asynchronous model through issuing a designated `SUPER_SYNC` event.

[2]This witness is translated into a STATEMATE simulation control program, which can be loaded and executed with STATEMATE's simulator.

Technically, stabilization bound analysis is a specialized application of pattern verification: the model is verified for a pattern instance of `inv_finally_P_B` with formal parameter `P` mapped to `SUPER_SYNC` and the bound-parameter `B` mapped to the user defined upper bound. Pattern verification will be discussed in section 6.4.

- Robustness Analyses. In contrast to stabilization bound checking, robustness checks are performed in two phases. In a static analysis phase, the model is inspected w.r.t. the chosen kind of possible error or non-determinism and annotated with dedicated additional outputs first, which indicate occurrence of the conflict under consideration. In the second phase reachability of conflict indication by these newly introduced outputs is examined.
  After model preparation it suffices to check for reachability of a truth valuation for the dedicated output *obs* in order to obtain the desired result. If the prepared model satisfies `AG`$\neg obs$, the potential conflict for which *obs* has been introduced does not occur for any computation of the model. If, on the other hand, the prepared model violates `AG`$\neg obs$, then there exists a computation along which the conflict arises. In case of reaching such a situation, the model checker produces a witness path which is translated into a simulation control program that can be animated using the STATEMATE simulator.
  In contrast to the stabilization bound check, robustness checks focus on rather local properties of the model, for which a Cone of Influence computation (cf. section 7.2) often yields a drastic reduction of the state space.

  - Write/Write and Read/Write Hazards: A Write/Write hazard can affect a data-item only if the control structure of the model representation permits more than one assignment to this data-item in the same step of the model. If there exists only one assignment to a particular variable or if the control structure of the model statically inhibits conflicting assignments to this variable, it can be excluded from further analysis. Otherwise, all assignments to the same variable which are not mutual exclusive due to the control structure must be treated as potentially conflicting. Each of these potential non-exclusive assignments is annotated with a newly introduced fresh auxiliary variable which is set to '1' when the assignment is executed. After annotation, each of the model variables which is potentially affected by multiple write accesses, is associated with a set of these auxiliary variables. Since auxiliary variables keep their value only within one step and are reset at the beginning of each step, their sum always represents the number of assignments to the associated model variable in the actual step. A new dedicated output is introduced which is set to 'true' if the sum of such a set of auxiliary variables is greater than '1'.
    At the end of the static analysis phase each potentially affected data-item is associated with a newly introduced output and the model is fully prepared for reachability analysis in order check for dynamic occurrence of conflicting write accesses.
    The user is offered a list of all data-items which are potentially affected by multiple assignments. For each user selection, reachability of a truth valuation of the associated new outputs obs_i is checked by claiming `AG`$\neg obs_i$.
    A similar analysis and annotation strategy is applied for detection of Read/Write Hazards: for Read/Write Hazard the preparation is implemented in a way that the introduced outputs associated with a particular data-item is set to true if at least one read and one write access take place within one step.

Recall, that conflicting data access hazards are captured by the simulation semantics of STATEMATE:

* Read/Write hazards are no real hazards in terms of the model, since STATEMATE's basic step algorithm guarantees a δ-delay for all assignments: read accesses always refer to the value of data-items at the beginning of a step, while write accesses do not change their values instantaneously. Write accesses to data-items only produce a *projected* new value, which will become visible first at the beginning of the following step. Although Read/Write hazards are no problem with respect to the simulation of a STATEMATE model, they can lead to misunderstandings or interpretation problems in later phases of the development process and hence, should be at least handled with care. Especially, when parts of a model - executed in parallel - trigger each other via producing and consuming data values, Read/Write hazards are often unavoidable. It depends on the subsequent development phases and the further use of the model, how these hazards have to be treated.
* Also Write/Write hazards are automatically resolved by the STATEMATE simulator. For conflicting assignments, the simulator chooses - using internal rules - an order of execution, s.t. the written value seems to be chosen non-deterministically to the developer. Since the result of conflicting write accesses depends on internal rules of the simulator, Write/Write hazards are undesirable even in terms of the model itself. From the perspective of a reference model, they must be treated as modeling flaws.

– Write/Write Hazards with different values: Write/Write hazards are reported by the STATEMATE simulator without regard of the written values. Designers often consider Write/Write hazards with equal values not to be harmful. But, even in these cases it is useful to establish evidence that the conflicting write accesses never assign different values to a data-item. In addition to Write/Write hazard detection as described above, all possible orders of execution of conflicting assignments must be checked and the assigned values have to be compared for this analysis.

– Transition Non-Determinism : multiple transitions starting in the same graphical state might be enabled if their triggers are non-exclusive according to the priority rules of STATEMATE[3]. In case of exclusive triggers, such as `'[not(cond)]'` and `'[cond]'` for a condition `cond` the affected transitions can be excluded from dynamic reachability analysis. For non-exclusive triggers, such as for example `'[i<3]'` and `'[j>2]'` only a dynamic reachability analysis can determine whether both expressions can evaluate to 'true' at the same step or not. The static analysis phase considers each scope of the model - i.e. each hierarchical state - and annotates all non-exclusive transition triggers of this scope with newly introduced outputs. The user is offered the list of scopes for which such outputs have been introduced. According to the user selection of scopes, again reachability analyses for the associated new outputs can be performed.

– Range Violations : The intermediate values of arithmetic expression are compared to the defined lower and upper bounds of the target domain before assigning the value to a variable and the comparison results are assigned to fresh auxiliary variables. Auxiliary variables indicating range violations for a particular data-item are collected in a disjunction. If any term of this disjunction becomes true,the domain definition of a data-item

[3] If transitions of different levels can be taken, STATEMATE priorizes the highest level transition (cf. section 3.1)

has been violated by an under- or overflow and a dedicated newly introduced output is set to true. For each of the variables guarded by such bound comparisons, a dynamic reachability analysis.r.t. the associated output can be performed. The model preparation needs to be performed only for variables which are assigned computed values or which are assigned values from variables with domains different from the target domain.

- Sequential Hazards: In contrast to hazards occurring within a single step, sequential hazard analysis has to memorize all accesses to variables between two successive stable states. Thus, it is unavoidable to introduce variables which memorize data accesses for a series of steps and which are reset in stable states. Except for this difference, the strategy is quite similar to the step hazard analysis.
- Analysis for sequential Write/Write hazards with different values is not offered at the moment of writing.

Often, application of analyses to the entire system model suffers from the complexity of the model. Thus, it is important to understand, that it is often sufficient to apply analyses only to sub-activities of the system model. Due to the compositional semantics [DJHP97], each sub-activity representation provides inputs for all variables and events which can be influenced from outside the sub-activity under consideration. All inputs from the surrounding model are unrestricted when considering an activity in isolation. Hence, analysis of an activity in isolation uses a strict over-approximation of the real interaction with the sibling activities, whereat over-approximation preserves unreachability results. E.g. if a potential transition non-determinism turns out to be unreachable already in an isolated component, it can in no case be reachable in the entire system model. The same counts also for data hazards: If a variable is not dynamically affected by e.g. a write/write hazard *within the minimal subsystem, which contains all write-accesses to the particular variable*, this result holds in the scope of the entire system model. Thus, identification of the minimal subsystem to which a particular analysis needs to be applied, often drastically reduces the required verification effort. Only if the particular analysis reveals a reachable conflict-situation for the selected scope, it might be necessary to apply the analysis to a larger part of the model. As a recommended methodology for analyses according to the compositional semantics, the developer should :

1. Identify potential problems by applying the static phase of the respective analysis to the top-level.

2. If the static phase of an analysis offers potential conflicts for selection, the developer should identify the minimal scope, i.e. the lowest hierarchy level, which contains the potential conflict and apply first the analysis to this scope. Either the model checker finds a witness for reachability of the potential conflict or the conflict is unreachable for all larger scopes in the system.

3. Only if a potential conflict turns out to be dynamically reachable for a particular scope, it might be necessary to apply the analysis also to the enclosing scope - if it is not desired or possible to eliminate the conflict already in the considered sub-system.

This methodology is applicable to all analyses except for stabilization bound checks. Unfortunately, always all parts of a model contribute to super-step stabilization. Hence, stabilization checks must be applied to the entire model and can not be established using simple compositional conclusions.

Formal Debugging Techniques

Beyond predefined analyses that are aimed at detection of conflicts in the model, the STVE offers also checks for reachability of states, transitions and particular valuations of variables. Reachability analysis for user defined model configurations can be used for dedicated generation of specific simulations. Witnesses obtained from these checks can be used to drive simulations into particular model configurations of specific interest. These simulations can for example be used as simulation prefixes to explore model behavior by starting interactive simulation with the reached configurations.

Robustness analyses are applied within an iterative cycle of finding a bug, fixing this bug and re-application of the analyses until hopefully no conflict remains in the model. Hence, the final goal of robustness analyses is to yield 'true' - results for all proof-obligations generated in the static analysis phases. Of course, reachability checks for user defined configurations can also be used for verification that e.g. an undesired combination of states can never be reached or that a variable can not be assigned an undesired value. But, in general, the *'simulation generation'* use-case is to utilize the model checker for generating witness traces of reachability for particular user defined situations. Hence - with few exceptions - the expectation is to obtain a 'false'-result and a simulation driving the model into the specified configuration.

In particular, the STVE supports the following formal debugging applications:

- Drive to property (white box view): A list containing all basic states, all variables, events, and conditions known to the user, plus the scheduling primitives (cf. section 15) is offered to the user. Using objects of this list, the user can enter an arbitrary STATEMATE-like expression that defines the goal for reachability analysis. Since this expression is entered in a STATEMATE-like syntax, it needs to be converted into an adequate SMI-expression first. Second, the resulting SMI expression is negated for formulation of an invariant. Verifying the formula "`AG`(¬<user-defined-expression>), then yields the desired result: if the configuration specified by the expression is reachable by some computation prefix of the model, the model checker provides a witness leading to the user-defined configuration.
  The expression language supported for user defined expressions is oriented towards the expression language supported by STATEMATE for transition triggers, except for the separation of events on the one-hand, and data-items on the other hand. STATEMATE only permits transition triggers of the form '*e*[*c*]' , where *e* may be an expression referring to events only and *c* is an arbitrary boolean expression which must not refer to events. For definition of a 'drive to property' goal, events can be used like boolean variables in the user-defined expression. Written and changed events (`wr()`, `ch()`) for data-items as well as *true* and *false* events for conditions are supported ('becoming true' is denoted by `tr()`, 'becoming false' is denoted by `fs()`). Regarding states, *entering*, *exiting* and being *in* a state can be specified by `en()`, `ex()` and `in()` , respectively. In addition to the expression language supported by STATEMATE, the STVE also supports references to values of events, variables and states of the previous step. Using `last()` in an expression of a drive-to-property check (as well as in expressions mapped to pattern-parameters), causes the STVE to introduce an additional variable. This variable memorizes in every step the 'last-step'-valuation of the affected expression[4]. In order to refer to a newly computed value of a data-item, an event to be emitted or a state to be entered in the next step, `primed()` can be used when defining a drive-to-property check. In contrast to `last()` , `primed()` can not be nested.

[4] `last()` can arbitrarily be nested in expressions.

- Drive to transition : A flattened view of transitions is offered to the user for selection. While in STATEMATE graphical transitions can start and end in arbitrary AND-states of the state hierarchy, the SMI representation differs a bit. Hierarchical states are not explicitly represented in the SMI representation. Each AND state can be characterized by an expression ranging over basic states. Consider for example the simple statechart shown in figure 6.1. The AND state `A` consists of the or states `C` and `B`. `A` is active if `C` as well as `B` are active. In contrast, `ST_TOP` is an OR states. States `A` and `C` are never active at the same instance of time. Since e.g. `C1` and `C2` (`B1` and `B2`, respectively) are active exclusively, `A` being active can be characterized by `'(C1 or C2) and (B1 or B2)'`. Hence, in the flattened view, transition `A` →`D` is represented by the flat transitions `C1` →`D`, `C2` →`D`, `B1` →`D` , and `B2` →`D`. Note that since the default transitions of `C (B)` enters `C1 (B1)`, in the flattened view the transition `D` →`A` is split into only the two transitions `D` → `C1` and `D` → `B1`. While in STATEMATE a so-called compound transitions may

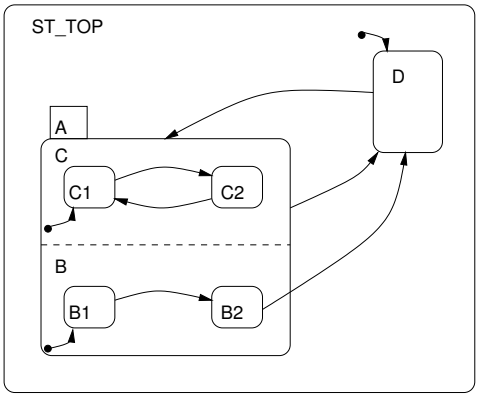


Figure 6.1: Simple Statechart

consist of transition fragments connected via condition connectors, in the flattened view all possible transitions (resulting from the possible decisions along a compound transition) start and end in basic states.
Part of the compilation of STATEMATE models into their SMI/SSL representation is the annotation of each flattened transition with a dedicated output, which is set to '1' whenever the transition is taken. Hence, "drive to transition" can be performed by searching a computation prefix for which the associated output finally becomes true. As for a "drive to property", this is realized with an invariant specification "`AG`($\neg$<transition-output>$= false$).

- Drive to state : "drive to state" is a specialization of "drive to property", where only basic states can be selected by the user. For multiple selection, multiple proof-obligations are generated. Each of the proof-obligations defines a reachability analysis task for one of the selected basic states.

- Drive to configuration : "drive to configuration" is like "drive to state" a specialization of "drive to property", where only basic states can be selected. In contrast to "drive to state", a multi-selection defines only one proof-obligation for a conjunctive combination of all selected basic states.

Due to the parallelism of STATEMATE models, it is sometimes very difficult to manually drive

simulations into a model configuration of specific interest. The situation of interest may be reachable only if a large set of internal triggers is provided, each at the exact timing. To drive simulations manually is time consuming and requires knowledge about internal communications of the model. Debugging aid by formal verification provides specific simulations on demand: if there exists any computation driving the model to a desired internal configuration, "drive to ..." techniques will discover the shortest prefix. Unreachability of internal configurations is revealed and can be further analyzed using the debugging techniques offered by the verification framework.

Like in the analysis use-case, all proof-obligations for debugging checks define invariant specifications. Hence, analysis and debugging checks can be realized by invariance checking (cf. section 4.7). Since in the context of debugging, reachability of the specified configuration is expected, also bounded model checking can be applied for debugging purposes. If there exists a run of the model finally reaching the specified configuration, bounded model checking will find this path often faster than invariance checking. On the other hand, if bounded model checking does not find a path violating the invariant within a user-defined upper bound, either the configuration is unreachable in principle or the user-defined bound has been chosen too small. In order to obtain a definite answer to this question, the formal debugging check can be re-executed with a larger bound or using invariance checking.

6.2 Certification Techniques

In a model based development process, models are refined in numerous iterations. Modeling starts with an abstract functional decomposition of a system's specification. With progressing development, the functional parts become more and more detailed and concrete. As the model's granularity and detailing increases, its overall functionality may become more and more subtle. Besides iterated application of robustness analyses and formal debugging checks in order to detect model flaws, it is therefore important to ensure that a refined model conforms to behavioral and functional requirements.

For example - in the Radio-based Signaling System - a functional requirement is: "*A train passes a crossing only after it has been secured*". In particular, the train may pass the crossing *only after* the crossing *has reported itself safe* or *has been released manually.*

It is a severe error, if the model does not fulfill this requirement under all circumstances. In contrast to robustness properties, model-specific properties can not be examined using predefined fully automatic analyses. Also, such a temporal relation between observations can, in general, neither be captured using a 'drive to property' definition nor can be expressed in terms of reachability of particular transitions and/or states.

Functional requirements must be formalized as (temporal) specifications in terms of observable behavior. Demand for sequentiality, causality or exclusiveness of observations has to be specified using adequate formalisms. In contrast to formal debugging which aims at witness generation, the goal of requirement verification is to verify that a model meets its requirements. Therefore, it is not sufficient to generate a witness of how the model can engage in a particular computation which is consistent with the requirement under consideration. In fact, the goal is to prove that the model can never engage in any computation violating the requirement. Hence, verifying a *universal* specification for all possible behaviors of a mode is oriented towards a "true" result. In particular, formal verification is used for *model certification.*

Late detection of design errors or flaws may become very expensive. Requirement verification of

the product under development only after completion of an entire development phase can enforce re-iteration of development steps. In order to avoid deep iterations in the development of safety critical systems, evidence for compliance with requirement specifications is of key importance and should be established even during development iteration, if possible whenever changes to the model have been applied. On the other hand, when completing a development phase and submitting the model to quality assurance, verification can be applied in order to assess the model. If problems or errors are discovered in the assessment phase, acceptance is rejected and the model is handed back to development. The effort spent for validation and verification in the product assessment phase can be reduced by antedating parts of the verification activities to the development phase itself.

Robustness analysis and formal debugging accompany the engineer during development and help to avoid incorrect modeling. How about verification of important requirements? In order to facilitate engineers work in practice, verification application must not require too much expert knowledge. Establishing evidence for meeting the requirement specifications must be as easy as possible. Thus, an easy-to-use specification formalism to capture requirements is needed. Having captured a requirement once, verification for subsequent development steps should best be applicable using a push-button technique.

Fortunately, many important safety critical requirements that an embedded control system must satisfy, are of rather simple temporal nature. For example, the requirement *"A train passes a crossing only after it has been secured"*, simply relates two observations.

In a model based development process, the model should be built in a way that indicators that are needed for the assessment of critical behavior are easy to observe. This can be achieved by introducing particular error-states or outputs indicating designated valuations of data-items. Since the goal of the model based development is to capture critical functionality, it is a measure for the quality of the model, how clear and definite correctness of behavior can be specified and decided[5].

As a rule of thumb, it can be stated: the more important a requirement is, the less subtle details should contribute to its validity. Hence, it should be necessary to only take few details into account in order to formalize the requirement specification. If some property is required to hold under all circumstances, it should not be necessary to specify detailed sequences of particular observations, in order to capture the requirement.

The verification framework copes with the specification problem with different strategies. Experience has shown that many specifications can be reduced to a relatively small set of temporal schemes [Bit00, Bit01, DAC98a, DAC98b, Hol05]. Typical informal requirements consist of formulations like "*observation a should be possible only after observation b*" or "*observation a causes observation b*". For this purpose, the verification framework offers a library of predefined temporal schemes - so-called *pattern* - as building blocks for requirement specification. In order to formalize a requirement, only appropriate pattern need to be instantiated and their formal parameters be mapped to expressions ranging over the observables of the model. Pattern are dedicated to on-the-fly verification during development. In order to be used in the most flexible way and to keep the

[5] Recall from chapter 2 the regulation of requirement allocation to components of a design according to the V-model:

- "Every requirement must be allocated to at least one element of the technical architecture, ideally exactly to one element
- ....
- The allocation must be realized in such a manner that it will be possible to prove the fulfillment of the requirement by checking the corresponding architecture element."

learning effort for their usage as low as possible, pattern can be used in a white box view of the model, i.e. there is no restriction w.r.t. visibility of (local) data-items, internal scheduling primitives or basic states. When mapping the formal parameters of a pattern to STATEMATE-like expressions, all user-defined interna of a component can be referred to. Pattern can be instantiated for use in commitment as well as in assumption place.

Due to the support of a white-box view, only component verification is supported for pattern specifications. In particular, no compositional techniques or hierarchical reasoning are supported. Furthermore, counters in pattern refer to either simulation time or to steps. The treatment of time depends on the chosen semantics. For asynchronous models time is uniquely measured by counting super-steps, for synchronous models time corresponds to steps.

While pattern are easy to use and are reasonable expressive for on-the-fly verification of many important requirements, capturing more sophisticated requirement specifications requires a more flexible formalism. Of course, the most flexible way of specifying requirements would be to enter specifications manually as temporal logic formulae. Besides requiring much expert knowledge about the model representation and semantical subtleties of temporal logics, a drawback of formulae is their lack of intuitiveness. Being more intuitive than temporal logic formulas, graphical specification formalisms have shown to be useful for the formalization of requirements.

For this purpose, the graphical formalism of Symbolic Timing Diagrams (STDx) has been integrated with the verification framework. Requirements in terms of model observables and their value changes over time can be expressed graphically and be intuitively understood from their visualizations. The temporal relationship of observations can be specified by graphical constraints in a variety of different, but well defined meanings. Thereby, constraints can specify pure qualitative as well as quantitative temporal relations between observations.

In contrast to patterns, STDx-specifications adopt a black-box view of the system or its components. STDx-specifications refer to behavior of a (sub-)component as observable at the interface, without referring to internals of the (sub-)component. Due to the representation of STATEMATE models according to the compositional semantics, even sub-component interfaces for internal activities of an asynchronous system in general contain both slow and fast inputs (cf. section 5.4). Since the asynchronous semantics is defined on the basis of synchronous execution, quantitative timing constraints must be capable of both referring to steps and to super-steps.

Commitments as well as assumptions can be specified using the same graphical formalism. Assumption and commitment diagrams are grouped in STDx-specifications. A formal semantics of STDx is defined by unwinding the diagrams into timed symbolic automata (TSA). From TSAs, observer modules are generated which are then combined with the model for verification.

Since also pattern are basically a special tailorization of timed symbolic automata observers, we will introduce TSA formally in the following section. In section 6.4 we will give a brief overview of the patterns offered by the verification environment. STDx-specifications and their formal semantics through unwinding into TSA representations will be presented in section 6.5.

6.3 Timed Symbolic Automata (TSA)

In a discrete time domain computations of a system can be described by sequences of consecutive valuations of the system's variables, where each of the valuations in such a sequence is a unique characterization of the system's status at a particular instance of time. Moreover, for finite state systems - i.e. systems whose variables all have finite domains -, every status can be described by a

finite set of atomic propositions. This way, computations can be treated as infinite words over the *alphabet of truth assignments* to these atomic propositions.

A well studied tool for reasoning about such words, i.e deciding whether a particular word belongs to a designated subset of words - a language - over an alphabet, are finite automata. Informally, finite automata are state-transition machines with designated sets of initial and accepting states. The transitions of such an automaton are triggered by symbols of the alphabet. This way, a finite automaton processes a word by reading it symbol after symbol. Starting in an initial state the automaton reads a symbol and takes a suiting transition - if there exists at least one - from its currently active state to a successor state. A word is accepted by the automaton if by reading the entire word at least one of the accepting states is reachable. The *language* accepted by the automaton is exactly the set of words for which the automaton can finally terminate in an accepting state.

Since the definition of acceptance is based on finiteness of computations, conventional finite automata are applicable only to finite words. Systems considered in the context of this work are reactive, i.e. non-terminating, their computations can not be captured by finite but only by infinite sequences of valuations. Obviously, when considering infinite words, termination can no longer be an adequate acceptance criterion.

Therefore, finite automata have been adapted to be capable of reasoning also about infinite words, where the major change applied to finite automata is the definition of a different acceptance criterion. Even though termination in a final state is no longer an adequate acceptance criterion, acceptance can nonetheless be defined in terms of a subset or subsets of automata states:

Obviously, when processing infinite words with finite-state automata, some of their states must be active infinitely often - otherwise processing infinite words would require infinitely many states.

Various different acceptance criteria for automata on infinite words in terms of - sets of - states being active infinitely often can be found in the literature. An excellent overview of the theory of automata on infinite objects can be found e.g. in [Tho90].

The most basic definition of automata on infinite words is the definition of *Büchi-automata*, whose acceptance criterion is defined using a subset of the state set, the so-called *fair states* set. An infinite word is accepted by a Büchi automaton if at least one of the fair states is active infinitely often when processing the word.

A common notation for sets of infinite words over a fixed alphabet extends the well known regular languages with the concept of countable but infinite repetition. Similar to the Kleene star '$\star$', which denotes finite repetition of a regular pattern in a word, 'ω' is used to denote countable - but also infinite - repetition of such pattern. Like the relation between regular languages and finite automata, the definition of ω-regular languages is given by their relation to Büchi automata: A set of infinite words is called a *ω-regular language* iff there exists a Büchi automaton, which accepts all words belonging to that set.

Definition 6.1 (Büchi Automaton)

Formally, a Büchi automaton is a tuple $\mathcal{B} := (X, S, S_0, \rightarrow, F)$, where

- X is an alphabet,
- S is a finite set of states of $\mathcal{B}$,
- $S_0 \subseteq S$ is a set of initial states,

- $\rightarrow \subseteq S \times X \times S$ is a transition relation, where (s, c, s') represents a transition from state $s \in S$ to state $s' \in S$ which is triggered by $c \in X$. The transition can be taken if s is the currently active state of $\mathcal{B}$ and the actually read symbol is c.
- $F \subseteq S$ is a set of fair states.

Let $w := c_0c_1c_2...$ be an infinite word over X. A run of $\mathcal{B}$ over w is an infinite sequence $\Pi := s_0s_1s_2...$ of states, s.t.

$$s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} s_2 \xrightarrow{c_2} ... \text{ where } s_0 \in S_0 \text{ and } \forall i \geq 0 : (s_i, c_i, s_{i+1}) \in \rightarrow .$$

Let $run_{\mathcal{B}}(w)$ denote the set of possible runs of $\mathcal{B}$ for the infinite word w. Let furthermore $inf(\Pi) \subseteq S$ denote the set of states which occur at infinitely many positions in Π. Then, the language $\mathcal{L}(\mathcal{B})$ accepted by $\mathcal{B}$, is defined by

$$\mathcal{L}(\mathcal{B}) := \left\{ \begin{array}{l} w = c_0c_1c_2... \in X^\omega \quad | \\ \quad \exists \Pi = s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} s_2 \xrightarrow{c_2} ... \in run_{\mathcal{B}}(w) : inf(\Pi) \cap F \neq \emptyset \end{array} \right\}$$

■

$\mathcal{L}(\mathcal{B})$ can alternatively be defined using a characterization in temporal logic :

$$\mathcal{L}(\mathcal{B}) := \left\{ \begin{array}{l} w = c_0c_1c_2... \in X^\omega \quad | \\ \quad \exists \Pi = s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} s_2 \xrightarrow{c_2} ... \in run_{\mathcal{B}}(w) : \mathcal{B}, \Pi \models \mathtt{GF} \left(\bigvee_{s \in F} s \right) \end{array} \right\} \quad (6.1)$$

The definition given in 6.1 is that of a non-deterministic Büchi Automaton. The automaton $\mathcal{B}$ is deterministic if S_0 is a singleton and if $\rightarrow$ is a (partial) function, i.e. if for each state $s \in S$ and for all symbols $c \in X$, there exists at most one successor state $s' \in S$ of s, s.t. $(s, c, s') \in \rightarrow$.

$\mathcal{B}$ is called complete if for all states $s \in S$ and for all symbols $c \in X$, there exists at least one state $s' \in S$, s.t. $(s, c, s') \in \rightarrow$.

In contrast to its counterpart on finite words - regular finite automata - , the class of deterministic Büchi automata is strictly less expressive than the class of non-deterministic automata (e.g. there exists no deterministic but only a non-deterministic Büchi automaton accepting the language $(ab)^*a^\omega$ for alphabet $X = \{a, b\}$). The class of deterministic Büchi automata is closed under union and intersection, but in contrast to regular finite automata is not closed under complementation. Although the class of non-deterministic Büchi automata is also closed under complementation, there exist no convenient way to construct the Büchi automaton complement of a Büchi automaton. Nonetheless, there exists a convenient complementation for arbitrary Büchi automata. Using the linear temporal logic characterization of acceptance as in equation (6.1), a complement automaton can be constructed by complementing the acceptance criterion. The resulting automaton is not a Büchi automaton:

$$\widetilde{\mathcal{L}(\mathcal{B})} := \left\{ \begin{array}{l} w = c_0c_1c_2... \in X^\omega \quad | \\ \quad \forall \Pi = s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} s_2 \xrightarrow{c_2} ... \in run_{\mathcal{B}}(w) : \mathcal{B}, \Pi \models \mathtt{FG} \left(\neg \bigvee_{s \in F} s \right) \end{array} \right\}$$

Usually, Büchi-automata are defined w.r.t. a fixed alphabet X as in the definition above. Since they allow one element of the alphabet per transition, they operate on single input symbols only.

In order to be able to capture the communication behavior of a reactive system, it is necessary to permit more than one observation per transition. This could be obtained by defining the automaton w.r.t. a sufficiently large alphabet, s.t. all combinations of observations are captured by symbols of the alphabet.

As an alternative approach relaxing the restriction of transitions triggered by single symbols, Schlör introduced *symbolic automata* [Sch00], which extend classical Büchi automata by admitting expressions as transition labels. In symbolic automata, transitions are labeled with boolean predicates ranging over the set of atomic propositions w.r.t. a set of variables. Correspondingly, the semantics definition of symbolic automata has to take satisfaction of predicates triggering the transitions into account.

Definition 6.2 (Predicates over a Set of Variables)

Let $\mathcal{V}$ be a set of variables $\mathcal{V}$. Then, predicates $\beta \in Pred_{\mathcal{V}}$ ranging over $\mathcal{V}$ are built according to :

$$\beta := \alpha \,|\, true \,|\, \neg\beta \,|\, (\beta) \,|\, \beta_1 \wedge \beta_2 \text{ , where}$$

α is an atomic proposition w.r.t. $\mathcal{V}$.

Let $\Sigma_{\mathcal{V}}$ denote the possible valuations of the variables $\mathcal{V}$ (cf. definition 4.1).

With the usual abbreviations

- $false{:=}\neg true$
- $\beta_1 \vee \beta_2 \quad :\Leftrightarrow \quad \neg(\neg\beta_1 \wedge \neg\beta_2)$
- $\beta_1 \Rightarrow \beta_2 \quad :\Leftrightarrow \quad \neg\beta_1 \vee \beta_2$

satisfaction of β w.r.t. $\sigma \in \Sigma_{\mathcal{V}}$ is defined by:

$$\forall \sigma \in \Sigma_{\mathcal{V}} : \sigma \models true$$

$$\begin{array}{rcl}
\sigma \models \alpha & :\Leftrightarrow & \sigma(\alpha) = true \\
\sigma \models \neg\beta & :\Leftrightarrow & \neg(\sigma \models \beta) \\
\sigma \not\models \beta & :\Leftrightarrow & \sigma \models \neg\beta \\
\sigma \models (\beta) & :\Leftrightarrow & \sigma \models \beta \\
\sigma \models \beta_1 \wedge \beta_2 & :\Leftrightarrow & \sigma \models \beta_1 \wedge \sigma \models \beta_2
\end{array}$$

We refer to the truth value of predicate β by writing $[\![\beta]\!](\sigma){:\Leftrightarrow}\sigma \models \beta$.

■

Symbolic automata permit a more flexible treatment of transition labels than conventional Büchi-automata. They are well suited for algorithmic transformations: For example, transition labels of newly introduced transitions can be constructed from boolean combinations of already existing transition labels. We will make use of this facility in the algorithm for unwinding timing diagrams (cf. section 6.5.4).

In [Sch00] symbolic automata are formally defined with a rigorous semantics, and a correspondence relation between symbolic automata and propositional linear temporal logic is established.

Time is treated only in a qualitative way throughout [Sch00], without referring to concrete quantitative measures of model time, such as intervals or specific temporal distances between observations.

In order to refer to time also in a quantitative way, we need to extend symbolic automata with a more concrete notion of time. For this purpose, we borrow the concept of *specification clocks* and *clock constraints* from Timed Automata [AD91, Alu98]: While the behavior of a symbolic automaton only depends on readings of input valuations, a symbolic automaton aimed at *reasoning quantitatively about timed observations* has somehow to count time, since enabledness of transitions depends also on the (relative) occurrence time of observations. This is realized by specification clocks, which can be set to zero at the transitions of an automaton and count the time since their last reset. Basically, specification clocks are variables capturing non-negative integer values, representing instances of time. As a discrete time model is assumed, clocks are increased by discrete portions according to the advancing time. Specification clocks are interpreted w.r.t. *clock environments*, which will be formally introduced in definition 6.5.

Thus, quantitative reference to time is introduced into symbolic automata by augmenting transitions by clock resets and clock constraints formulated over the set of clocks.

In STATEMATE's synchronous execution semantics, there exists a direct association between time units and steps of the execution. In contrast, STATEMATE's asynchronous execution semantics imposes a more complicated treatment of time, because the execution semantics is built upon synchronous execution (cf. section 3.2). On the one hand, synchronous steps are executed strictly sequentially with a so-called delta-delay. On the other hand, simulation time passes only between stable states of the asynchronous model. Several steps take place sequentially at the same instant of simulation time, whereas in the context of compositional verification the sequentiality of step execution can not be disregarded. Interfaces of sub-components in asynchronous models are in general comprised of *slow* inputs that can change only in stable states as well as of *fast* inputs that can change every step (cf. section 5.4) . For example, even becoming stable depends on step based interaction between the components of a system, which entirely takes place at the same instant of time.

As we have hence to regard both delta delays and simulation time for compositional verification of asynchronous STATEMATE models, we have to distinguish between two sorts of clocks: one sort referring to simulation time and another sort counting δ-delays.

Before we give the formal definition of Timed Symbolic Automata we first have to define legal expressions for clock constraints.

Definition 6.3 (Clock Constraints)
Let C be a set of specification clocks. Let C be partitioned into two (possibly empty) subsets:

- C_{step} is a set of clocks referring to steps of the model, and
- C_τ is a set of clocks referring to the time sequence portion τ of a timed observation sequence

Specification clocks may be compared only to non-negative integer constants and not to each other. Hence, legal clock constraints $\gamma \in \Gamma(C)$ are of the form:

$$\gamma := c \sim m \text{ , where } c \in C, m \in \mathbb{N}_0 \text{ , and } \sim \in \{\leq, <, =, >, \geq\}$$

Let $\Gamma(C_{step}) \subseteq \Gamma(C)$ denote the set of clock constraints referring to clocks $c \in C_{step}$ and let $\Gamma(C_\tau) \subseteq \Gamma(C)$ denote the set of clock constraints referring to clocks $c \in C_\tau$.

From $C = C_\tau \cup C_{step}$ follows $\Gamma(C) = \Gamma(C_\tau) \cup \Gamma(C_{step})$.

■

In the context of this work it is sufficient to permit comparison of clocks to constants in order to refer to the actual value of a particular clock.

Based upon these basic clock constraints, predicates referring to clocks are built using boolean connectives. Notice that definition 6.4 preserves the restriction that clocks may only be compared to constants but not to other clocks. Although different clocks can be compared to constants within the same timing predicate, they may never be compared with each other.

Definition 6.4 (Timing Constraint Predicates)
Let ψ be a predicate ranging over clock constrains $\gamma \in \Gamma(C)$:

$$\psi{:=}\gamma \,|\, true \,|\, \psi_1 \wedge \psi_2 \,|\, \neg\psi \,|\, (\psi) \quad \text{, with}$$

$\gamma \in \Gamma(C)$. We use the abbreviations:

- $false{:=}\neg true$
- $\psi_1 \vee \psi_2 \quad :\Leftrightarrow \quad \neg(\neg\psi_1 \wedge \neg\psi_2)$
- $\psi_1 \Rightarrow \psi_2 \quad :\Leftrightarrow \quad \neg\psi_1 \vee \psi_2$

Let $\Psi(C)$ denote the set of timing-predicates $\psi \in \Psi(C)$ built from clock constraints $\Gamma(C)$ according to the above syntax.

■

Specification clocks $c \in C$ are interpreted w.r.t *clock environments,* which are formally defined by the following definition:

Definition 6.5 (Clock Environments)
A clock environment ξ is an interpretation

$$\xi : C_\tau \cup C_{step} \rightarrow \mathbb{N}_0,$$

which assigns each clock $c \in C_\tau \cup C_{step}$ a value of the time domain $\mathbb{N}_0$.

Let ξ be the combination of the two functions

$$\xi_{step} : C_{step} \rightarrow \mathbb{N}_0 \quad , \quad \xi_\tau : C_\tau \rightarrow \mathbb{N}_0.$$

The set of all clock environments regarding C is defined by Ξ_C.

The valuation of a timing constraint ψ ranging over clock constraints $\gamma \in \Gamma(C)$ w.r.t. a clock environment $\xi \in \Xi_C$ is defined by :

$$[\![.]\!](.) : \Psi(C) \times \Xi_C \rightarrow B, \quad \text{s.t.}$$

- $\forall \xi \in \Xi_C : [\![true]\!](\xi){:=}true$
- $[\![\neg\psi]\!](\xi){:=}\neg[\![\psi]\!](\xi)$
- $[\![\gamma]\!](\xi){:=} \begin{cases} [\![\gamma]\!](\xi_\tau) & \gamma \in \Gamma(C_\tau) \\ [\![\gamma]\!](\xi_{step}) & \gamma \in \Gamma(C_{step}) \end{cases}$

- $[\![\gamma]\!](\xi_\tau) := \begin{cases} \xi_\tau(c) = m & \text{if } \gamma = (c = m) \\ \xi_\tau(c) \leq m & \text{if } \gamma = (c \leq m) \\ \xi_\tau(c) \geq m & \text{if } \gamma = (c \geq m) \\ \xi_\tau(c) < m & \text{if } \gamma = (c < m) \\ \xi_\tau(c) > m & \text{if } \gamma = (c > m) \end{cases}$, for some $c \in C_\tau$, $m \in \mathbb{N}_0$

- $[\![\gamma]\!](\xi_{step}) := \begin{cases} \xi_{step}(c) = m & \text{if } \gamma = (c = m) \\ \xi_{step}(c) \leq m & \text{if } \gamma = (c \leq m) \\ \xi_{step}(c) \geq m & \text{if } \gamma = (c \geq m) \\ \xi_{step}(c) < m & \text{if } \gamma = (c < m) \\ \xi_{step}(c) > m & \text{if } \gamma = (c > m) \end{cases}$, for some $c \in C_{step}$, $m \in \mathbb{N}_0$

- $[\![\psi_1 \wedge \psi_2]\!](\xi) := [\![\psi_1]\!](\xi) \wedge [\![\psi_2]\!](\xi)$

Elapsing time increases *all clocks of one sort* (interpreted w.r.t. ξ_{step} and ξ_τ respectively) by the same amount of time, denoted by $((\xi) \oplus (t_\tau, t_{step})) : C_\tau \cup C_{step} \to \mathbb{N}_0$ for $(t_\tau, t_{step}) \in \mathbb{N}_0^2$:

$$(\xi \oplus (t_\tau, t_{step}))(c) \quad := \quad \begin{cases} \xi_\tau(c) + t_\tau & \text{if } c \in C_\tau \\ \xi_{step}(c) + t_{step} & \text{if } c \in C_{step} \end{cases} \tag{6.2}$$

Let $\xi_{[c:=t]}$ denote the clock environment that agrees with clock environment ξ on all clocks except for c; c is set to $t \in \mathbb{N}_0$.

■

6.3.1 Timed Symbolic Automata

Using the above definitions, we now formally define Timed Symbolic Automata[6]:

Definition 6.6 (Timed Symbolic Automaton (TSA))
A Timed Symbolic Automaton is a tuple
$\mathcal{A} := (\mathcal{V}, S, s_0, C, T, F)$, where

- $\mathcal{V}$ is a set of variables.
- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $C = C_\tau \cup C_{step}$ is a set of clocks. The sets C_τ and C_{step} are disjoint. All clocks $c \in C$ have the the domain $\mathbb{N}_0$.

[6] A similar definition - ad hoc, without the formal basis of symbolic automata - was given in [Fey96] in order to define the semantics of an earlier Real-Time version of Symbolic Timing Diagrams. In [Klo03], Klose uses a similar definition as basis of the semantics definition of Live Sequence Charts. Both definitions use only one sort of clocks.

- $T \subseteq S \times Pred_{\mathcal{V}} \times S \times 2^C \times \Psi(C)$ is a transition relation.
  A transition $t = (s, enable, s', clocks, timing)$ represents a state-change of $\mathcal{A}$ from state s to s' for an observation satisfying predicate *enable* and consistent with timing constraint *timing*, where
  - $enable \in Pred_{\mathcal{V}}$ is a predicate ranging over the atomic propositions AP w.r.t. $\mathcal{V}$.
  - $timing \in \Psi(C)$ is a predicate ranging over clock constraints $\Gamma(C)$.
  - $clocks \subseteq C$ specifies the set of clocks to be reset when taking the transition.
- $F \subseteq S$ is a set of Büchi accepting states.

■

The semantics of TSA is defined over timed observation sequences $ts = (\pi, \tau)$ (c.f. definition 5.6), where $\pi = \sigma_0\sigma_1...$ is a sequence of valuations of the variables $\mathcal{V}$ and $\tau = \tau_1\tau_2...$ is a time sequence.

Specification clocks either count units of model time or steps of a model. They can be reset when taking transitions of a symbolic automaton, and can be referred to at transitions using clock constraints. A transition annotated with a clock constraint is enabled only if both its *enable*-predicate is satisfied for the current valuation of model variables, and the *timing*-predicate is satisfied in the actual clock environment. Hence, acceptance of a timed observation sequence by a TSA depends on the observation sequence σ of a timed observation sequence, as well as on the relation between the clock environment ξ and the time sequence τ of ts. We write $(s_i, \xi_i) \overset{\sigma_i}{\underset{\tau_i}{\rightarrow}} (s_j, \xi_j)$ to describe the correspondence of timed observation sequence $ts = (\pi, \tau)$ and the reaction of TSA $\mathcal{A}$: for the i-th valuation σ_i with i-th time stamp τ_i, $\mathcal{A}$ takes a transition from state s_i and clock environment ξ_i to state s_j and ξ_j. Sometimes it might also be useful, to annotate the *index* i of σ_i and τ_i to this notation, because clocks of C_{step} refer to the position of valuation σ_i in the observation sequence portion π of ts. In order to explicitly refer to the position of σ_i in π, we write $(s_i, \xi_i) \overset{\sigma_i}{\underset{(\tau_i,i)}{\rightarrow}} (s_j, \xi_j)$.

Definition 6.7 (Semantics of TSA)
Let be given a TSA $\mathcal{A} = (\mathcal{V}, S, s_0, C_{step} \cup C_\tau, T, F)$
A timed observation sequence ts is accepted by $\mathcal{A}$ if at least one state $s \in F$ is active infinitely often when processing ts:

- Let $run_{\mathcal{A}}(ts)$ be the set of recorded sequences (timed runs) of pairs (s_i, ξ_i) , $i \geq 0$, where $s_i \in S$ is the i-the state and ξ_i is the i-th clock environment of the run for processing timed observation sequence ts (if such sequence exists).
  For timed run $r \in run_{\mathcal{A}}(ts)$,

$$r = (s_0, \xi_0) \overset{\sigma_0}{\underset{(\tau_0,0)}{\rightarrow}} (s_1, \xi_1) \overset{\sigma_1}{\underset{(\tau_1,1)}{\rightarrow}} (s_2, \xi_2) \overset{\sigma_2}{\underset{(\tau_2,2)}{\rightarrow}} ...$$

- The following holds:
  - $\forall c \in C : \xi_0(c) = 0$ (initially the clock environment is 0 for all clocks)
  - $\forall i \geq 0 : \exists (s_i, enable, s_{i+1}, clocks, timing) \in T$, s.t.

  - $(\sigma_i \models enable) \wedge [\![timing]\!](\xi_i) = true$
    (a transition can be taken only if both its *enable*-predicate and its *timing*-predicate evaluate to true)
  - $\forall c \in clocks : \xi_{i+1}(c) = 0$
    (the clock environment is set to 0 for all clocks reset at a transition)
  - $\forall c \notin clocks : \xi_{i+1}(c) = \xi_i(c) \oplus ((\tau_{i+1} - \tau_i), 1)$
    (the clock environment is updated for all other clocks, which are not reset at the actual transition. Clocks $c \in C_\tau$ are updated according to time sequence τ , whereas clocks $c \in C_{step}$ are updated according to the position of the actual observation in π.)

- For $r \in run_{\mathcal{A}}(ts)$ let $inf(r) \subseteq S$ denote set of states which are visited infinitely often along r. Then, the language accepted by TSA $\mathcal{A}$ is defined by :

$$\mathcal{L}(\mathcal{A}) = \{ts \quad | \quad \exists r \in run_{\mathcal{A}}(ts) : inf(r) \cap F \neq \emptyset\}. \tag{6.3}$$

- Let $runs(\mathcal{A})$ denote the set of of accepting timed runs of $\mathcal{A}$:

$$runs(\mathcal{A}) := \{r| \quad \exists ts \in \mathcal{L}(\mathcal{A}) : r \in run_{\mathcal{A}}(ts) \wedge inf(r) \cap F \neq \emptyset\}$$

  Let $stateseq : runs(\mathcal{A}) \rightarrow S^\omega$ be a projection of the timed runs to only the state sequence portion of timed runs.
  Let further $Runs(\mathcal{A})$ denote the projection of $runs(\mathcal{A})$ to only the respective state sequence portions of the timed runs $r \in runs(\mathcal{A})$:

$$Runs(\mathcal{A}) := \{seq \quad | \quad \exists r \in runs(\mathcal{A}) \wedge seq = stateseq(r)\}.$$

- Using the definition of $stateseq$, the definition of $\mathcal{L}(\mathcal{A})$ (equation 6.3) can be rephrased by:

$$\mathcal{L}(\mathcal{A}) = \left\{ts \quad | \quad \exists r \in run_{\mathcal{A}}(ts) : stateseq(r) \models \texttt{GF}\left(\bigvee_{s \in F} s\right)\right\}. \tag{6.4}$$

■

According to definition 6.7, a timed observation sequence ts is not accepted by $\mathcal{A}$ - $ts \notin \mathcal{L}(\mathcal{A})$ - iff $\nexists r \in run_{\mathcal{A}}(ts) : inf(r) \cap F \neq \emptyset$. Hence ts is not accepted iff either:

- $run_{\mathcal{A}}(ts) = \emptyset$ (for some i in $ts = ...(\sigma_i, \tau_i)(\sigma_j, \tau_j)...$ no transition of $\mathcal{A}$ is enabled - this case is considered explicitly because $\mathcal{A}$ is by definition 6.6 not required to be complete)
- or $\forall r \in run_{\mathcal{A}}(ts) : inf(r) \cap F = \emptyset$ (*none* of the possible sequences of states and clock environments for processing ts visits infinitely often a fair state).

Definition 6.8 (Congruence of Clock Environments)
Let

$$\equiv_m \subseteq \Xi_C \times \Xi_C$$

be a congruence relation, with $\xi \equiv_m \xi'$ if for all clocks $c \in C$ either $\xi(c) = \xi'(c)$ or $\xi(c) \geq m \leq \xi'(c)$, where m is the largest constant appearing in all clock constraints of $\mathcal{A}$ regarding c. Note, that there either exists such a constant m for each clock $c \in C$, or c is not referred to by any clock constraint in $\mathcal{A}$.

Let $mc : C \to \mathbb{N}_0$ assign each clock $c \in C$ with the maximal constant, to which c is compared at any transition of $\mathcal{A}$. Let $mc(c){:=}0$, if c is not referred to in any clock constraint.

We then can redefine the clock environment update according to equation 6.2 of definition 6.5 by:

$$(\xi \oplus (t_\tau, t_{step}))(c){:=}\begin{cases} \xi_\tau(c) + t_\tau & \text{if } c \in C_\tau \wedge (\xi_\tau(c) < mc(c)+1) \\ \xi_\tau(c) & \text{if } c \in C_\tau \wedge (\xi_\tau(c) \geq mc(c)+1) \\ \xi_{step}(c) + t_{step} & \text{if } c \in C_{step} \wedge (\xi_{step}(c) < mc(c)+1) \\ \xi_{step}(c) & \text{if } c \in C_{step} \wedge (\xi_{step}(c) \geq mc(c)+1) \end{cases} \tag{6.5}$$

■

Lemma 6.1 (Finiteness of Clocks)

The language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ (cf. definition 6.7) is not affected if clock environment update $(\xi \oplus (t_\tau, t_{step}))(c)$ according to equation 6.5 is applied instead of the update defined in definition 6.5.

■

Proof 6.1

Trivial. Since no clock constraints in $\mathcal{A}$ compare any clock $c \in C$ to a constant larger than $mc(c)$, the interpretation of no timing constraint predicate $timing \in \Psi(C)$ in $\mathcal{A}$ can be affected.

□

By introducing $\equiv_m$ w.r.t. the largest constant m to which a particular clock is compared in $\mathcal{A}$, we avoid dealing with infinite clock domains, since it is not necessary to distinguish two clock environments which only differ in valuations of clocks irrelevant to any clock constraint.

Definition 6.9 (TSA Wellformedness)

A TSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ is said to be *well-formed* if the following requirement is satisfied[7]:

(no parallel transitions) For all states $s, s' \in S$, $s \neq s'$ there exists at most one transition from s to s':

$$\forall s, s' \in S : \sharp \left\{ t \in T \quad | \quad t = (s, -, s', -, -) \right\} \leq 1$$

■

Lemma 6.2 (TSA Wellformedness)

For a TSA with parallel transitions, there exists a well-formed TSA $\mathcal{A}'$ accepting the same language.

■

[7] We use '-' as a don't care for each part of a transition tuple, if we are not interested in the concrete appearance of the respective part. Let '-' denote an arbitrary choice of the respective part of the tuple. In particular, we will use '-' for parts of a transition tuple in order to focus on the - remaining - relevant parts of the tuple for the actual context.

Proof 6.2 W.l.o.g. let $s \in S$ be the source state of two transitions $t_1, t_2 \in T : t_1 = (s_s, enable_1, s_t, clocks_1, timing$ $t_2 = (s_s, enable_2, s_t, clocks_2, timing_2)$, with target state $s_t \in S$, s.t. :

$$(enable_1 \neq enable_2) \vee (clocks_1 \neq clocks_2) \vee (timing_1 \neq timing_2)$$

Construct $\mathcal{A}'$:=$(\mathcal{V}, S', s_0, C, T', F')$ from $\mathcal{A}$, with

1. S':=$S \cup \{s_{new}\}$ for some new state $s_{new} \notin S$,
2. F':=$F \cup \{s_{new}\}$ iff $s_t \in F$,
3. T' is build from T by modification of t_2. Transition t_1 remains unmodified in T'.
   Instead of entering s_t as with transition t_1, the modified transition

   $$t_2'\text{:=}(s_s, enable_2, s_{new}, clocks_2, timing_2)$$

   enters the newly added state s_{new}. Accordingly, for all outgoing transitions of s_t, equivalent transitions starting in s_{new} are added:

   $$\begin{aligned} T'\text{:=} \quad & (T \backslash \{t_2\}) \cup \{(s_s, enable_2, s_{new}, clocks_2, timing_2)\} \\ & \cup \{(s_{new}, enable_x, s_x, clocks_x, timing_x) \quad | \\ & \quad \exists t = (s_t, enable_x, s_x, clocks_x, timing_x) \in T\} \end{aligned}$$

The above construction preserves $\mathcal{L}(\mathcal{A})$:

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$$

Proof:

- If $\mathcal{A}$ enters s_t by taking either transition t_1 or t_2, $\mathcal{A}'$ enters either s_t or s_{new} by either taking t_1 or t_2':=$(s_s, enable_2, s_{new}, clocks_2, timing_2)$.
- For all outgoing transition of state s_t an equivalent transition can be taken also from s_{new} in $\mathcal{A}'$.
- The newly added state s_{new} in $\mathcal{A}'$ is fair iff s_t is fair. Hence acceptance is not affected. If for some timed observation sequence ts:$\exists r \in run_{\mathcal{A}}(ts)$:$s_t \in inf(r)$, then $\exists r' \in run_{\mathcal{A}'}(ts) : s_t \in inf(r') \vee s_{new} \in inf(r')$ holds.

This construction can iteratively be applied to each pair of parallel transitions occurring anywhere in $\mathcal{A}$, until no pair of parallel transitions remains in $\mathcal{A}$.

□

In the following we will only consider well-formed TSA according to lemma 6.2.

Definition 6.10 (Completeness and Determinism of TSA)
Let TSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ be given.
(Let $\Sigma_{\mathcal{V}}$ denote the possible valuations of the variables $\mathcal{V}$, and Ξ_C denote the set of all clock environments regarding C.)
For a state $s \in S$ the set of outgoing transitions is given by:

$$out(s) := \{t \mid \exists s' \in S, \exists enable \in Pred_{\mathcal{V}}, \exists timing \in \Psi(C) : t = (s, enable, s', -, timing) \in T\}$$

Then, $\mathcal{A}$ is called *deterministic at* s, if at most one outgoing transition - for all valuations and for all clock environments - can be enabled at a time:

$$\forall t_1, t_2 \in out(s), t_1 \neq t_2 : \forall \sigma \in \Sigma_{\mathcal{V}}, \forall \xi \in \Xi_C : \\ ((\sigma \models enable_{t_1}) \wedge [\![timing_{t_1}]\!](\xi)) \Rightarrow \neg((\sigma \models enable_{t_2}) \wedge [\![timing_{t_2}]\!](\xi))$$

(for all transitions the conjunctions of trigger predicate and timing constraint are mutual exclusive.)

$\mathcal{A}$ is called *complete at* s, iff always - for all valuations and for all clock environments - at least one outgoing transition is enabled, s.t.

$$\forall \sigma \in \Sigma_{\mathcal{V}}, \forall \xi \in \Xi_C : \left(\bigvee_{t \in out(s)} \sigma \models (enable_t) \wedge [\![timing_t]\!](\xi) \right)$$

$\mathcal{A}$ is called *deterministic* if $\mathcal{A}$ is deterministic at all states $s \in S$, and $\mathcal{A}$ is called *complete* if $\mathcal{A}$ is complete at all states $s \in S$.

■

Unfortunately, since determinization is already infeasible for Büchi automata due to their acceptance criterion, determinization is also infeasible for symbolic automata.

The definition of TSA and their semantics (cf. definition 6.7) explicitly deals with incompleteness: the transition relation does not guarantee, that for all states always an outgoing transition is enabled. Thus, a timed observation sequence ts can be refused by $\mathcal{A}$ for two reasons: (1) there are runs of $\mathcal{A}$ for ts, but $\forall r \in run_{\mathcal{A}}(ts) : inf(r) \cap F = \emptyset$, or (2) there are no runs of $\mathcal{A}$ for ts, i.e. $\mathcal{A}$ reaches a state s for a finite prefix of ts, from which no transition can be taken to further process ts.

Completion of a TSA requires an absorbing sink-state - i.e. an unfair state, which can never be left again once it has been entered. For each state s, at which $\mathcal{A}$ is incomplete, a transition has to be added entering the sink-state. This newly added transition then has to be taken if none of the other outgoing transition of s is enabled, i.e. if for the actual valuation σ of $\mathcal{V}$ and the actual clock environment ξ holds in s:

$$\neg \left(\bigvee_{t \in out(s)} (\sigma \models enable_t) \wedge [\![timing_t]\!](\xi) \right)$$

According to the definition of TSA, a transition is enabled if both its *enable*-predicate and its *timing*-constraint evaluate to true. In contrast, enabledness of required transition from state s to the sink-state would depend on negation of the disjunction of both the *enable*-predicate and the *timing*-constraint of *all other outgoing transitions* of s. Hence a formal completion on the level of TSA violates definition 6.6, since the completing transitions have to be triggered by expressions ranging over a combination of the variables and the specification clocks, which is not expressible transition triggers according to definition 6.6. An enhanced syntax definition - permitting the definition of more intricate transition triggers - would complicate all other definitions and conclusions of this section. Moreover, an enhanced definition would complicate the algorithms presented in the remainder of this section.

Therefore, we prefer to formalize the technical realization of TSA completion as provided by the STVE. It will become apparent on the following pages that the required information about

enabledness of transitions w.r.t. all outgoing of a particular state can efficiently be obtained in the observer encoding of TSA . The realization allows a simple implementation of *TSA with default-transitions.*

Informally, an incomplete TSA $\mathcal{A}$ can be completed using default-transitions, which enter a newly added sink-state whenever none of the transitions in the automaton's original transition relation is enabled.

Definition 6.11 (TSA with Default Transitions ($\mathbf{TSA_{def}}$))

A TSA with default transitions is a tuple $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, D, F)$, where $\mathcal{V}, S, s_0, C, T$, and F are defined as for TSA and $D \subseteq S \times S$ is a default-transition relation. Let $\mathrm{TSA}_{\mathrm{def}}$

denote the set of TSA with default transitions.

■

Definition 6.12 (Semantics of $\mathbf{TSA_{def}}$)

The semantics of $\mathrm{TSA}_{\mathrm{def}}$ is defined similarly to the semantics of TSA with the only modification of the additional default-transition relation.

Let $\mathrm{TSA}_{\mathrm{def}}$ $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, D, F)$ be given. Let

$$\textcircled{\perp}(.)(.,.) : S \times \Sigma_{\mathcal{V}} \times \Xi_C \to \mathbb{B}$$

be a function which determines for a state $s \in S$, whether any of its outgoing transitions $t \in out(S)$ is enabled for a given valuation $\sigma \in \Sigma_{\mathcal{V}}$ and a clock environment $\xi \in \Xi_C$. Let

$$\textcircled{\perp}(s)(\sigma, \xi) \quad := \quad \neg \left(\bigvee_{t \in out(s)} (\sigma \models enable_t) \wedge [\![timing_t]\!](\xi) \right).$$

A default transition $d = (s, s') \in D$ is enabled if $\textcircled{\perp}(s)(\sigma, \xi) = true$

■

Using the definition of $\mathrm{TSA}_{\mathrm{def}}$, we now can establish completion of TSA by construction of a adequate TSA with default-transitions:

Lemma 6.3 (Completion-$\mathbf{TSA_{def}}$)

Completion of a given TSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ can be realized by a TSA_{def} :

$$\mathcal{A}_C{:=}(\mathcal{V}, S', s_0', C', T', D, F'), \text{ where}$$

$S'{:=}S \cup \{s_{sink}\}$ *(an additional unfair sink-state is added to the set of states)*

$s_0'{:=}s_0$ *(the initial state is the same for both automata)*

$T'{:=}\ \ T_{\mathcal{A}} \cup \{(s_{sink}, true, s_{sink}, \emptyset, true)\}$ *(a self-loop for the sink-state is added to the transition relation, allowing the sink-state to be re-entered infinitely often once it has become the activate state)*

$D = \{(s, s_{sink}) | s \in S\}$ *(D consists of default-transitions entering the sink-state for each state except for the sink-state itself)*

$C'{:=}C$ *(the clocks are the same for both automata)*

$F'{:=}F$ *(the set of fair-states is the same for both automata)*

In $\mathcal{A}_C$ always some transition is enabled $\forall \sigma \in \Sigma_{\mathcal{V}}$ and $\forall \xi \in \Xi_C$. Hence, $\mathcal{A}_C$ does not reject a timed observation sequence due to non-enabledness of all transitions. $\mathcal{A}_C$ is complete by means of always being able to take some transition.

■

Proof 6.3

Follows immediately from the definition of $\textcircled{1}$. Either a transition $t \in T'$ is enabled for some $\sigma \in \Sigma_{\mathcal{V}}$ and $\xi \in \Xi_C$ in state s or $\textcircled{1}(s)(\sigma, \xi)$ enables the default-transition $(s, s_{sink}) \in D$. □

Lemma 6.4 (Language Preservation of Completion)

$\mathcal{A}_c$ and $\mathcal{A}$ accept the same language : $\mathcal{L}(\mathcal{A}_c) = \mathcal{L}(\mathcal{A})$

■

Proof 6.4

1. $\mathcal{L}(\mathcal{A}_c) \subseteq \mathcal{L}(\mathcal{A})$. Obvious, since $F'{:=}F$, and only default-transitions are added to $\mathcal{A}_c$ leading into an unfair state, from which no fair states are reachable. Hence $\mathcal{A}_C$does not accept more timed observation sequences than $\mathcal{A}$.

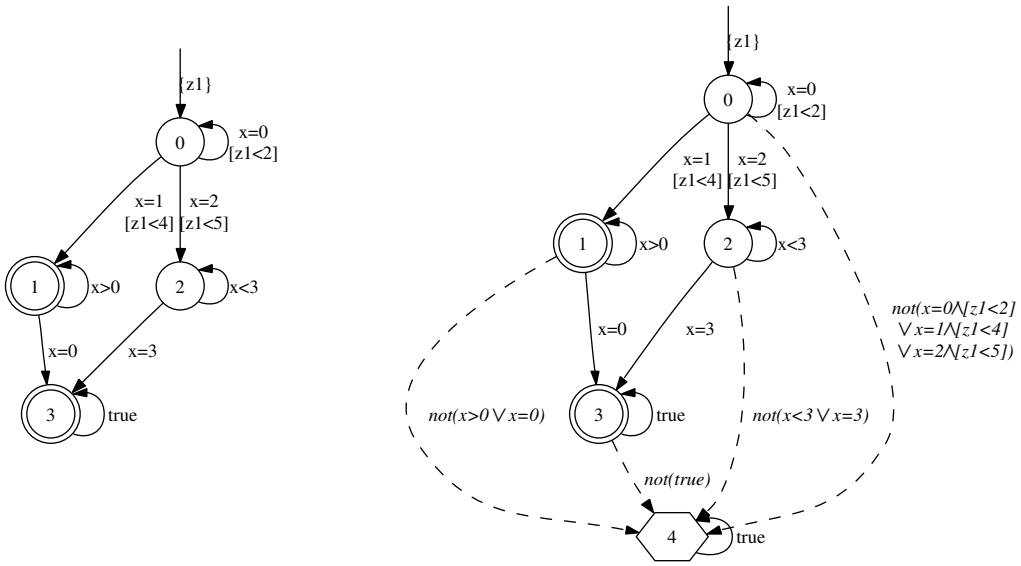
2. $\mathcal{L}(\mathcal{A}_c) \supseteq \mathcal{L}(\mathcal{A})$. The sink-state s_{sink} can only be entered by a default-transition, i.e. when no transition $t \in T$ of $\mathcal{A}_c$ is enabled. s_{sink} is entered in $\mathcal{A}_c$, iff $\mathcal{A}$ rejects a timed observation sequence ts because no transition is enabled. Hence $\mathcal{A}_c$ does not accept less timed observation sequences than $\mathcal{A}$.

□

6.3.2 Verification using Fair Synchronous TSA-Observers

For their application as specification observers in verification, TSAs are translated into SMI modules, which are then combined in parallel with the model to be verified. Part of this translation is the TSA completion according to definition 6.3:

In the code representing a TSA, default transitions are realized by introducing an additional sink-state which is entered if for the currently active state no outgoing transition is enabled. By setting a flag when taking one of the ordinary transitions of the TSA, it can be determined at the end of a step whether one of the original transitions has been taken or if the sink-state has to be entered by a default-transition.



(a) Simple TSA $\mathcal{A}$

(b) Completed automaton for $\mathcal{A}$

Figure 6.2: A simple TSA and its Completion TSA. Dashed lines denote default transitions. The labels at the default transitions are added only for illustration purposes.

As example for the completion of a TSA by default transitions consider the TSA of figure 6.2(a). Let `z1`$\in C_{step}$ be a step clock, which is initially set to `0`. Fair states of the TSA are denoted by double-circles, while single circles denote unfair states. The automaton can re-enter its initial state `0` by taking the self-loop as long as `x=0` and `z1<2` hold. If the self-loop is no longer enabled due to increase of `z1`, but the value of `x` remains `0`, then neither the self loop of state `0` nor the transitions to state `1` or state `2` are enabled.

In figure 6.2(b), an additional unfair state has been added for completion, denoted by the polygon shaped state `4`. The dashed lines denote transitions which would have to be added in order to complete the simple TSA, but are not compliant with definition 6.6, because their triggers are not separated into conjunctive *enable* and *timing* portion. By the encoding of TSA in SMI, these transitions are realized using *default*-transitions (without triggers) instead of the depicted ones.

Listing 6.1 illustrates, how the encoding of completion-TSA $\mathcal{A}_c$ for TSA $\mathcal{A}$ of figure 6.2 in principle looks like.

Recall from section 5.4 that in SMI primed and unprimed versions of variables are distinguished. In general, all variables except for inputs exist in a primed and in an unprimed version. In the description of one step of a model, the unprimed versions of variables refer to their values at the beginning of the step, whereas the primed versions represent the values computed in the actual step. At the beginning of a step the values of the primed versions of all variables is set to the value of their unprimed value[8]. Consequently, assignments are permitted only to primed versions

[8] Of course, this does not hold for auxiliary and observer variables, of which only primed versions exist. These are undefined at the beginning of a step and have to be set to a defined value before reading them anywhere in the

```
do step
begin
    DCASE
        [] ( z1'< (mc(z1)+1)) :
            --increment step-clock z1
            z1':=z1'+1
    DESAC
    stuck':=true                    – guess : none of the outgoing transitions is enabled
    NDCASE
        – for state0: consider outgoing transitions
        [] ( state0'==true) and (x==0) and (z1'<2) :
            state0':=false          – guess : state0 will be left

            state0':=true           – reenter state0 via selfloop
            stuck':=false           – revise previous guess
        [] ( state0'==true) and (x==1) and (z1'<4) :
            state0':=false          – guess : state0 will be left
            state1':=true           – enter state1 via transition
            stuck':=false           – revise guess
        [] ( state0'==true) and (x==2) and (z1'<5) :
            state0':=false          – guess : state0 will be left
            state2':=true
            stuck':=false
       – for state1: consider outgoing transitions, etc.
        [] ( state1'==true) and (x==0) :
             ...
             – encoding for state1, state2, state3 and their transitions
             – is similar to the encoding of state0
             ...
        [] ( sink_state'==true) and (true) : – if sink-state is active
            SKIP                    – do nothing
    NDESAC
    DCASE
      [] ( stuck'==true) :          – if the guess of 'no tranistion can be taken'
                                    – has not been revised by any transition
        sink_state':=true           – then enter the sink-state
    DESAC
    fair_cond':= ((state1'==true) or (state3'==true))
                                    – indicate whether the model is in a
                                    – fair state or not
end
```

Listing 6.1: Example TSA encoding

of variables, whereas read accesses are permitted to both versions of variables.

Clocks are represented by counter-variables which are reset (set to 0) when a transition is taken, whose reset specification *clocks* contains the respective clocks. These counter-variables are incremented by the program according to the perception of time, which will be discussed later in this section. Since only comparisons of a clock with constants form legal clock-constraints by definition, for each clock c a largest constant m_c to which the clock is compared throughout the entire TSA can be determined by inspection of all clock-constraints. According to lemma 6.1, each clock can be represented in a finite integer domain $0, ..., m_c+1$. In the clock updates part of the SMI representation, the clock is increased only until it exceeds its respective maximal constant $m_c = mc(c)$.

The states of TSA $\mathcal{A}_c$ are represented using a *one-hot encoding*, such that each state is represented by a boolean variable:

Let `state0`, `state1`, `state2`, `state3` as well as the added sink-state `sink_state` in listing 6.1 encode the automata states. A non-deterministic choice is used for the representation of the transition relation: the conjunction of *enable*-predicate and *timing*-constraint of a transition forms the guard of the code block implementing the respective state change when taking the transition[9]. In order to react to expiration of time limits within the same step the clock-variables are referred to by their primed version in all conditions encoding timing constraints. Note, that also the states are referred to using the primed versions of their encoding variables[10]. Later in this section, TSA will be extended with an additional activation control. This activation control will activate the initial state conditionally, which requires an instantaneous interpretation of actually being in a particular state. For the example, assume that `state0` is initialized to `true,` while all other state-variables are initially `false`. Furthermore, let `z1` - being the only clock in the example of figure 6.2 - be initialized to `0`.

Default-transitions according to definition 6.3 are implemented using a local boolean variable `stuck,` which is set to `true` at the beginning of each step. In each code block encoding the effect of taking some transition, `stuck` is set to `false` again. If at the end of a step the primed version of `stuck` remains `true` - after having considered all choices regarding the outgoing transitions of the active state, then `stuck==true` indicates that no transition could be taken in the actual step, and hence the sink-state has to be entered.

In order to adhere to a clearly defined interface, a dedicated output `fair_cond` is introduced in the SMI code, indicating for each step, whether one of the fair states is active or none of them. This additional output permits a compact specification of acceptance instead of building an expression ranging over the local state variables of the SMI representation. Moreover, referring to this step oriented '*fairness condition*', the formal treatment of acceptance can be kept independent from choosing a one hot or an integer based state encoding.

code.

[9] By definition, the outgoing transitions of a state need not to be mutual exclusive. In order to avoid inputs which later have to be introduced for the resolution of non-determinism, a tool which analyses non-deterministic choices for statically decidable determinism can be applied before applying verification. If possible, this tool splits the non-deterministic choice in a deterministic and a non-deterministic part, this way reducing non-determinism to the unavoidable minimum.

[10] Since at most one case can be chosen within a non-deterministic choice, guards referring to the primed version of a state variable disregard changes applied before to this particular variable within the same choice.

```
2  < A_SMI >:=create_SMI_program(<>)

4  /* create assignments increasing each individual clock according to advancing time */
   create_clocks_update(< A_SMI >)
6
   /* create guess, that no transition will be enabled in the actual step */
8  create_assignment(< A_SMI >,stuck', true)

10 /* here begins the encoding of original transition relation */
   < ndcase >:=create_nondeterministic_choice(< A_SMI >)
12 foreach ( [ s ∈ S ] ) {      /* forall states */
       foreach ( [ t ∈ out(s) ] ) {    /* forall outgoing transitions of state s */
14     < transition >:=create_case_in_choice(< ndcase >,
           ( primed_ref( enc_S ( [ s ] )) == true ) and
16             enc_pred( [ t.enable ] ) and primed_ref( enc_Ψ ( [ t.timing ] ) )
       /* guess: active state will be left */
18     create_assignment(< transition >, primed_ref ( enc_S ( [ s ] )), false )
       /* encode state change by original transition */
20     create_assignment(< transition >, primed_ref ( enc_S ( [ t.target ] )), true )
       /* create revision of guess, that no transition wil be enabled */
22     create_assignment(< transition >, stuck', false )
   }
24 }
   /* add sink-state with self-loop */
26 < case >:=create_case_in_choice(< ndcase >, sink_state' == true)
   < ndcase >:=create_SKIP_statement(< case >)
28
   /* default-transition: if stuck-guess has not been revised, then enter sink-state */
30 < dcase >:=create_deterministic_choice(< A_SMI >)
   < case >:=create_case_in_choice(< dcase >,stuck' == true)
32 create_assignment(< case >, sink_state', true )

34 /* indicate " being in a fair state " by fairness observer fair_cond'*/
   create_assignment(< A_SMI >, fair_cond' , generate_primed_disjunction ( [ {s ∈ F} ] ))
```

Listing 6.2: SMI-Generation Algorithm for TSA

Listing 6.2 illustrates how TSA can be encoded algorithmically in SMI . Each creation-procedure in the algorithm is called with a creation-context as its first argument. Creation contexts reflect the nesting of program control structures; for example first a non-deterministic choice is created, then a new guard is added to this choice. Afterwards, successively the assignments are added within the code block associated with the particular guard, which are to be executed if the guard evaluates to true. In the listing, creation contexts are marked by surrounding <>.

In the context of this work, we bind the interpretation of clocks $c \in C_\tau$ to the asynchronous time model of STATEMATE, i.e. while clocks $c \in C_{step}$ are updated in every step, clocks $c \in C_\tau$ are updated only if the model to be verified indicates a stable-status by issuing `SUPER_SYNC` (cf. section 5.3, definition 5.7). Note that this interpretation only tailors the formalism of TSA to the STVE. Clocks $c \in C_\tau$ could be interpreted w.r.t. arbitrary other discrete time models in other contexts.

We have extracted the generation of the clock updates in a separate procedure, which can be found in listing 6.3.

```
/* the domain of a clock is calculated, according
the definition of mc : C → ℕ0 in lemma 6.1 */

procedure create_clocks_update(context scope) {
  < τ_clocks > :=create_deterministic_choice(< scope >)
  < stable_status >:=create_case_in_choice(< τ_clocks >,"SUPER_SYNC == true")
    foreach (c ∈ Cτ) {
      /* update c (unless c = mc(c) + 1) only if the observed model
         indicates being in a stable status */
      < consider_c > :=create_deterministic_choice(< stable_status >)
      < upd_c >:=create_case_in_choice(< consider_c >,
                                       primed_ref( enc_C(c)) <  mc(c) + 1)
      create_assignment(< upd_c >, primed_ref (enc_C(c)),primed_ref (enc_C(c)) + 1)
    }
    foreach (c ∈ C_step) {
      /* update step clock c in every step, unless c = mc(c) + 1 */
      < consider_c > :=create_deterministic_choice(< scope >)
      < upd_c >:=create_case_in_choice(< consider_c >,
                                       primed_ref( enc_C(c))< mc(c) + 1)
      create_assignment(< upd_c >, primed_ref (enc_C(c)),primed_ref (enc_C(c)) + 1)
    }
}
```

Listing 6.3: procedure create_clock_updates

We assume, that the set C of clocks of $\mathcal{A}_c$ consists of only relevant clocks, i.e. all clocks $c \in C$ are referred to at some transition of $\mathcal{A}_c$.

In order to treat expressions regarding the states of $\mathcal{A}_c$ as well as timing constraints formally correct in the algorithm, we need to introduce the mappings $\mathbf{enc}_s : S \rightarrow SMIVARS$ and $\mathbf{enc}_C : C \rightarrow SMIVARS$, which respectively map the states of an automaton and its clocks to SMI variable representations (plus provision of adequate declarations). Moreover, let $\mathbf{enc}_\Psi : \Psi(C) \rightarrow SMIEXPR$ be a mapping which maps timing constraints to their encoding in SMI-expressions, by applying $\mathbf{enc}_C$ to each occurrence of a clock name in a timing constraint ψ. Similarly, let $\mathbf{enc}_{pred} : Pred_{\mathcal{V}} \rightarrow SMIEXPR$ encode predicates by corresponding SMI expressions. Furthermore, let $\mathbf{primed\_ ref} : SMIEXPR \rightarrow SMIEXPR$ be a mapping, which replaces each occurrence of a local or output variable in a SMI expression by its primed versions. Finally, for a set of states, let $\mathbf{generate\_primed\_disjunction} : 2^S \rightarrow SMIEXPR$ yield a disjunction of the primed versions of the SMI variables encoding the states in the set.

The semantics of the SMI program obtained from applying algorithm 6.2 to $\mathcal{A}$ is given by a CSTS $\Omega(\mathcal{A}) = (V_\Omega, \Theta_\Omega, \rho_\Omega, E_\Omega)$ with :

- $V_\Omega :=$ $\mathcal{V}$
  $\cup\{\mathbf{enc_S}(s) : boolean | s \in S\} \cup \{sink\_state : boolean\}$
  $\cup\{\mathbf{enc_C}(c) : integer(0..(mc(c) + 1)) | c \in C\}$
  $\cup\{stuck : boolean\} \cup \{fair\_cond : boolean\}$

- Θ_Ω:= $(\mathbf{enc_S}(s_0) = true)$
  $\wedge \left(\bigwedge_{s \in S, s \neq s_0} \mathbf{enc_S}(s) = false \right) \wedge (sink\_state = false)$
  $\wedge \left(\bigwedge_{c \in C} \mathbf{enc_C}(c) = 0 \right)$
  $\wedge (fair\_cond = false)$

- E_Ω:=$\mathcal{V} \cup \{fair\_cond\}$, and

- ρ_Ω as described by the SMI program.

Definition 6.13 (Fair Synchronous TSA-Observer)
Let $\Omega(\mathcal{A})$ denote the CSTS representing a given TSA $\mathcal{A}$, which is obtained by adding default transitions and encoding $\mathcal{A}$ according to algorithm 6.2.
Let further $\Omega(\mathcal{A}) \circ_f f$ denote the CSTS representing TSA $\mathcal{A}$ with designated output f which indicates whether $\Omega(\mathcal{A})$ is in one of its fair states.

■

In order to verify, whether all runs of a model $\mathcal{C}$ are accepted by $\mathcal{A}$, $\Omega(\mathcal{A}) \circ_f f$ is combined in parallel with $\mathcal{C}$. In the parallel composition, $\Omega(\mathcal{A}) \circ_f f$ serves as *observer* of the timed observation sequences $ts \in TComps(\mathcal{C})$ of $\mathcal{C}$.
$\Omega(\mathcal{A}) \circ_f f$ only reads the externally observable variables of the model without ever modifying them. Since $\Omega(\mathcal{A}) \circ_f f$ is complete, there always exists a transition in $\Omega(\mathcal{A}) \circ_f f$ for consecutive valuations of the observed variables. Thus, in contrast to the parallel composition of compositional synchronous transition systems in general, the parallel composition of a model with observer $\Omega(\mathcal{A}) \circ_f f$ preserves the possible runs of the model without restricting them. In order to distinguish parallel composition in general from the parallel composition with observers, we will write $||$ for the normal parallel composition in contrast to $||_\Omega$ which denotes the parallel composition with observers.

Definition 6.14 (Parallel Composition of CSTS with Synchronous Observers)
Let CSTS $\mathcal{C} = (V, \Theta, \rho, E)$ and observer $\Omega(\mathcal{A}) \circ_f f = (V_\Omega, \Theta_\Omega, \rho_\Omega, E_\Omega)$ obtained from translation of a TSA $\mathcal{A}$ be given, s.t.

- $(V \backslash E) \cap V_\Omega = \emptyset$ and $(V_\Omega \backslash E_\Omega) \cap V = \emptyset$
  (the sets of local variables of $\mathcal{C}$ and $\Omega(\mathcal{A}) \circ_f f$ are disjoint. Informally $V_\Omega \backslash E_\Omega$ consists of only clock variables and variables encoding basic states of $\mathcal{A}$, while V consists of all local model variables of $\mathcal{C}$ from which only subset E is observable outside of $\mathcal{C}$)

- $E_\Omega \subseteq (E \cup \{f\})$, moreover $E_{in_\Omega} \subseteq E$ and $E_{out_\Omega} = \{f\}$,
  ($\Omega(\mathcal{A})$ has only inputs from the externally observable variables E of $\mathcal{C}$ and $\Omega(\mathcal{A}) \circ_f f$ has only one single output f which indicates for each step whether one of the fair states of $\Omega(\mathcal{A})$ is active - in listing 6.1 f is named `fair_cond`).

The parallel composition $\mathcal{C}_{par} := \mathcal{C} ||_\Omega \Omega(\mathcal{A}) \circ_f f$ is defined by :
$\mathcal{C}_{par} = (V_{par}, \Theta_{par}, \rho_{par}, E_{par})$, with:

- V_{par}:=$V \cup V_\Omega$

- Θ_{par}:=$\Theta \wedge \Theta_\Omega$
- E_{par}:=$E \cup \{f\}$
- $\rho_{par} \subseteq \Sigma_\mathcal{V}(V_{par}) \times \Sigma_\mathcal{V}(V_{par})$ is given by

$$(\sigma, \sigma') \in \rho_{par} \text{ iff } (\sigma \downarrow_V, \sigma' \downarrow_V) \in \rho \wedge (\sigma \downarrow_{V_\Omega}, \sigma' \downarrow_{V_\Omega}) \in \rho_\Omega$$

Note, that ρ_Ω imposes no restriction on ρ_{par} w.r.t. variables of V. Since $\Omega(\mathcal{A}) \circ_f f$ is complete, $\forall \sigma, \sigma' \in \Sigma_\mathcal{V}(V_{par})$: $\exists(\sigma \downarrow_{E_{in_\Omega}}, \sigma' \downarrow_{E_{in_\Omega}}) \in \rho_\Omega$ (there always exists a transition in ρ_Ω for all possible valuations of the observed variables E_{in_Ω}).
Also Θ_Ω imposes no restriction on the initial valuation of V (Θ_Ω only determines the initial state of $\Omega(\mathcal{A})$).

Since $\Omega(\mathcal{A}) \circ_f f$ does not restrict the externally observable variables E, $||_\Omega$ is associative and commutative.

■

Figure 6.3 illustrates definition 6.14 informally.

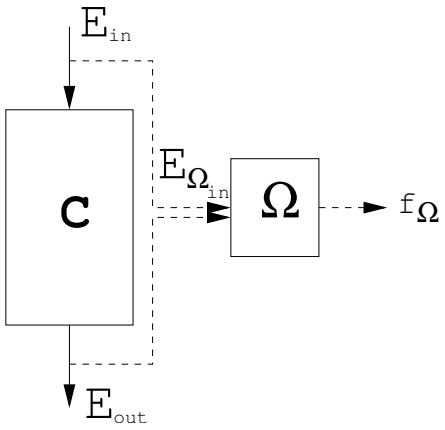


Figure 6.3: Parallel Composition with Observer

Theorem 6.1 (CTL-Verification using fair Synchronous Observers)
Given a model $\mathcal{C}$ and an observer $\Omega(\mathcal{A}) \circ_f f$ obtained from TSA $\mathcal{A}$. In general, for the parallel composition $\mathcal{C}_{par} := \mathcal{C}||_\Omega \Omega(\mathcal{A}) \circ_f f$ holds[11]*:*

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{AGAF}(f) \quad \Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{E_{\Omega_{in}}} \in \mathcal{L}(\mathcal{A}), \tag{6.6}$$

where $TComps(\mathcal{C})$ is the set of all timed observation sequences of $\mathcal{C}$.
Moreover, iff $\mathcal{A}$ is deterministic, then the following holds:

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{AGAF}(f) \quad \textit{iff} \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{E_{\Omega_{in}}} \in \mathcal{L}(\mathcal{A}) \tag{6.7}$$

[11] Let $\mathcal{K}(\mathcal{C})$ denote the Kripke structure for CSTS $\mathcal{C}$ according to definition 5.5.

In a similar way, observers can be used as assumptions: Let a deterministic TSA $\mathcal{A}_{ass}$ with output f_{ass} be given, which accepts a subset of $\Sigma_V^\omega \downarrow_{E_{in}}$ (infinite sequences of valuations of the inputs E_{in} of $\mathcal{C}$) and a deterministic TSA $\mathcal{A}$ as for equation 6.7. Then we can check whether all runs of the model, which conform to assumption $\mathcal{A}_{ass}$ are accepted by $\mathcal{A}$:

$$\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f_c||_\Omega\Omega(\mathcal{A}_{ass})\circ_f f_{ass}) \models \texttt{AGAF}(f_c) \tag{6.8}$$

with fairness constraint f_{ass} ***iff***

$$\forall ts \in TComps(\mathcal{C}) : \Big(ts \downarrow_{E_{\Omega(\mathcal{A}_{ass})\circ_f f_{ass}}} \in \mathcal{L}(\mathcal{A}_{ass})\Big) \Rightarrow \Big(ts \downarrow_{E_{\Omega(\mathcal{A})\circ_f f_c}} \in \mathcal{L}(\mathcal{A})\Big) \tag{6.9}$$

■

Proof of Theorem 6.1

1. Proof of (6.6):
   By construction of $\mathcal{C}_{par}$, $\Omega(\mathcal{A})$ only observes computations of $\mathcal{C}$ without restricting the transition relation ρ_{par} of the parallel composition $\mathcal{C}_{par}$ regarding computations of $\mathcal{C}$. If the fairness condition f is true always for all computations of $\mathcal{C}_{par}$, then obviously $\mathcal{C}$ can not engage in any timed observation sequence, which is not accepted by $\mathcal{A}$:
   Using the definitions of 4.3
   $\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f) \models \texttt{AGAF}(f)$ is equivalent to :
   $\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f) \models \neg\texttt{EF}(\neg\texttt{AF}(f))$
   $\Leftrightarrow \mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f) \models \neg\texttt{EF}(\texttt{EG}(\neg f))$
   $\Leftrightarrow \mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f) \models \neg\texttt{E}(true\,\texttt{U}\,(\texttt{EG}(\neg f))$
   $\Leftrightarrow \neg(\exists \Pi = (\sigma_0, \sigma_1, ...) : \exists k \geq 0 : (\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f)), \sigma_k \models \texttt{EG}(\neg f) \wedge \forall 0 \leq j \leq k :$
   $(\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f)), \sigma_j \models true)$

2. Hence, if $\mathcal{C}_{par}$ satisfies $\texttt{AGAF}(f)$, then there definitely exists no run of $\mathcal{C}_{par}$ violating $\mathcal{A}'s$ acceptance criterion.

   The other way round, if $\mathcal{A}$ is a non-deterministic TSA ,

   $$(\mathcal{K}(\mathcal{C}||_\Omega\Omega(\mathcal{A})\circ_f f) \not\models \texttt{AGAF}(f))$$

   does not imply that $ts \downarrow_{E_{\Omega_{in}}} \notin \mathcal{L}(\mathcal{A})$, because the definition of

   $$\mathcal{L}(\mathcal{A}) = \left\{ts \downarrow_{E_{\Omega_{in}}} \quad | \quad \exists r \in run_{\mathcal{A}}(ts \downarrow_{E_{\Omega_{in}}}) : \mathcal{A}, stateseq(r) \models \texttt{GF}\left(\bigvee_{s\in F} s\right)\right\}$$

   is only based on the existence of an accepting run for $ts \downarrow_{E_{\Omega_{in}}}$. If $\mathcal{A}$ non-deterministically accepts $ts \downarrow_{E_{\Omega_{in}}}$, this does not need to hold always for all computations of the parallel composition $\mathcal{C}_{par}$. Hence, there may exist computations of $\mathcal{C}_{par}$which do not fulfill $\texttt{AGAF}(f)$, even though $ts \downarrow_{E_{\Omega_{in}}} \in \mathcal{L}(\mathcal{A})$.

3. Proof of (6.7):
   It suffices to show that
   $$\Big(\forall ts \in TComps(\mathcal{C}) \,:\, ts \downarrow_{E_{\Omega_{in}}} \in \mathcal{L}(\mathcal{A})\Big) \Rightarrow (\mathcal{K}(\mathcal{C}||_{\Omega}\Omega(\mathcal{A})\circ_f f) \models \mathtt{AGAF}(f))\,.$$
   This follows immediately from the deterministic acceptance of TSA $\mathcal{A}$ and the fact that the possible computations of $\mathcal{C}$ are not restricted by composition with $\Omega(\mathcal{A})\circ_f f$.
4. Proof of (6.8):
   Follows from validity of (6.7) and the algorithm of fair-CTL model checking according to section 4.3.

□

Lemma 6.5 (LTL-Verification using fair Synchronous Observers)
For deterministic TSA holds :

$$\mathcal{K}(\mathcal{C}||_{\Omega}\Omega(\mathcal{A})\circ_f f) \models_{LTL} \mathtt{GF}(f) \quad \textit{iff} \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{E_{\Omega_{in}}} \in \mathcal{L}(\mathcal{A}) \tag{6.10}$$

If also $\mathcal{A}_{ass}$ is deterministic, then instead of checking 6.8 of theorem 6.1 the LTL implication :

$$\mathcal{K}(\mathcal{C}||_{\Omega}\Omega(\mathcal{A})\circ_f f_c||_{\Omega}\Omega(\mathcal{A}_{ass})\circ_f f_{ass}) \models_{LTL} \mathtt{GF}(f_{ass}) \Rightarrow \mathtt{GF}(f_c)$$

can be checked in order to establish (6.9) of theorem 6.1 .

■

Proof 6.5
Follows directly from the acceptance criterion as defined in definition 6.7 and from the determinism of $\mathcal{A}_{ass}$ and $\mathcal{A}$.

□

By theorem 6.1 TSA can be used to capture requirement specifications for verification purposes. TSA can be represented efficiently in SMI and this way be used as formal specification with a model checker in commitment place as well as for assumptions about the environment of the model. By distinguishing two different classes of specification clocks, real-time specifications in terms of simulation time can be expressed. Quantitative constraints in terms of δ-delays permit specifications of inter-activity protocols as required for the compositional verification of STATEMATE's asynchronous execution semantics.

6.3.3 Partially Ordered TSA

It will become apparent in section 6.5, that unwinding of Symbolic Timing Diagrams yields TSA of only a restricted subclass. Since Symbolic Timing Diagrams specify legal sequences of events, they induce a partial order on the states of the resulting TSA. In particular, the graphical waveforms are unwound from left to right with respect to the constraints. Consequently, the resulting TSA is a directed automaton, without cycles but containing only self-loops and branching transitions. In the remainder of this section, the partial order of TSA obtained from unwinding Symbolic Timing Diagrams will be exploited to establish an alternative acceptance criterion for a relevant subset of TSA.

Definition 6.15 (Partially Ordered TSA(POTSA))
Let TSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ be given :
Define a binary relation $\rightarrow$ on the set of states S:

$$s \rightarrow s' \text{ iff } \exists t = (s, -, s', -, -) \in T$$

Let $\rightarrow^*$ *be* the reflexive and transitive closure of $\rightarrow$.
The TSA $\mathcal{A}$ is called a *partially ordered* TSA (POTSA) , iff the relation $\rightarrow^*$is a partial order. In particular, $\mathcal{A}$ is partially ordered if $\rightarrow^*$ is anti-symmetric, i.e.:

$$(s_1 \rightarrow^* s_2) \wedge (s_2 \rightarrow^* s_1) \quad \Rightarrow \quad s_1 = s_2$$

■

Lemma 6.6 (Acyclicity of POTSA)
Let $\mathcal{A}$ be a POTSA. Then each projection of a run $\pi \in runs(\mathcal{A})$ to the states portion $\Pi \in Runs(\mathcal{A})$ has the form: $\Pi = s_0....s_k s_{k+1}....$, such that s_0 is the initial state of $\mathcal{A}$, and

$$\forall i, j : 0 \leq i \leq j : s_i \rightarrow^* s_j \tag{6.11}$$

$$\exists k \geq 0 : \forall i \geq k : s_i = s_k \tag{6.12}$$

.

■

Proof 6.6
6.11 follows from the definition of the transition relation T and anti-symmetry of $\rightarrow^*$.
6.12 follows from the finiteness of the set of states S and the partial order of $\mathcal{A}$.

□

6.12 combined with the fact that clocks can not be reset at self-loop transitions, permits some conclusion about the clock-environment portion of recorded runs. Since legal clock-constraints are restricted to comparisons of clocks with constants, of which let be $m \in \mathbb{N}$ the largest , we have :

$$(\exists k \geq 0 : \forall i \geq k : s_i = s_k) \Rightarrow \exists n \in \mathbb{N} : \forall l \geq n : \xi_l \equiv_m \xi_{l+1}.$$

6.3.4 Global Constrainedness

The TSA acceptance criterion demands for application of either CTL-model checking with fairness constraints or LTL-model checking to the parallel composition of a model with fair synchronous observers.

According to the definition of acceptance, a TSA $\mathcal{A}$ accepts a timed observation sequence ts, if some of its fair states is active infinitely often, while each of its unfair states must *eventually* be left - otherwise ts is not accepted. Hence, rejection of ts due to violation of fairness is a property of the entire run of the automaton; there is in general no determined point in time at which ts is rejected. The situation is different, if all outgoing transitions of an unfair state are explicitly constrained. If for an unfair state s none of the outgoing transitions can be enabled due to expiration of time, then rejection of ts can be decided *locally* in s.

Though in general, the proof-obligations of theorem 6.1 or lemma 6.5, respectively, have to be checked with their inherent complexity, explicitly constrained POTSA form a relevant subclass of TSA for which the acceptance criterion can be reduced to a simple invariant: If all unfair states of POTSA $\mathcal{A}$ are explicitly constrained by upper bounded clock constraints at all outgoing transitions (including their self-loops), they all must be left within determined finite time bounds, since after all clocks constraining a state have exceeded their upper bound constraints, none of the outgoing transitions can be enabled anymore. In such a case, the completion-POTSA of $\mathcal{A}$ will enter its sink-state via a default-transition. Hence, if all unfair states of a POTSA are explicitly constrained by upper bounds, the acceptance criterion can be rephrased by a *Non-Failure Acceptance* criterion:

> An *explicitly constrained completion-POTSA* accepts a timed observation sequence iff it never enters its sink-state.

As discussed above, a completion-TSA can refuse a timed observation sequence for two possible reasons:

1. the unfair sink-state is entered via default-transition because none of the ordinary transitions is enabled at a particular position in the timed observation sequence, or
2. the completion-TSA engages forever in an unfair loop.

Since the only unfair loop in the completion-TSA of an explicitly constrained POTSA is the self-loop of the sink-state, a timed observation sequence is accepted if the automaton never enters its sink-state.

Also if not all outgoing transitions of an unfair state s of POTSA $\mathcal{A}$ are explicitly constrained by upper bounds, the state may nonetheless be *transitively* constrained: If all paths to fair states in the automata starting in s are constrained by upper bounds regarding clocks which are not reset reset on the path to s, then obviously these upper bounds transitively constrain also s. If in a run entering s, the regarded clocks exceed these upper bound constraints before leaving s, then no fair state will be reached from s ever more. Hence, the run can not be extended into an accepting run of $\mathcal{A}$. Consequently, transitive constraints can be made explicit by adding them to the transitions along the affected paths. We will show below, that propagation of transitive constraints does not affect the accepted language of the automaton.

If along every possible path through a POTSA all unfair states are transitively constrained by upper bound clock constraints, then propagation of these constraints will yield an explicitly constrained POTSA, for which Non-Failure Acceptance is a suitable acceptance criterion. POTSA which are explicitly constrainable by propagation of transitive constraints will be called *globally constrained*. A formal definition of global constrainedness will be given on page 126.

In order to examine POTSA $\mathcal{A}$ for transitive upper bound constraints, it is sufficient to only inspect paths in which each self-loop transition is taken at most once.

Definition 6.16 (Progress-Closure of POTSA)

Let $\mathcal{A}$ be a given POTSA.

Let $paths(\mathcal{A})$ be the set of all legal (finite and infinite) sequences of states according to $\rightarrow$.

Let $progress\_closure(\mathcal{A})$ be the set of all *progress-paths* which are obtained from $paths(\mathcal{A})$ by eliminating all occurrences of each state s but its first and second in all $\Pi \in paths(\mathcal{A})$.

For a progress-path Π_p we write $s_i \rightarrow^*_{\Pi_p} s_j$ if s_j occurs sometimes after s_i in progress-path Π_p. ■

According to lemma 6.2, there always exists *at most one* transition from any state s_i to any successor state s_j. Lemma 6.6 established that for a POTSA $\mathcal{A}$, for each run $\Pi = s_0 s_1 ... s_m$ there exists a $k > 0$, s.t. $\forall i \geq k : s_i = s_k$. Obviously, this argument applies also to paths. Hence, Π uniquely determines a sequence of transitions. Consequently, each progress path Π_p is of finite length. Furthermore, due to finiteness of S and T, $progress\_closure(\mathcal{A})$ is a finite set. In contrast to $Runs(\mathcal{A})$, $paths(\mathcal{A})$ also captures non-accepting runs of $\mathcal{A}$, and hence $progress\_closure(\mathcal{A})$ covers $\mathcal{A}$ entirely, regardless of finally reaching a fair state or not.

In order to examine POTSA for transitive constraints, we have to apply a normalization to the timing constraints. Since timing constraints (definition 6.4) are build from clock constraints using boolean connectives, they may contain negated clock constraints or disjunctions of clock constraints and also redundant, i.e. logically superfluous clock constraints. For examining transitive constraints affecting a particular state only unique clock constraints are of interest, i.e. non-negated clock constraints, which must necessarily hold for all accepting runs entering this state.

Definition 6.17 (Unique Clock Terms in Timing-Constraints)

Given set of timing-constraints $\Psi(C)$, w.r.t. the set C of clocks belonging to TSA $\mathcal{A}$.

We call a clock constraint $\gamma \in \Gamma(C)$ *absolute* w.r.t. a timing constraint $\psi \in \Psi(C)$, iff $\forall \xi \in \Xi_C : [\![\psi]\!](\xi) \Rightarrow [\![\gamma]\!](\xi)$. (if satisfaction of γ is a necessary condition for satisfaction of ψ)

Let

$$uniqct : \Psi(C) \rightarrow 2^{\Gamma(C)}$$

be a mapping which determines the set of absolute clock constraints of a timing constraint in a unique, non-negated form.

- $uniqct(true) := \emptyset, uniqct(\neg true) := \emptyset$
- $uniqct(\gamma) := \{\gamma\}$,
- $uniqct(\,(\gamma)\,) := uniqct(\gamma)$
- $uniqct(\neg\gamma) := \begin{cases} uniqct((c < n) \vee (c > n)) & \text{if } \gamma = (c = n) \\ \{(c < n)\} & \text{if } \gamma = (c \geq n) \\ \{(c > n)\} & \text{if } \gamma = (c \leq n) \\ \{(c \leq n)\} & \text{if } \gamma = (c > n) \\ \{(c \geq n)\} & \text{if } \gamma = (c < n) \end{cases}$
- $uniqct(\neg\psi) := \begin{cases} uniqct(\neg\psi_1 \vee \neg\psi_2) & \text{if } \psi = (\psi_1 \wedge \psi_2) \\ uniqct(\neg\psi_1 \wedge \neg\psi_2) & \text{if } \psi = (\psi_1 \vee \psi_2) \\ uniqct(\psi_1) & \text{if } \psi = \neg\neg\psi_1 \\ uniqct(\neg\gamma) & \text{if } \psi = \gamma \end{cases}$
- $uniqct(\psi_1 \vee \psi_2) :=$

$$\begin{array}{rcl} \{\gamma_1 \in uniqct(\psi_1) & | & \exists \gamma_2 \in uniqct(\psi_2) : \gamma_2 \Rightarrow \gamma_1\} \\ \cup \{\gamma_2 \in uniqct(\psi_2) & | & \exists \gamma_1 \in uniqct(\psi_1) : \gamma_1 \Rightarrow \gamma_2\} \end{array}$$

- $uniqct(\psi_1 \wedge \psi_2) := uniqct(\psi_1) \cup uniqct(\psi_2)$

Let furthermore

$$reduce : 2^{\Gamma(C)} \rightarrow 2^{\Gamma(C)}$$

be a reduction of a set of clock constraints s.t.

- $reduce(\emptyset) := \emptyset$
- $reduce(\{\gamma\}) := \{\gamma\}$
- $reduce(\{\gamma_1, \gamma_2\}) := \begin{cases} \{\gamma_1\} & \text{if } \gamma_1 \Rightarrow \gamma_2 \\ \{\gamma_2\} & \text{if } \gamma_2 \Rightarrow \gamma_1 \\ \{\gamma_1, \gamma_2\} & \text{otherwise}\} \end{cases}$
- for a subset $G \subseteq \Gamma(C)$, which contains at least three clock constraints, let $reduce(G)$ be the recursive application of $reduce$ to each pair of clock constraints $\gamma_1, \gamma_2 \in G$ and hence yield the set : $reduce(G) := \{\gamma \quad | \quad \forall \gamma_1, \gamma_2 \in G : \gamma \in reduce(\{\gamma_1, \gamma_2\})\}$.

Let $ubc : \Gamma(C) \times C \times \mathbb{N}_0 \rightarrow \mathbb{B}$, be a function which decides whether a given clock constraint γ is an upper bound constraint of c w.r.t. $m \in \mathbb{N}_0$:

$$ubc(\gamma, c, m) := true \text{ iff } \gamma = \begin{cases} c < m + 1 \\ c = m \\ c \leq m \end{cases}$$

■

The following lemma establishes a normal form for POTSA, which guarantees that clocks are used in a unique way, i.e. they are neither "reused" along a path, nor referred to with different upper bounds. This normal form will be required for the formal definition of global constrainedness.

In order to establish this normal form, we need a few more definitions:

- let $constraints : C \rightarrow 2^{\Gamma(C)}$ denote for each clock $c \in C$ the set of all (non-negated) constraints referring to clock c occurring in $\mathcal{A}$, i.e.

$$constraints(c) := \left\{ \begin{array}{l} \gamma \,|\, \exists t = (s, enable, s', clocks, timing) \in T : \\ \gamma \in reduce(uniqct(timing)) \wedge \gamma = c \sim m \end{array} \right\},$$

  where $m \in \mathbb{N}_0$ and $\sim$ is $\leq, <, >\geq$ or $=$.

- let $upper\_bounds(c)$ be the set of constants occurring as upper bounds of c in $\mathcal{A}$, i.e.

$$upper\_bounds(c) := \{m \in \mathbb{N}_0 \,|\, \exists \gamma \in constraints(c) : ubc(\gamma, c, m) = true\}$$

Lemma 6.7 (Normalform of POTSA - Unique Clocks)
For a partially ordered POTSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T_{\mathcal{A}}, F)$, there exists a partially ordered POTSA $\mathcal{A}' = (\mathcal{V}, S', s_0, C', T_{\mathcal{A}'}, F')$, which is language-equivalent to $\mathcal{A}$, i.e. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, and along every path Π through $\mathcal{A}'$ holds :

1. *each clock $c \in C$' is reset exactly once, and*
2. *each clock $c \in C'$ has either no upper bound, or the upper bound is uniform in all timing constraints referring to c, i.e.*

$$\forall c \in C' : \sharp\{\gamma \in constraints(c) \quad | \quad \exists m \in \mathbb{N}_0 : ubc(\gamma, c, m) = true\} \leq 1$$

Thus,

$$upper\_bounds(c) \quad = \quad \begin{cases} \{m\} & if\ \exists^1 \gamma \in constraints(c)\ :\ \exists m \in \mathbb{N}_0 : ubc(\gamma, c, m) = true \\ \emptyset & otherwise \end{cases}$$

■

Proof 6.7

$\mathcal{A}'$ can be constructively obtained from $\mathcal{A}$ through replacing non unique clock names by fresh clock names. By definition, $\mathcal{A}$ has no cycles except for self-loops (lemma 6.6). By lemma 6.2, clocks are not reset at self-loops.

Initially let $\mathcal{A}'$:=$\mathcal{A}$. For each progress-path $P = s_0...s_{i1}s_{i2}...s_{j1}s_{j2}...s_{k1}s_{k2}...$ through $\mathcal{A}$, we have to consider two possible cases:

1. $\exists c_1 \in C$, s.t. c_1 is

   - referred to in the timing constraint $timing_{t_1}$ of transition $t_1 = (s_{i1}, enable_{t_1}, s_{i2}, clocks_{t_1}, timing_{t_1})$,
   - reset in $clocks_{t_2}$ at transition $t_2 = (s_{j1}, enable_{t_2}, s_{j2}, clocks_{t_2}, timing_{t_2})$, and
   - again referred to in the timing constraint $timing_{t_3}$ of transition $t_3 = (s_{k1}, enable_{t_3}, s_{k2}, clocks_{t_3}, timing_{t_3})$.

   Then the reference to c_1 at transition t_3 is independent from the reference to c_1 at t_1. Uniqueness of clocks can be established by introducing a fresh clock c_{new} and replacing c_1 in the set of clocks to be reset at transition t_2 as well as in the timing constraint of t_3:

   a) add c_{new} to C',
   b) eliminate t_2 from $T_{\mathcal{A}'}$
   c) add $t_{2'}$ to $T_{\mathcal{A}'}$, s.t. $T_{\mathcal{A}'} \ni t'_2 = (s_{j1}, enable_{t_2}, s_{j2}, clocks_{t_2} \backslash \{c_1\} \cup \{c_{new}\}, timing_{t_2})$
   d) eliminate t_3 from $T_{\mathcal{A}'}$
   e) add $t_{3'}$ to $T_{\mathcal{A}'}$, s.t. $T_{\mathcal{A}'} \ni t'_3 = (s_{k1}, enable_{t_3}, s_{k2}, clocks_{t_3}, timing_{t_3}[c_{new}/c_1])$

2. $\exists c_1 \in C$, s.t. c_1 is

   - reset at transition $t_1 = (s_{i1}, enable_{t_1}, s_{i2}, clocks_{t_1}, timing_{t_1})$,
   - first referenced in a timing constraint at transition $t_2 = (s_{j1}, enable_{t_2}, s_{j2}, clocks_{t_2}, timing_{t_2})$, and

- again referenced in a timing constraint of $t_3 = (s_{k1}, enable_{t_3}, s_{k2}, clocks_{t_3}, timing_{t_3})$.

Then the reference to c_1 at transition t_3 can be replaced by a reference to a fresh clock which is added to the reset list of t_1:

a) add c_{new} to C'
b) eliminate t_1 from $T_{\mathcal{A}'}$
c) add $t_{1'}$ to $T_{\mathcal{A}'}$, s.t. $T_{\mathcal{A}'} \ni t'_1 = (s_{j1}, enable_{t_2}, s_{j2}, clocks_{t_2} \cup \{c_{new}\}, timing_{t_2})$
d) $T_{\mathcal{A}'} \ni t'_2 = t_2 \in T_{\mathcal{A}}$
e) eliminate t_3 from $T_{\mathcal{A}'}$
f) add $t_{3'}$ to $T_{\mathcal{A}'}$, s.t. $T_{\mathcal{A}'} \ni t'_3 = (s_{k1}, enable_{t_3}, s_{k2}, clocks_{t_3}, timing_{t_3}[c_{new}/c_1])$

Since $\mathcal{A}$ is partially ordered, there exist only finitely many such progress-paths through $\mathcal{A}$.

□

It will become apparent in section 6.5 that unwinding of Symbolic Timing Diagrams yields POTSA adhering to this normal form.

Deciding Global Constrainedness

If for POTSA $\mathcal{A}$ all paths from state s to fair states are constrained by upper bounds regarding clocks which were reset before entering s, then these upper bounds transitively constrain also s.

Informally, we call state s *globally constrained*, if either s is itself a fair state or if s has to be left always within a finite time interval - determined by explicit or transitive upper bound constraints. If $\mathcal{A}$ is in normal form, then no clock is referred to more than (at most) once in upper bound clock constraints along any path of $\mathcal{A}$. Hence, for deciding global constrainedness of s, only *running* clocks have to be regarded, i.e. clocks which were reset but have not been referred to in a clock constraint before entering s.

Definition 6.18 (Global Constrainedness of POTSA)
Let $\mathcal{A} = (\mathcal{V}, S, s_0, C, T_A, F')$ be a POTSA in normal form. For a state $s \in S$ and a progress-path $\Pi_p \in progress\_closure(\mathcal{A})$, let

1. $running\_clocks(s_0, s)$ be the clocks $c_i \in C$ which have not been referred to in an *upper bound clock constraint* along progress-path Π_p between s_0 and s, and let
2. $ub\_regarded\_clocks(s, s_2)$ be the clocks $c_j \in C$ which are referred to in *upper bound clock constraints* along progress-path Π_p between s and s_2.
3. $final(\Pi_p)$ denote the right most state occurring in Π_p

A state $s \in S$ of $\mathcal{A}$ is *globally constrained* , if :

- $s \in F$, or

- $s \notin F \quad \wedge \quad \forall \Pi_p \in progress\_closure(\mathcal{A})$ in which s occurs :

$$\forall s_j \in F : (s \rightarrow^*_{\Pi_p} s_j \Rightarrow (running\_clocks(s_0, s) \cap ub\_regarded\_clocks(s, s_j) \neq \emptyset))$$
$$\wedge running\_clocks(s_0, s) \cap ub\_regarded\_clocks(s, final(\Pi_p)) \neq \emptyset$$

$\mathcal{A}$ is globally constrained, if every state $s \in S$ is globally constrained.

■

In the sequel we will present an clock-algorithm, which decides global constrainedness for a POTSA by constructively propagating the discovered upper bound constraints of each globally constrained state to its incoming transitions.

Clock-Algorithm

In order to present the algorithm, we need a few more definitions (recall the definition of $out(s)$ from definition 6.10) :

Let $in : S \rightarrow T$ denote the incoming transitions of state s:

$$in(s) := \{t \in T \quad | \quad \exists s' \in S : t = (s', -, s, -, -)\}$$

Let $upclocks : S \rightarrow 2^C$ denote the set of clocks, for which an upper bound constraint exists at all outgoing transitions of a state s,

$$upclocks(s) := \left\{ \begin{array}{l} c \in C \quad | \quad \forall t = (s, -, s', -, timing) \in out(s), s \neq s' \text{ if } s \notin F : \\ \exists \gamma \in reduce(uniqct(timing)), \exists m \in \mathbb{N}_0 : ubc(\gamma, c, m) = true \end{array} \right\}$$

In the definition of *upclocks*, fair states and unfair states are treated differently w.r.t. self-loops. Since $out(s)$ contains also the self-loop of state s, $upclocks(s)$ explicitly does not consider self-loops of unfair states, because the automaton can only accept a timed observation sequence if every unfair state is eventually left by taking a (series of) transition(s) entering a fair state. Hence we are only interested in satisfiability of upper bound constraints at transitions leaving unfair states. For fair states, the situation is different. Since POTSA only accept a timed observation sequence iff some fair state is re-entered forever by its self-loop, a fair state is constrained by an upper bound w.r.t. clock c only if also the self-loop is restricted by this upper bound.

Let $upbound : C \times S \rightarrow \mathbb{N}_0$ be a partial function, which is defined only for $c \in upclocks(s), s \in S$:

$$upbound(c, s) := max \left\{ m \in \mathbb{N}_0 \quad | \quad \exists \gamma \in \left(\bigcup_{t \in out(s)} uniqct(t.timing) \right) : ubc(\gamma, c, m) = true \right\}$$

In the definition of *upbound* the maximum of the upper-bounds is chosen, because the algorithm will strengthen the upper bounds of step-clocks before propagating them to the incoming transitions of a constrained state. Such *backward-strengthening* makes use of the fact, that if a state has to be left before the value of some step clock exceeds an upper bound $n \in \mathbb{N}$, then this state must have been entered before the value of this step clock has exceeded $n - 1$.

Therefore, the upper bound constraints of the outgoing transitions w.r.t. a step-clock $c \in C_{step}$ may differ. Backward-strengthening can only be applied for step clocks: For clocks $c \in C_\tau$no such decrementation can be applied, since several steps of the automaton can take place between two

consecutive updates of the clock. Hence, for clocks $c \in C_\tau$ the upper bound has to be propagated without decrementation.

By taking the maximum of differing upper bound constants instead of requiring uniformness, we can refrain from requiring uniform upper bound constraints in all timing constraints referring to a particular clock. The price to be paid is possibly weakening the strength of the propagated bound, which is irrelevant w.r.t. overall acceptance criterion of the automaton[12].

Taking the maximum of the upper bound constraints w.r.t. a particular clock for propagation is compensated by application of *csimplify* to the resulting timing constraint, since *csimplify* eliminates logically superfluous terms in a timing constraint. For example, $csimplify((c \leq m) \wedge (c \leq m+1)) = (c \leq m)$ for any clock $c \in C$ and $m \in \mathbb{N}_0$:

Definition 6.19 (Simplification of Timing Constraints)
Let C be a given set of clocks. Let

$$csimplify : \Psi(C) \rightarrow \Psi(C)$$

be a simplification and normalization of timing constraints, s.t. for $\psi \in \Psi(C)$, $csimplify(\psi)$ is negation-free and minimal w.r.t. conjunctive terms. In particular, let

- $csimplify(\gamma) := \gamma$
- $csimplify(\,(\gamma)\,) := \gamma$
- $csimplify(\neg\gamma) := \begin{cases} ((c < n) \vee (c > n)) & \gamma = (c = n) \\ (c < n) & \gamma = (c \geq n) \\ (c > n) & \gamma = (c \leq n) \\ (c \leq n) & \gamma = (c > n) \\ (c \geq n) & \gamma = (c < n) \end{cases}$
- $csimplify(\psi) := \begin{cases} \bigwedge\limits_{\gamma \in reduce(uniqct(\psi))} \gamma & \text{iff } \exists i \geq 2 : \psi = \bigwedge\limits_i \gamma_i \\ \psi & \text{otherwise} \end{cases}$
- $csimplify\left(\neg\left(\bigvee\limits_{i\geq 2} \psi_i\right)\right) := \bigwedge\limits_i \neg\psi_i$
- $csimplify\left(\neg\left(\bigwedge\limits_{i\geq 2} \psi_i\right)\right) := \left(\bigvee\limits_i \neg\psi_i\right)$
- $csimplify\left(\bigvee\limits_{i\geq 2} \psi_i\right) := \begin{cases} \psi_j & \text{iff } \exists j : \psi_j \Rightarrow \left(\bigvee\limits_{j\neq i} \psi_i\right) \\ \bigvee\limits_i csimplify(\psi_i) & \text{otherwise} \end{cases}$

[12] The weaker the propagated upper bounds are, the later the automaton detects a violation of a clock constraint by a timed observation sequence.

- $csimplify\left(\bigwedge_{i\geq 2}\psi_i\right) := \bigwedge_i csimplify(\psi_i)$

■

Because POTSA contain no loops except for self-loops, there exists a natural measure of the distance of a state from the initial state. Due to the partial order of the states, for two states $s \neq s'$ with $s \rightarrow^* s'$, the outgoing transitions of s can obviously not constrain s'. The other way round, upper bound constraints of $out(s')$ may transitively constrain also s. Hence, in order to consider each state of a POTSA only once, the clock-algorithm starts with the most distant states according to $\rightarrow^*$ and successively constrains the less distant states if this is feasible. Since different paths with different lengths may lead to the same state in the automaton, the distance of a state from s_0 has to be computed as maximum of the lengths of the possible paths.

Let `distance` be an array whose indices are the states $s \in S$. Algorithm 6.4 calculates - using a breadth-first search - for each state $s \in S$ the minimal length of the maximal path $\Pi : s_0...s$ from the initial state s_0, s.t. no self-loops are taken along Π.

```
foreach ( s ∈ S ) distance := 0

depth := 0
mark := {s_0}
nextmark := ∅
while ( mark ≠ ∅ ) {
    foreach ( s ∈ mark ) {
      if ( depth > distance(s) ) distance(s) := depth
      foreach ( t ∈ out(s) ) {
          if ( t.source ≠ t.target )
              nextmark := nextmark ∪ {t.target}   }   }
    depth := depth + 1
    mark := nextmark
    nextmark := ∅
}
```

Listing 6.4: Calculating the Distances of States to the Initial State

The clock-algorithm starts with (one of) the most distant states and determines whether this state is constrained by upper bounds at all outgoing transitions. If this is the case, the upper bound constraint is added to the timing constraint of all incoming transitions except for the transition at which the respective clocks are reset (to which the upper bound refers). If the self-loop of a fair state s has already been restricted, this is taken into account by $upclocks(s)$. Otherwise s must not be restricted, because this would essentially modify the acceptance of the automaton. Since the self-loops of unfair states are explicitly excluded when determining $upclocks(s)$, they have to be restricted if there exist upper bound constraints for all transitions to successor states. For clocks $c \in C_{step}$ referring to steps, the upper bound constraints added to the incoming transitions can safely be strengthened. The upper bound constraint of an transition can only be met, if the source state was entered within an upper bound of 1 step less or the respective clock was reset at the transition entering the state.

Consequently, if for some state an upper bound constraint w.r.t. a step clock has to be propagated which already requires the absolute minimum, the algorithm reveals that an entire path of the automaton is logically cut off due to an unsatisfiable constraint. This is not an error w.r.t. the semantics of TSA, but in practice it is an indication that the constraint might have been chosen too small. Hence, the algorithm indicates this case, by setting $satisfiable\_constraints$ to $false$.

For clocks $c \in C_\tau$ referring to the time portion of the timed observation sequence, in general no such knowledge about the relationship between clock environments for consecutive transitions can be applied. Since several observations can have the same time stamp, several steps of the automaton can take place without incrementing the respective clocks. Hence, upper bound clock constraints referring to clocks $c \in C_\tau$ have to be propagated as such, without decrease.

The considered state is added to the set of already regarded states after propagation of all relevant upper bound constraints to its incoming transitions. The algorithm terminates after either all states have been regarded, or an unfair state has been detected which is not constrained by any upper bound constraint. The algorithm iterates its outermost while-loop as many times as states have to be regarded. Hence, $\sharp S$ iterations have to be performed when the algorithm is applied to a globally constrained POTSA, while un-constrainedness can be detected earlier.

```
2  Regarded := ∅
   failure := false
4  satisfiable_constraints := true

6  while ( (Regarded ≠ S) ∧ (failure = false) ) {
       choose s ∈ S\Regarded with maximal distance from s_0
8      /* if an unconstrained unfair state is detected, the
                automaton is not constrainable */
10     if ( s ∉ F ∧ upclocks(s) = ∅ ) failure := true
       foreach ( t ∈ in(s) ) {        /* t=(source,enable,target,clocks,timing) */
12         /* a selfloop of a fair state must not be restricted */
           if ( ¬((s ∈ F) ∧ (t.source = t.target)) )
14             foreach ( c ∈ upclocks(s) ) {
                   if ( c ∉ t.clocks )
16                     if ( c ∈ C_step )
                           if ( upbound(c, s) = 0 ) {
18                             satisfiable_constraints := false
                               t.timing := csimplify(t.timing ∧ (c ≤ 0))
20                         } else
                               t.timing := csimplify(t.timing ∧ (c ≤ upbound(c, s) − 1))
22                     if ( c ∈ C_τ )
                           t.timing := csimplify(t.timing ∧ (c ≤ upbound(c, s)))
24             }
       }
26     Regarded := Regarded ∪ {s}
   }
```

Listing 6.5: Clock Algorithm for POTSA

Upon successful termination of the algorithm we have $Regarded = S \wedge failure = false$. It depends on the use case of the algorithm whether the side condition $satisfiable\_constraints = false$ is interpreted as an error or just as a hint that some upper bound w.r.t. a step clock has been chosen too small.

Notice that algorithm 6.5 is not applicable to completion-TSA, since the sink-state of a completion-TSA is an unconstrained unfair state from which no fair state can ever be reached.

Lemma 6.8 (Language Preservation of Explicit Constraining)
Let $\mathcal{A}$ be a globally constrained POTSA, and let $\mathcal{A}_\sharp$ be the explicitly constrained POTSA which is obtained from $\mathcal{A}$ by successfully applying the clock algorithm algorithm. Then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\sharp)$ holds. ■

Proof 6.8

1. Self-loops of fair states are explicitly not targeted by upper bound propagation. Hence, once a fair state is entered, its self-loop can be taken without being additionally constrained in $\mathcal{A}_\sharp$ compared to $\mathcal{A}$.

2. The *enable* predicates of all transitions are not affected by the algorithm

3. *clocks* resets are not affected by the algorithm

4. Regarding the modified timing constraints, it suffices to show that propagation of clocks does not affect the acceptance of the automaton.
   The algorithm starts with the most distant states and works backward until all states have been considered. Therefore, the iterations of the algorithm can be characterized by a finite series of automata, each of which is the result of applying one iteration of the algorithm, where $\mathcal{A}_0{:=}\mathcal{A}$.
   Consider a transition $t = (s, enable, s', clocks, timing)$ of $\mathcal{A}_i$ which has been modified into $t' = (s, enable, s', clocks, timing_\sharp)$ of $\mathcal{A}_{i+1}$ by the algorithm:
   s' is globally constrained, because the algorithm in each iteration only modifies the incoming transitions of already explicitly constraint states.
   Regarding the set of clocks $upclocks(s') \subseteq C$ constraining state s' with upper bounds at all outgoing transitions, $timing_\sharp$ has been obtained by conjunction of *timing* with

$$\gamma = \begin{cases} (c \leq upbound(c, s') - 1) & \text{if } c \in C_{step} \\ (c \leq upbound(c, s') & \text{if } c \in C_\tau \end{cases}, \text{ where}$$

   $upbound(c, s')$ is the maximal upper bound w.r.t. c at all outgoing transitions of s'. Hence, $timing_\sharp$ disables $t_\sharp$ to be taken in $\mathcal{A}_{i+1}$ only for these clock environments, for which entering s' can in no case be continued with an accepting run of $\mathcal{A}_i$.
   If t is enabled in $\mathcal{A}_i$ for some clock environment ξ with $[\![c > mc(c)]\!](\xi)$, because *timing* does not contain an upper bound constraint regarding $c \in upclocks(s')$, then $\mathcal{A}_i$ will refuse the currently processed timed observation sequence ts in s' because no outgoing transition of s' is enabled in $\mathcal{A}_i$. For the same clock interpretation ξ, $\mathcal{A}_{i+1}$ will refuse ts already in s, because $t_\sharp$ is not enabled.
   The other way round, if $[\![c \leq m]\!](\xi)$ for clock environment ξ when taking t in $\mathcal{A}_i$, and hence

at least one of the outgoing transitions of s'in $\mathcal{A}_i$ can be enabled, then also $timing_\sharp$ will allow $\mathcal{A}_{i+1}$ to enter s'.

□

6.3.5 Non-Failure Acceptance

Definition 6.20 (Language Accepted by Non-Failure Acceptance)
Let $\mathcal{A}$ be a globally constrained POTSA. Let $\mathcal{A}_c$ be its completion-TSA according to definition 6.3 with dedicated sink-state $sink\_state$. Since, by construction, the sink-state is only entered in $\mathcal{A}_c$ when $\mathcal{A}$ can take none of its transitions, we also refer to this state as *failure-state*.
The set of timed observation sequences for which $\mathcal{A}_c$ does not enter its sink-state is given by:

$$\mathcal{L}_{non-failure}(\mathcal{A}_c) := \{ts = (\pi, \tau) \,|\, inf(run_{\mathcal{A}_\sharp}(ts)) \cap \{sink\_state\} = \emptyset\}$$

And since $sink\_state$ can never be left once it has been entered, we can rephrase this by:

$$\mathcal{L}_{non-failure}(\mathcal{A}_c) := \{ts \quad | \quad \exists r \in run_{\mathcal{A}}(ts) : \mathcal{A}_c, stateseq(r) \models \mathtt{G}\,(\neg sink\_state)\}$$

We call $\mathcal{L}_{non-failure}(\mathcal{A}_c)$ the *non-failure accepted language* of $\mathcal{A}_c$.

■

Theorem 6.2 (Equivalence of Non-Failure and Büchi Acceptance)
For the explicitly constrained completion-TSA $\mathcal{A}_c$ of an globally constrained POTSA $\mathcal{A}$ holds :

$$\mathcal{L}_{non-failure}(\mathcal{A}_c) = \mathcal{L}(\mathcal{A}_c)$$

■

Proof of Theorem 6.2

1. $\mathcal{L}(\mathcal{A}_c) \subseteq \mathcal{L}_{non-failure}(\mathcal{A}_c)$:
   Suppose there is a timed observation sequence $ts \in \mathcal{L}(\mathcal{A}_c)$, for which $ts \notin \mathcal{L}_{non-failure}(\mathcal{A}_c)$ holds. This implies, that $\mathcal{A}_c$enters its $sink\_state$ for some position in ts instead of taking an enabled transition. By construction of $\mathcal{A}_c$ the $sink\_state$ is only entered when none of the other transitions is enabled. Hence, $\mathcal{L}(\mathcal{A}_c) \subseteq \mathcal{L}_{non-failure}(\mathcal{A}_c)$

2. $\mathcal{L}_{non-failure}(\mathcal{A}_c) \subseteq \mathcal{L}(\mathcal{A}_c)$:
   Suppose there exists a timed observation sequence $ts \in \mathcal{L}_{non-failure}(\mathcal{A}_c)$, for which $ts \notin \mathcal{L}(\mathcal{A}_c)$ holds. This implies that $\mathcal{A}_c$ for some position in ts does not enter its $sink\_state$, although no transition is enabled. By construction of $\mathcal{A}_c$, this is impossible.

□

By theorem 6.2, for globally constrained POTSA $\mathcal{A}$ only an invariant has to be checked in order to verify whether all runs of a model are accepted by $\mathcal{A}$.

Definition 6.21 (Observer with Non-Failure Acceptance)
Let $\mathcal{A}$ be an explicitly constrained POTSA. Let $\Omega(\mathcal{A})$ be the observer obtained from algorithm 6.2, to which a dedicated output is added, which indicates in every step that $sink\_state$ is not the currently active state. In particular, let output $o{:=}(\neg sink\_state)$.

Let $\Omega(\mathcal{A}_\sharp) \triangleright o$ denote this modified observer module (representing $\mathcal{A}$ with Non-Failure acceptance observer o).

■

Lemma 6.9 (Verification using Non-Failure Acceptance)
For an explicitly constrained POTSA $\mathcal{A}$ and a model $\mathcal{C}$, instead of verifying, whether

$$\mathcal{C}||_\Omega \Omega(\mathcal{A}) \circ_f f \overset{?}{\models} \mathtt{GF}(f)$$

holds, it can be checked :

$$\mathcal{K}(\mathcal{C}||_\Omega \Omega(\mathcal{A}_\sharp) \triangleright o) \overset{?}{\models} \mathtt{G}(o) \tag{6.13}$$

■

Proof 6.9
Follows immediately from theorem 6.2.

□

Proof obligation 6.13 can be checked using *invariance checking* with worst case complexity of a reachability computation for $\mathcal{C}||_\Omega \Omega(\mathcal{A}_\sharp) \triangleright o$ (cf. section 4.7).

6.3.6 POTSA with Activation Control (POTSA$_{\mathsf{AC}}$)

POTSA as introduced so far always consider entire timed observation sequences. They start processing a timed observation sequence with the initial position. Due to the partial order of their states POTSA can process repetitions of subsequences of observations in a timed observation sequence in only a very restricted way. For example a language $(abc)^\omega$ can not be recognized by a partially ordered automaton - disregarding the time sequence for the moment, since the partial order of the states does not permit cycles other than self-loops. Hence, only finitely many repetitions of sequence *abc* are recognizable by a partially ordered automaton through providing appropriately many states and transitions.

Even such iterative repetitions of sub-sequences of observations are of particular interest in the verification of embedded systems. Basically, all protocols consist of series of actions in reaction to a series of triggering events. Hence, often a triggered activation of observers is desired, such that processing of a timed observation sequence is started with the first position for which a particular activation condition holds instead always beginning with the initial observation. Therefore, a mechanism has to be provided which permits iterated activation of POTSA.

In general, the acceptance criterion of TSA and also of POTSA does not provide a notion of finite acceptance, and hence a POTSA $\mathcal{A}$ is started only once without permitting re-activation, since a finite acceptance can not take place. Contrariwise, there exists definitely no way to reject any continuation of a timed observation sequence if once a fair state has been entered from which no other state can be reached and whose only outgoing transition is an always enabled self-loop.

Lemma 6.6 has shown that for POTSA $\mathcal{A}$ holds:
$\forall ts \in \mathcal{L}(\mathcal{A}) : \exists k \geq 0 : \forall i \geq k : \exists m \in \mathbb{N}_0 : (s_0, \xi_0)...(s_i, \xi_i)(s_{i+1}, \xi_{i+1})... = run_{\mathcal{A}}(ts) :$

$$s_i = s_{i+1} : \xi \equiv_m \xi_{i+1}$$

If s_k has been reached for prefix $\overrightarrow{p}:=(\sigma_0, \tau_0)...(\sigma_k, \tau_k)$ of a timed observation sequence, then this prefix can not be continued by any suffix $\overrightarrow{q}:=(\sigma_{k+1}, \tau_{k+1})...$, such that $ts = \overrightarrow{p}\,\overrightarrow{q}$ and $ts \notin \mathcal{L}(\mathcal{A})$, iff s_k is a *definitely accepting state* according to the following definition:

Definition 6.22 (Definitely Accepting State)
Let POTSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ be given. A state $s \in S$ is called a definitely accepting state, iff the following conditions hold:

1. $s \in F$ (s is a fair state)
2. $\sharp out(s) = 1 \wedge t = (s, true, s, \emptyset, true) \in out(s)$.
   (s has only one outgoing transition t, which is a self-loop. Both the *enable*-predicate of t and the *timing*-predicate are true, and no clocks are reset)

Let $fin_{\mathcal{A}} \subseteq F \quad (\subseteq S)$ denote the set of definitely accepting states of $\mathcal{A}$.

■

If POTSA $\mathcal{A}$ enters a state $s \in fin_{\mathcal{A}}$ for some timed observation sequence ts, then ts is accepted by this *'logical instance'* of $\mathcal{A}$ regardless of any possible extension of ts. Hence, adding transitions from $s \in fin_{\mathcal{A}}$ back to the initial state which are triggered with some reactivation condition, does not affect the acceptance of the first logical instance of $\mathcal{A}$ but logically starts a new instance of $\mathcal{A}$. This motivates the following definition:

Definition 6.23 (Non-Overlapping Reactivation of POTSA)
Let

$$react : POTSA \times Pred_{\mathcal{V}} \rightarrow TSA$$

be a transformation of POTSA which modifies the self-loops of the states in $fin_{\mathcal{A}}$.
For POTSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$, let $\mathcal{A}^{\nwarrow} := react(\mathcal{A}, \alpha)$ be a TSA $\mathcal{A}^{\nwarrow} := (\mathcal{V}, S, s_0, C, T^{\nwarrow}, F)$, where

$$\begin{aligned} T^{\nwarrow} := \quad & T \backslash \{(s, true, s, \emptyset, true) \mid s \in fin_{\mathcal{A}}\} \\ & \cup \{(s, \neg(\alpha), s, \emptyset, true) \mid s \in fin_{\mathcal{A}}\} \\ & \cup \{(s, \alpha, s_0, C, true) \mid s \in fin_{\mathcal{A}}\} \end{aligned}$$

■

react preserves determinism: if $\mathcal{A}$ is deterministic, then so is $react(\mathcal{A}, \alpha)$. The proof is trivial, because *react* only 'splits' the self-loops of the states in $fin_{\mathcal{A}}$ into a *mutual exclusive* reactivation-transition and a restricted self-loop.

The order of applying of *react* and the clock algorithm is not exchangeable, because *react* yields a cyclic TSA to which the clock algorithm is not applicable, because its states are not partially ordered. *react* preserves global constrainedness: by definition of $fin_{\mathcal{A}}$ the states of $fin_{\mathcal{A}}$ are (i) fair states which are not constrained at their self-loop and (ii) have no successor in $\mathcal{A}$. Adding reactivation transitions introduces no unfairness. Hence, if all unfair states in $\mathcal{A}$ are explicitly constrained, then they are in $react(\mathcal{A}, \alpha)$.

We now can formally define a activation control for POTSA:

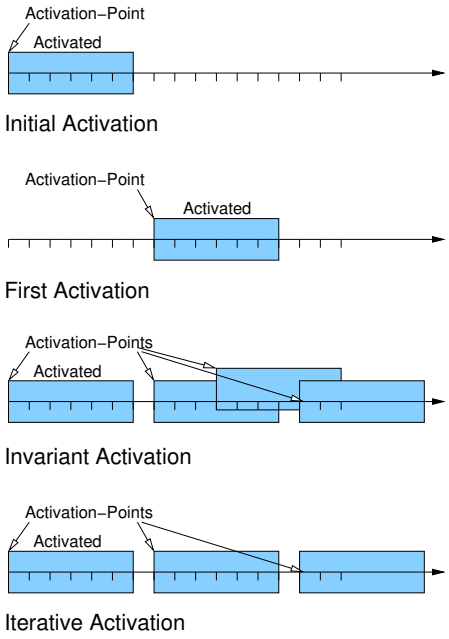


Figure 6.4: Activation Modes of POTSA$_{\mathrm{AC}}$

Definition 6.24 (POTSA with Activation Control (POTSA$_{\mathbf{AC}}$))
A POTSA with activation control (POTSA$_{\mathrm{AC}}$) is a tuple
$\mathfrak{A}$:=$(actmode, e_{ac}, e_{ae}, \mathcal{A})$, where

- $\mathcal{A} = (\mathcal{V}, S, s_0, C, T, F)$ is a POTSA
- $actmode \in \{initial, first, invariant, iterative\}$ is the activation mode for $\mathcal{A}$
- $e_{ac} \in Pred_{\mathcal{V}}$ is the activation condition, for which $\mathcal{A}$ is activated
- $e_{ae} \in Pred_{\mathcal{V}}$ is an activation exception, for which activation of $\mathcal{A}$ is canceled. Activation exceptions are only relevant for POTSA$_{\mathrm{AC}}$ with initial activation. For activation modes *first*, *iterative* and *invariant* an activation exception is not regarded.

Using the definition of *react*, activation mode *iterative* is defined by:

$$(iterative, e_{ac}, e_{ae}, \mathcal{A}):=(first, e_{ac}, e_{ae}, react(\mathcal{A}, e_{ac}))$$

■

Figure 6.4 illustrates the different activation modes.

A POTSA$_{\mathrm{AC}}$ $\mathfrak{A} = (initial, e_{ac}, e_{ae}\mathcal{A})$ restricts the normal initial activation of its instantiated POTSA $\mathcal{A}$ by the additional activation condition e_{ac} and activation exception e_{ae}. $\mathfrak{A}$ activates $\mathcal{A}$ only for timed observation sequences ts for which which $\sigma_0 \models e_{ac}$ holds. All timed observation sequences ts, for which $\sigma_0 \models (e_{ae} \wedge \neg e_{ac})$ holds are accepted by $\mathfrak{A}$ without activation of $\mathcal{A}$.

In particular for $\mathfrak{A}$:=$(initial, true, false, \mathcal{A})$, we have $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathcal{A})$.

A POTSA$_{\text{AC}}$ with *invariant* activation $\mathfrak{A} = (invariant, e_{ac}e_{ae}, \mathcal{A})$ activates a new instance of POTSA $\mathcal{A}$ whenever e_{ac} evaluates to true for some σ_i along timed observation sequence ts[13] . Since $\mathcal{A}$ is activated whenever e_{ac} evaluates to true regardless of which is the currently active state, more than one instance of $\mathcal{A}$ can be active at a time. In general, such overlapping instantiations of POTSA $\mathcal{A}$ will not accept a timed observation sequence. In particular, overlapping instances of an invariantly activated POTSA $\mathcal{A}$ in assumption place will in general contradict each other and thus not accept any timed observation sequence - whenever instantiation can overlap. Hence, at least for the usage as assumption POTSA$_{\text{AC}}$ with iterative activation are preferable over invariant activation. Moreover, generation of an observer for an invariantly activatable POTSA $\mathcal{A}$ in assumption-place would require building the product automaton of $\mathcal{A}$ with itself to be capable of all possible overlapping activations. In general, the resulting product automaton can not be encoded as POTSA, but requires back-leading transition for every state. W.r.t. complexity for application in verification imposed by such a product automaton and the limited use of invariant assumptions, we restrict the usage of invariant POTSA$_{\text{AC}}$ to the commitment place only. For usage of POTSA$_{\text{AC}}$ $\mathfrak{A}$ as commitment, a non-deterministic activation of $\mathcal{A}$ is an adequate realization of invariant activation. Since model checkers are aimed at detecting violations of a specification, a model checker will activate $\mathcal{A}$ only if the commitment specification expressed by $\mathcal{A}$ can be violated for this particular activation. In particular, if $\mathcal{A}$ can reject a timed observation sequence ts when activated for some observation σ_i in ts, then the model checker will activate $\mathcal{A}$ at σ_i. If otherwise, all possible activations for all possible timed observation sequences lead to accepting runs of $\mathcal{A}$, then the model checker will refrain from activating $\mathcal{A}$.

This argument does not apply to the assumption place. Informally, the role of assumptions is constructive by means of filtering possible behaviors of the environment, i.e. selection of timed observation sequences for which the commitment specification has to be checked. Non-deterministic activation is too weak for assumptions, because the assumption is satisfied if it is never activated. Because a commitment can only be violated if all considered observation sequences satisfy the assumptions a model checker will never activate a non-deterministically activatable assumption.

Definition 6.25 (Language of POTSA with Activation Control (POTSA$_{\text{AC}}$))

The language accepted by a POTSA$_{\text{AC}}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ is defined over timed observation sequences.

Let $ts = (\pi, \tau)$ be a timed observation sequence, where $\pi = \sigma_0\sigma_1\sigma_2...$ is a sequence of valuations of $\mathcal{V}$. Let $\overrightarrow{ts_i}$ denote the suffix of ts starting with the i-th position. Then :

$$ts \in \mathcal{L}(\mathfrak{A}) := \begin{cases} (\sigma_0 \models e_{ae} \wedge \sigma_0 \not\models e_{ac}) & \\ \vee(\sigma_0 \models e_{ac} \quad \wedge \quad ts \in \mathcal{L}(\mathcal{A})) & \text{iff } actmode = initial \\ (\exists i \geq 0 : \sigma_i \models e_{ac} \wedge \forall k < i : \sigma_k \not\models e_{ac} & \\ \qquad \wedge \quad \overrightarrow{ts_i} \in \mathcal{L}(\mathcal{A})) & \text{iff } actmode = first \\ \qquad \vee(\nexists i \geq 0 : \sigma_i \models e_{ac}) & \\ (\exists i \geq 0 : \sigma_i \models e_{ac} \wedge \forall k < i : \sigma_k \not\models e_{ac} & \\ \qquad \wedge \quad \overrightarrow{ts_i} \in \mathcal{L}(react(\mathcal{A})) & \text{iff } actmode = iterative \\ \qquad \vee(\nexists i \geq 0 : \sigma_i \models e_{ac}) & \\ \forall i \geq 0 : \sigma_i \models e_{ac} \quad \Rightarrow \quad \overrightarrow{ts_i} \in \mathcal{L}(\mathcal{A}) & \text{iff } actmode = invariant \end{cases}$$

■

[13] As stated in definition 6.24, e_{ae} is ignored for the invariant activation mode as well as for iterative activation.

Although, the language of invariantly activated POTSA is formally defined in the above definition, the implementation of the observer generation for POTSA_{AC} restricts the use of invariantly activated POTSA to the commitment place.

Lemma 6.10 (Preservation of Determinism)
Let $POTSA_{AC}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ with $actmode \neq invariant$ be given. If POTSA $\mathcal{A}$ is deterministic, then so is $\mathfrak{A}$.

■

Proof 6.10
Follows from definition 6.25:
initial activation activates $\mathcal{A}$ deterministically for the first valuation in ts iff $\sigma_0 \models e_{ac}$. Activation exception is mutual exclusive:$(\sigma_0 \models e_{ae} \wedge \sigma_0 \not\models e_{ac}) \Rightarrow \neg(\sigma_0 \models e_{ac})$ and $(\sigma_0 \models e_{ac}) \Rightarrow \neg(\sigma_0 \models e_{ae} \wedge \sigma_0 \not\models e_{ac})$.
$first$ activation activates $\mathcal{A}$ deterministically for the *first* position i in ts , s.t. $(\sigma_i \models e_{ac} \wedge \forall k < i : \sigma_k \not\models e_{ac})$.
iterative activation is based on activation mode $first$ and definition of *react*. It has been shown, that *react* preserves determinism.

□

6.3.7 Observer Generation for POTSA_{AC}

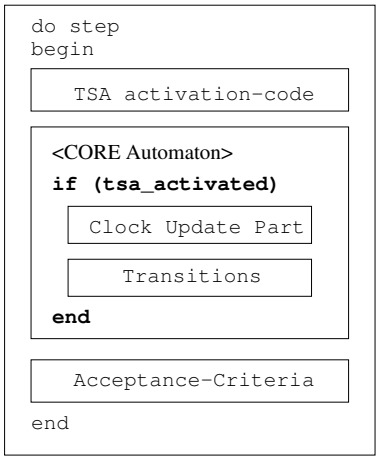


Figure 6.5: Structure of SMI Observer Encoding of POTSA_{AC}

In figure 6.5 the structure of the SMI representation for POTSA_{AC} $\mathfrak{A}$ is illustrated schematically. The representation of the instantiated POTSA $\mathcal{A}$ is the same for *initial*, *first* and *invariant* activation. Only the *iterative* activation imposes a modification on $\mathcal{A}$: reactivation transitions from the states in $fin_{\mathcal{A}}$ to the initial state are added and the *enable* predicates of the self-loops of the states in $fin_{\mathcal{A}}$ are restricted to $\neg e_{ac}$ in order to represent $react(\mathcal{A})$ according to definition 6.23.

Listings 6.6 and 6.7 describe algorithmically the generation of a SMI observer representation for $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$. The representation of the instantiated POTSA $\mathcal{A}$ is embedded in a `CASE`

statement which activates the code only for a truth-valuation of `tsa_activated` which is a boolean variable that is set to true in the respective activation code blocks in order to activate the code representing the instantiated POTSA. The representations of the different activations can be found in listings 6.10, 6.11 and 6.12 respectively. Since the clock update part is created exactly the same as in listing 6.3 on page 116 we omit this part in listing 6.6 in order to avoid repetitions.

```
< 𝔄_SMI >:=create_SMI_program(<>)

create_activation_code_block(< 𝔄_SMI >, actmode , e_ac , e_ae , s_0 , relevant_clocks )

< core >:=create_deterministic_choice(< 𝔄_SMI >)
< 𝒜_SMI >:=create_case_in_choice(< core >,tsa_activated' == true)

create_clocks_update(< 𝒜_SMI >)

create_assignment(< 𝒜_SMI >,stuck', true)

< ndcase >:=create_nondeterministic_choice(< 𝒜_SMI >)
foreach ( s ∈ S ) {
    foreach ( t ∈ out(s) ) {
      if ((s ∈ fin_𝒜 ) and (act_mode == iterative)) then {
            /* here react(𝒜, e_ac) is applied */

            /* stay in s as long as "not(e_ac)" holds */
            < modified_selfloop >:=create_case_in_choice(< ndcase >,
                    ( primed_ref ( enc_S ( s )) == true ) and not ( enc_pred ( e_ac )))
            create_transition_action_code(< modified_selfloop > , s , s)
            /* if "e_ac" holds then re-enter s_0 */
            < reactivation_trans >:=create_case_in_choice(< ndcase >,
                    ( primed_ref(enc_S ( s )) == true ) and (enc_pred ( e_ac )))
            create_transition_action_code(< reactivation_trans > , s , s_0)
       } else {
            /* this is the normal transition encoding */

            < transition >:=create_case_in_choice(< ndcase >,
                  ( primed_ref(enc_S ( s )) == true ) and ( enc_pred ( t.enable)) and
                      primed_ref( enc_Ψ ( t.timing )))
            create_transition_action_code(< transition > , s , t.target)
       }
    }
}
< case >:=create_case_in_choice(< ndcase >, sink_state' == true)
< ndcase >:=create_SKIP_statement(< case >)

< dcase >:=create_deterministic_choice(< 𝒜_SMI >)
< case >:=create_case_in_choice(< dcase >,stuck' == true)
create_assignment(< case >, sink_state', true )
```

Listing 6.6: SMI-Generation Algorithm for 𝔄: Main Part

In order to increase the readability of listing 6.6 and due to limited space, generation of the

action-part for the representation of transitions has been extracted and can be found as procedure-declaration in listing 6.8.

```
44
if (act_mode == initial) then
46   create_assignment(< 𝔄_SMI >, fair_cond′,
       not(tsa_activated′ == true) or (initial_accept′ == true) or
48       (generate_primed_disjunction( {s ∈ F} ))
else
50   create_assignment(< 𝔄_SMI >, fair_cond′,
       not(tsa_activated′ == true) or (generate_primed_disjunction( {s ∈ F} ))
52 if ( 𝒜 is explicitly constrained) then
     create_assignment(< 𝔄_SMI >, nfa_cond′, not(sink_state′ == true) )
```

Listing 6.7: SMI-Generation Algorithm for $\mathfrak{A}$: Acceptance Criteria

```
procedure create_tranition_action_code (context scope, state source, state target) {
    create_assignment(< scope >, primed_ref( enc_S( source )), false )
    create_assignment(< scope >, primed_ref( enc_S( target )), true )
    create_assignment(< scope >, stuck′, false )
}
```

Listing 6.8: Procedure for Action Part Generation of Transition

```
input boolean enc_pred(e_ac)
input boolean enc_pred(e_ae)
foreach (t ∈ T) {
    input boolean enc_pred( t.enable )
}
foreach (s ∈ S) {
    local boolean enc_S( s ):=false
}
foreach (c ∈ C) {
   /* encode clocks in integer_domains according to the definition of
      mc : C → ℕ_0 of lemma 6.1 */
   local int_type(0,mc(c) + 1) enc_C( c ):=0
}
local boolean tsa_activated:=false
local boolean initial_step:=true
local boolean initial_accept:=false
local boolean stuck:=true
local boolean sink_state:=false
output boolean fair_cond:=true
output boolean nfa_cond:=false
```

Listing 6.9: Variables and their Initialization

In listing 6.9, the used variables of the SMI representation as well as their initialization can be found. Note, that inputs are not initialized, since their values are controlled by the environment.

Initial Activation

Initial activation can only take place for the first observation of a timed observation sequence, i.e. in the first step performed by the SMI observer encoding $\mathfrak{A}$. In order to avoid later activations a special variable `initial_step` is introduced, which is initialized to `true` and set to `false` after considering the variable for the first time. Hence, the code block for initial activation is considered only in the first step. If in the first step e_{ac}evaluates to true, the initial state of the $\mathcal{A}$ is activated and `tsa_activated` is set to `true`. If e_{ac} is `false` whereas e_{ae} is `true` in the first step, then the local variable `initial_accept` is set to `true`. `initial_accept` is considered in the definition of `fair_cond` according to line 46 of the algorithm. If both e_{ac} and e_{ae} evaluate to `false`, then the `sink_state` is activated.

```
DCASE
   [] not (tsa_activated) and (initial_step ) :
      initial_step ':=false
      DCASE
         [] enc_pred(e_ac) == true :
            tsa_activated' := true
            /* set initial state to true */
            primed_ref(enc_S(s_0)) := true
            set all clocks to 0
         [] not (enc_pred(e_ac) == true) :
            DCASE
               [] (enc_pred(e_ae) == true ) :
                  initial_accept ' := true
               [] not (enc_pred(e_ae) == true ) :
                  sink_state':=true
            DESAC
      DESAC
DESAC
```

Listing 6.10: Initial Activation

Invariant Activation

If the core automaton has not already been activated (`tsa_activated==false`) and if the activation condition e_{ac} is true, the activation code non-deterministically either sets `tsa_activated` to `true` and activates the initial state of the core POTSA $\mathcal{A}$ or leaves `tsa_activated` unchanged.

```
DCASE
   [] not (tsa_activated)
      NDCASE
         [] enc_pred(e_ac) == true :
            SKIP
         [] enc_pred(e_ac) == true :
            tsa_activated' := true
            /* set initial state to true */
            primed_ref(enc_S(s_0)) := true
            set all clocks to 0
```

```
        [] not ( enc_pred(e_ac) == true) :
           SKIP
    NDESAC
DESAC
```

Listing 6.11: Invariant Activation

First and Iterative Activation

Finally, first and iterative activation enables execution of the code block encoding $\mathcal{A}$ deterministically by setting `tsa_activated` to `true` if e_{ac} is true and activation not already took place. Recall, that iterative reactivation is encoded in the representation of $\mathcal{A}$'s transition relation (cf. lines 16-27 of the algorithm).

```
DCASE
   [] not (tsa_activated)
      DCASE
         [] enc_pred(e_ac) == true :
            tsa_activated' := true
            /* set initial state to true */
            primed_ref(enc_S(s_0)) := true
            set all clocks to 0
         [] not (enc_pred(e_ac)== true) :
            SKIP
      DESAC
DESAC
```

Listing 6.12: First and Iterative Activation

Definition 6.26 (SMI Observer-Module of POTSA$_{\mathbf{AC}}$)
Let POTSA$_{\text{AC}}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ be given.
Let $\Omega(\mathfrak{A})$ denote the observer SMI module representing POTSA$_{\text{AC}}$ $\mathfrak{A}$, which is obtained by encoding $\mathcal{A}$ into SMI together with an activation control according to the definition of activation mode *actmode*, activation condition e_{ac} and activation exception e_{ac}.
Let $\Omega(\mathfrak{A})\circ_f f$ denote the observer SMI module representing POTSA$_{\text{AC}}$ $\mathfrak{A}$ with designated *fairness condition* output f.
Moreover, if $\mathcal{A}$ is an explicitely constrained POTSA, let $\Omega(\mathfrak{A})\triangleright o$ denote the observer SMI module representing $\mathfrak{A}$ with designated *non-failure acceptance condition* output o.

■

For $\Omega(\mathfrak{A})\circ_f f$, f is the stepwise indicator of the fairness condition (`fair_cond` in listing 6.7 and 6.9, respectively), which is true in a step iff :

- $\mathcal{A}$ is in one of its fair states,
- (if $actmode = initial$) $\mathfrak{A}$ has *initially* accepted timed observation sequence ts according to the definition of activation condition $e_{ac}$and activation exception e_{ae}.
- (if $actmode = iterative$ or $actmode = invariant$) $\mathcal{A}$ has not been activated yet, according to the definition of e_{ac}.

For $\Omega(\mathfrak{A}) \triangleright o$, o is the stepwise indicator of the non failure acceptance condition (`nfa_cond` in listing 6.7 and 6.9, respectively), which is true in a step iff :

- *sink_state* is not the active state of the instantiated TSA $\mathcal{A}$ (which is also the case, if $\mathcal{A}$ has not been activated yet).

Theorem 6.3 (Verification using POTSA$_{AC}$)
Let CSTS $\mathcal{C} = (V, \Theta, \rho, E)$ and POTSA $\mathcal{A} = (\mathcal{V}, S, s_0, C, T_A, F')$ be given.

1. *For the parallel composition $\mathcal{C}_{par} := \mathcal{C}||_\Omega \Omega(\mathfrak{A}) \circ f$ of $\mathcal{C}$ with observer $\Omega(\mathfrak{A}) \circ f$ obtained from POTSA$_{AC}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ holds :*

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{AGAF}(f) \quad \Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{\mathcal{V}} \in \mathcal{L}(\mathfrak{A}), \tag{6.14}$$

   where $TComps(\mathcal{C})$ is the set of all possible timed observation sequences of $\mathcal{C}$.

   Moreover, iff $\mathcal{A}$ is deterministic and actmode $\neq$ invariant, the following holds:

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{AGAF}(f) \quad iff \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{\mathcal{V}} \in \mathcal{L}(\mathfrak{A}). \tag{6.15}$$

2. *If $\mathcal{A}$ is deterministic, then for $\mathcal{C}_{par} := \mathcal{C}||_\Omega \Omega(\mathfrak{A}) \circ f$ with observer $\Omega(\mathfrak{A}) \circ f$ obtained from POTSA$_{AC}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ with actmode $\neq$ invariant holds:*

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{GF}(f) \quad iff \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{\mathcal{V}} \in \mathcal{L}(\mathfrak{A}). \tag{6.16}$$

3. *If $\mathcal{A}$ is deterministic and explicitely constrained, then for $\mathcal{C}_{par} := \mathcal{C}||_\Omega \Omega(\mathfrak{A}) \triangleright o$ with observer $\Omega(\mathfrak{A}) \triangleright o$ obtained from POTSA$_{AC}$ $\mathfrak{A} = (actmode, e_{ac}, e_{ae}, \mathcal{A})$ with actmode $\neq$ invariant holds:*

$$\mathcal{K}(\mathcal{C}_{par}) \models \texttt{G}(o) \quad iff \quad \forall ts \in TComps(\mathcal{C}) : ts \downarrow_{\mathcal{V}} \in \mathcal{L}(\mathfrak{A}). \tag{6.17}$$

■

Proof of Theorem
(6.14) follows directly from theorem 6.1 (regarding CTL verification).
(6.15) follows from lemma 6.10 and theorem 6.1 (regarding CTL verification).
(6.16) follows from lemma 6.10 and lemma 6.5 (regarding LTL verification)
(6.17) follows from lemma 6.10 and lemma 6.9 (regarding verification using non-failure acceptance)
□

Conclusion

Timed Symbolic Automata provide a well defined semantical basis for the generation of (fair) synchronous observers, which are capable of quantitative treatment of time. It has been shown that for a relevant sub-class of TSA invariance checking instead of LTL model checking with fairness constraints is applicable.

6.3.8 Related Work

Symbolic Automata (SA) and in particular the subclass of partially ordered symbolic automata (POSA) were - to our knowledge - first presented in the PhD-Thesis of Rainer Schlör [Sch00]. There, POSA serve as semantical basis for the definition of symbolic timing diagrams and linear symbolic timing diagrams respectively. In particular, the correspondence of deterministic POSA and LTL is considered and a LTL formula generation for deterministic POSA is presented. This correspondence forms the basis of the integration of symbolic timing diagrams with verification techniques, where POSA do not refer to time explicitly, but only the usual temporal operators `X`, `G`, `F`, and `U` permit reference to temporal relations of observations in this logical frame-work.

For an earlier RT-version of symbolic timing diagrams (cf. section 6.5) Konrad Feyerabend [FJ97, Fey96] presented a timed variant of POSA serving as intermediate representation for the translation of RT-symbolic timing diagrams into TPTL^C. The real-time temporal logic TPTL^c presented in [Fey96, FJ97] is a derivative of the freeze quantifier logic TPTL [AH89]. In TPTL specification clocks are introduced, which are frozen to the actual value of a global clock by so-called freeze-quantification. Therefore, propositional temporal logic is extended with an infinite supply C of specification clocks, which can be frozen to the actual global time and referred to within the formula. Whenever a freeze quantifier *z.* is encountered in a formula, the actual value of the global time is stored in a variable z for later reference. Global time is only accessible through freezing. In contrast to TPTL, TPTL^c uses *reset quantification* instead of *freeze quantification*. Whenever a reset quantifier *z.* is encountered in a formula the clock variable *z* is set to 0 instead of setting *z* to the global time. Hence, as in our approach specification clocks count time units since their last reset. All clock variables are incremented with the global time according to a timed observation sequence. The timed variant of symbolic automata as presented in [Fey96, FJ97] only regards one sort of clocks. Hence, either counting steps can be captured by TPTL^c or interpretation of clocks w.r.t. simulation time, but not a combination of both. Due to unavailability of a Tableau generation for TPTL^c, this interpretation has - to the best of our knowledge - been used in practice only for counting steps. Based on the hard-coded assumption that 'next time' is interpreted as 'next step' , clock environments for step-counters have been unwound into sequences of `X` operators. Hence, ordinary LTL formulae have been generated from RTSTD for application of verification.

This pragmatic approach of only referring to steps has been followed also by Jochen Klose in [Klo03], where timed symbolic automata are chosen as basis for the semantics definition of Live Sequence Charts (LSC). The TSA definition in [Klo03] is closely related to our approach and has served as semantical basis for LSC already in an earlier common publication [KW01]. We have extended the definition of TSA by an enhanced notion of time and continued the formalization where [Klo03] ends. Consequently, the enhanced treatment of time as well as the results of this section could easily be transferred to the formalism of Live Sequence Charts.

A first version of an automatic observer generation for POTSA$_{\mathrm{AC}}$ for only one sort of clocks was developed by Bertrand Gregoire in the context of his master thesis [Gre02]. Large parts of the research and implementation took place in a co-operation with the C.v.O-University of Oldenburg and OFFIS. This work, which we had the pleasure to advise, forms the basis of this section.

A major motivation for this co-operative research activity has been the identification of a restricted subclass of POTSA for which *non-failure acceptance* can be used instead of Büchi acceptance.

Most of the definitions and conclusions presented in [Gre02] are given in a rather informal way. The requirements regarding POTSA as well as the clock-algorithm and applicability-criteria for Non-Failure Acceptance, respectively, are rather ad-hoc in [Gre02], which imposed a re-consideration and

formal foundation of the formalism.

Also tableaux techniques for temporal logic formulae as for example explained in [BCM+90, CGH97, CGP99] represent specifications by automata for the application of verification.

On the one hand, a direct representation of TSA by observers avoids application of complex algorithms, involving (1) generation of a formula representation for TSA and (2) afterwards generating automata representations for the obtained formulae. On the other hand, representing TSA as observers permits application of invariance checking for an important subclass of TSA instead of applying more complex verification techniques. For all liveness requirements, for which a particular quantitative bound restricts the required or expected reaction[14], invariance checking is applicable, conserving a linear time interpretation.

Verification of safety (including bounded liveness) requirements using synchronous observers has been considered by Halbwachs et al. in [HLR93] and for example in [Hol00] but also by many others in the context of synchronous languages. To the best of our knowledge, the observers applied in the cited approaches are hand written specifications using the same language as for the model itself, but are not generated automatically from more abstract formats, such as TSA.

6.4 Observer Pattern

For the formalization of rather standard requirements the verification framework offers a variant of Timed Symbolic Automata, so-called *specification pattern*. These pattern are predefined propositional schemes for capturing basic temporal requirement specifications and are designed to be applicable in a very flexible way. By analysis of a large set of industrial requirements from avionics, automotive and rail-system application domains, a set of most frequently applied temporal schemes has been identified.

For each of these schemes an automaton pattern implementation is provided by a library. Pattern can be used to specify assumptions about the environment as well as to specify commitments, which have to be fulfilled by the considered component.

Pattern specify temporal relationships of user-defined events, which have to be specified by the user. Therefore, transition triggers as well as upper and lower bounds for clocks are implemented using formal parameters.

A pattern is instantiated by choosing it from the library and by mapping its formal parameters to actual expressions referring to expressions ranging over model observables as desired for the actual requirement specification. Two kinds of parameters are distinguished for instantiation of pattern in specifications:

- *proposition*-parameters are mapped to user-defined expressions ranging over basic states, events and data-items of the model.
- *bound*-parameters are mapped to positive values, in order to form an upper bound to a counter of the pattern. It depends on the chosen execution semantics of the Statemate model whether the counter refers to steps or super-steps.

In contrast to TSA as discussed in the previous section, pattern are combined with an additional "prefix automaton": All pattern are available for three possible *system setup phases* of the system.

[14] aka. bounded liveness requirements.

System setup phases permit an initialization phase of the system before first activation of the pattern, regardless of the particular activation mode of the pattern:

after_reaching_R first activation of the pattern according to its activation condition can take place only after the system first satisfies condition R, where R is a proposition parameter of the respective pattern implementation .

after_N_steps first activation of the pattern only takes place N step or super-steps, respectively, after initialization of the system

immediate no setup phase is regarded; the pattern is first activated according only to its activation mode and condition.

According to the different setup phases and activation modes, the set of pattern is structured following the naming scheme:

<activation_ mode>_<kernel_ property>__<startup_ phase>

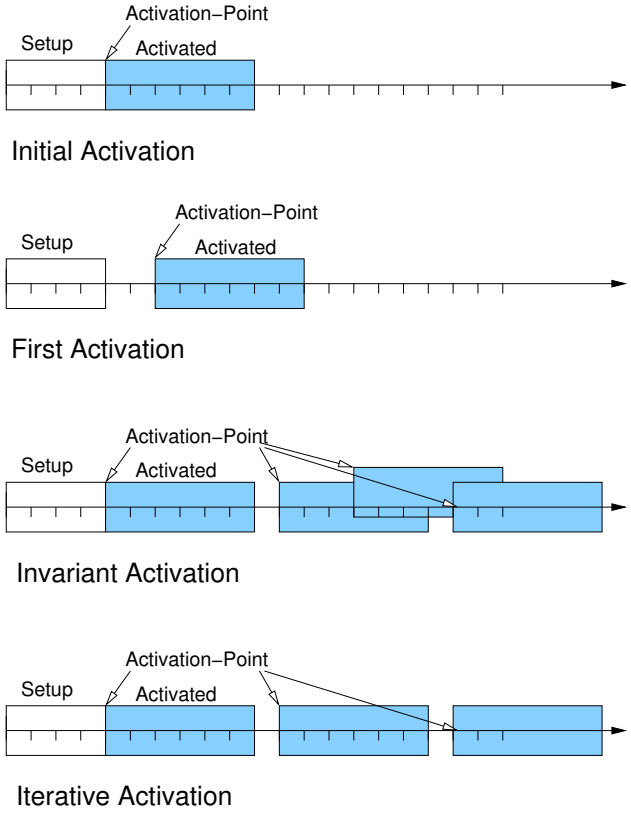


Figure 6.6: Activation Modes

For example, `inv_P__after_reaching_R` (with kernel property `P`; kernel properties will be explained below) implements a pattern expressing the requirement that proposition `P` holds invariantly

after the first occurrence of proposition R. In contrast, inv_P__immediate requires that proposition P holds immediately from system start, while init_P__immediate requires that proposition P holds at system start, without caring about the subsequent steps. Consequently, init_P__after_N_steps states that proposition P is true N steps or super-steps respectively after system start. Figure 6.6 illustrates the interplay of startup phases, activation-modes and kernel pattern.

The interpretation of time depends on the chosen execution semantics for the specified component. Clocks refer to steps if the model refers to the synchronous execution semantics. Otherwise, clocks refer to stable states of the model.

In the following we list the pattern offered by the verification framework at the moment of writing. In principle, each of the kernel pattern is defined for at least the activation modes initial, first and iterative. Table 6.1 lists the supported combinations of kernel pattern with activation modes. Some of the pattern also provide an invariant activation mode.

Pattern can be instantiated in commitment as well as in assumption place. Thus, whenever we use the verb "require" in the explanations below, it can be re-interpreted as "expect" for the use of the pattern as assumption:

P Proposition P. Obviously only a combination with activation modes *initial* or *invariant* is meaningful. In the special case of kernel pattern **P,** activation mode invariant has to be understood in a different way than usual: "invariantly P" means the same as G(P).

P_implies_finally_Q_B If proposition P is observed then it is required that proposition Q is observed at most B time units after P.

finally_P_B It is required that proposition P is observed at most B time units after activation of the pattern.

P_implies_finally_globally_Q_B If proposition P is observed then it is required that proposition Q is observed at most B time units after observation of P and then holds forever.

finally_globally_P_B Proposition P is required to be observed at most B time units after activation of the pattern and then to hold forever. The same as for kernel property P, only a combination with activation modes *initial* and *invariant* is meaningful, since the pattern is activated unconditionally.

P_implies_globally_Q If proposition P is observed then it is required that proposition Q holds forever from the same instant of time on in which P has been observed.

P_implies_Q_X_steps_later If proposition P is observed then it is required that proposition Q is observed at exactly B time units after observation of P.

P_implies_Q_during_next_X_steps If proposition P is observed then it is required that proposition Q holds at the same instance of time and since then to remain valid for the next X time units.

P_implies_Q_atleast_X_steps_after_P If proposition P is observed then it is required that proposition Q holds not earlier than X time units after observation of proposition P.

P_stable_X_steps_implies_afterwards_Q If proposition P holds for an interval of X time units, then proposition Q is required to be observed afterwards.

P_stable_X_steps_implies_finally_Q_B If proposition `P` holds for an interval of `X` time units then proposition `Q` is required to be observed at most `B` time units later (after completing the minimal interval length - regardless of proposition `P` in between).

Q_while_P Proposition `Q` is required to hold as long as proposition `P` holds.

Q_while_P_B Proposition `Q` is required to hold as long as proposition `P` holds, where `P` has to hold at least `B` time units.

Q_only_after_P Proposition `Q` is required to be observed *only after* proposition `P` has been observed.

Q_not_before_P Proposition `Q` is required to be observed *not before* proposition `P` has been observed.

All pattern specify progress at most in the form of bounded liveness. Thus all properties that can be specified using pattern are safety properties, for which invariance checking is applicable.

activation mode kernel property	Initial	First	Invariant	Iterative
P	x	-	x	-
P_implies_finally_Q_B	x	x	x	2)
finally_P_B	x	-	x	-
P_implies_finally_globally_Q_B	x	x	x	2)
finally_globally_P_B	x	-	1)	-
P_implies_globally_Q	x	x	x	2)
P_implies_Q_X_steps_later	x	x	-	x
P_implies_Q_during_next_X_steps	x	x	-	x
P_implies_Q_atleast_X_steps_after_P	x	x	-	x
P_stable_X_steps_implies_afterwards_Q	x	x	-	x
P_stable_X_steps_implies_finally_Q_B	x	x	-	x
Q_while_P	x	1)	x	2)
Q_while_P_B	x	1)	-	2)
Q_only_after_P	x	1)	-	x
Q_not_before_P	x	1)	-	x

Table 6.1: Pattern Overview

x : available
-: not available
1) same as initial
2) same as invariant

Figure 6.7 shows an example pattern implementation.

Pattern `P_implies_Q_X_steps_later` provides the proposition parameters `P` and `Q`, while `X` is a bound parameter specifying an upper bound for the amount of steps or super-steps, respectively, after which `Q` has to be observed - after observation of `P`.

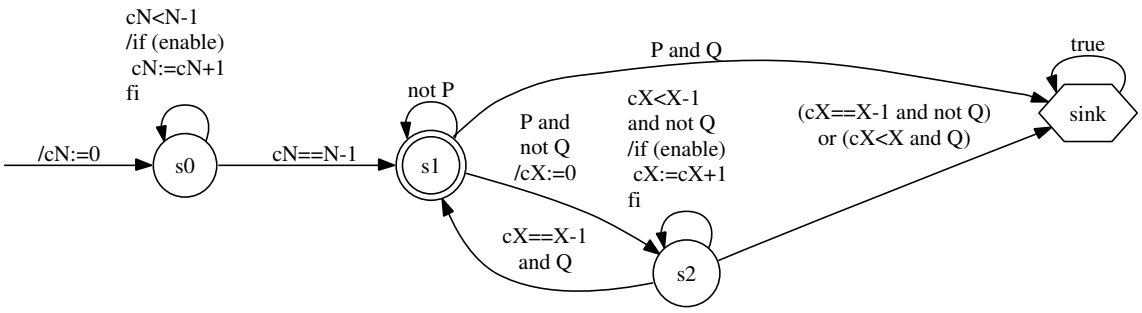


Figure 6.7: Pattern "`iter_P_implies_Q_X_steps_later__after_N_steps`"

`X` and `N` are bound parameters determining upper bounds for the clocks `cX` and `cN`, respectively. By setting the parameter `enable` to either `''SUPER_SYNC=true''` or `''true''` it can be determined whether the counters refer to steps or super-steps. Parameter `enable` is not offered to the user, but automatically mapped at instantiation time according to the execution semantics to which the specified model refers.

Clock `cN` and state `s0` are used to implement the setup phase `after_N_steps`. If `cN` has reached its maximal value - bounded by `N`, the transition (`s0` to `s1`) to the kernel pattern (states `s1`,`s2`, and `sink`) is taken. The pattern remains in state `s2` unless the expression mapped to `P` evaluates to true. If at the same step also the expression mapped to `Q` becomes true, the transition to the sink-state `sink` is taken in which the execution remains forever. Otherwise counter `cX` is reset and state `s2` is entered. If the expression mapped to `Q` becomes true too early or too late, the sink-state `sink` is entered. Only if the expression evaluates to true exactly at the right step, `s2` is left and `s1` is entered again - the kernel pattern is ready for reactivation. Since the activation mode is encoded in the pattern, the *initial* and *first* variant (figure 6.8) require an additional accepting state `s3` , which is entered instead of entering `s1` again, because if the pattern has been activated once no further activation is possible. For the *initial* activation the self-loop at state `s1` has to be removed and instead a transition to the `sink_state` has to be added which is taken if `P` is not observed.

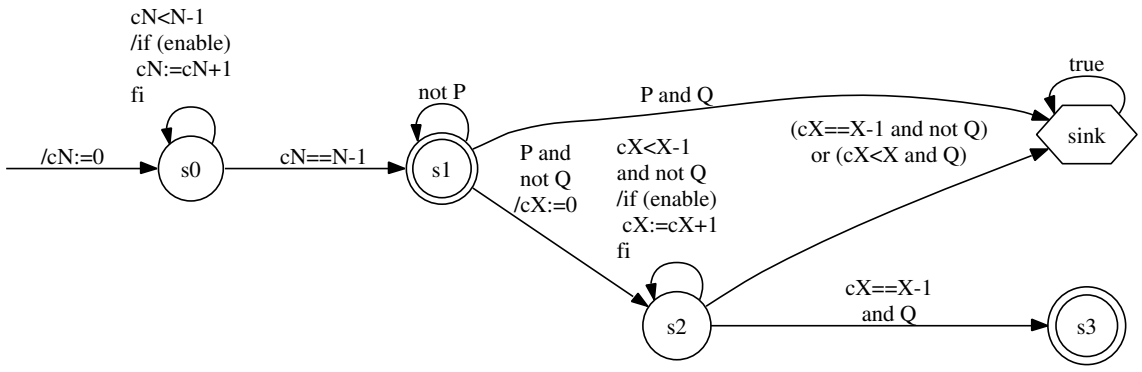


Figure 6.8: Pattern "`first_P_implies_Q_X_steps_later__after_N_steps`"

6.4.1 Related Work

Pattern and templates are well known from many application areas. For example, program libraries provide standardized functions and procedures for programmers; word processing systems provide text templates for various use-cases; modern CASE-tools offer libraries of design patterns.

The typical reason to introduce pattern for an application area is that pattern solve a recurrent problem, for which either the solution is not obvious to every user or for which a particular proven concept can be offered, which is advanteguous to use.

Specification of requirements for formal verification in terms of temporal logic requires considerable expert knowledge, and thus hinders the use of formal verification techniques by non-expert engineers.

Several approaches to pattern-based verification exist, which aim at offering simple but formal specification techniques, in order to enable non-expert users to apply formal verification. A fundamental observation regarding formal specification is that often requirements are identical except for the concrete observables or expressions to which they refer. As an example the basic definition of liveness and safety specification by Lamport could be cited, which states that safety properties are of the general form "*Something bad will never happen*", whereas liveness properties are of the form "*Something good will finally happen*". Obviously, many safety specifications are of the form "*event b must not occur before event a*" or "*event b must occur only after event a*" or similar formulations, while many liveness specifications are for example of the form "*event a implies that finally event b must occur*" .

Analysis of typical requirements thus exposes often recurring specification schemes, which can be offered as templates for specifications of similar properties. The semantical representation of such specification schemes has to be realized only once and an adequate mapping mechanism associating user-defined event- or condition-specifications to the formal parameters of the template has to be offered.

For example, Dwyer, Avrunin and Corbett [DAC98a, DAC98b] report on an analysis for which they collected over 500 examples of property specifications, for which they found that 92% could be covered by instances of their pattern collection. This collection consists of pattern of a number of formalisms, e.g. CTL, LTL and Quantified Regular Expressions. Pattern are classified according to the categories (1) occurrence: absence, existence, bounded existence, universality, and (2) order: precedence, response, chain precedence, and chain response. The advantage of this approach is a lucid classification and a broad coverage of recurring specification. Obviously, a drawback of this multi-formalism approach is that the collection is rather a knowledge base than tool support for users, which offers tight integration with a concrete verification environment. In contrast, the observer pattern presented in this section are offered and integrated as ready-to-use specification templates in the STVE.

Bitsch [Bit00, Bit01] presented a pattern collection for the formalisms CTL, LTL and μ-calculus, which follows the classification : (1) necessary behavior, (2) permitted or forbidden behavior, (3) necessary behavior which is only permitted, and (4) behavior, which must be guaranteed under certain conditions. As the approach of Dwyer et al., also this approach offers rather a catalogue as knowledge base, than tool support for application in verification.

Another interesting approach of pattern-based specification is the specification logic Sugar of the IBM Haifa Research Laboratory [AFF+02]. Based on the observation, that engineers are rather familiar with regular expressions than with temporal logic, the specification logic allows the specification of repetition, concatenation, disjunction and conjunction of event sequences or single events

specified in a regular language. In Sugar, regular expressions are combined with temporal operators like `always` and `eventually`. For model checking, the Sugar specification can automatically be translated into one of the standard temporal logics LTL or CTL.

6.5 Symbolic Timing Diagrams (STDx)

The visual formalism STDx (extended Symbolic Timing Diagrams) has been developed for capturing requirement specifications of reactive systems in a graphical way. While the graphical formalisms used in the STATEMATE system are tailored towards an operational description of reactive systems, STDx takes a complementary role: properties of a system or sub-system are specified in terms of its input/output behavior. The semantics of STDx is based on a declarative paradigm: a STDx-specification consists of a set of diagrams (definitions), where each diagram describes one aspect of the required input/output behavior independently. The diagrams are not referred to directly by specifications but a declaration layer permits grouping of diagrams and mapping of formal parameters, which can be used in order to keep the diagrams more general.

STDx-specifications of a (sub-)system A refer to the interface declaration $intf(A)$ as defined in section 5.4. This way, the system is treated as a *black-box,* i.e. local variables, states, conditions and events are hidden to STDx-specifications. The restriction of STDx to interfaces is not stringent from a technological view-point but is of methodological nature: Allowing direct reference to local variables might seem to be more comfortable at first glance, but has the disadvantage that specifications depend on particular implementations. For such dependent specifications, it is not possible to replace the implementation by another implementation with the same interface, but e.g. different local variables. Keeping specifications independent from implementation details is of paramount importance for compositional verification. Restriction of specifications to the externally visible communication behavior of sub-systems permits deduction of system-level properties from the composition of sub-system specifications instead of considering the composition of the concrete sub-systems.

A STDx-diagram consists of a set of symbolic waveforms and constraints, where each waveform represents a totally ordered sequence of symbolic events regarding the interface to which the specification refers. No order is defined between events of different waveforms unless a constraint explicitly requires an order or particular timing between events of the different waveforms.

Informally, a diagram is interpreted by moving a front from left to right through the diagram, such that the front always crosses each waveform only once. Whenever an observation fits to the specification of a symbolic event right next to the front, the front moves - now including the respective symbolic event - and waits for the next observation. Thereby, constraints restrict the legal moves of the front. The possible shapes of the front are referred to as *phases* of a diagram. If an observation forces the front to move but there exists no legal move to a next phase, since this move would violate the specification of a symbolic event or a constraint, the diagram is violated. In order to weaken this interpretation, STDx provides also constructs to treat particular violations only as expected exceptions. In case of such expected exceptions, the diagram is prematurely exited in a state of acceptance.

Following the intuition of moving a front through a diagram, a TSA is constructed with the phases as states and transitions capturing the legal ordering of the phases according to the order of events along the waveforms and the restrictions imposed by constraints. The construction of a TSA is also referred to as *unwinding.* We will show in this section that for a well-defined subset

of Symbolic Timing Diagrams, invariance checking can be applied according to the theorems and conclusion of section 6.3.

STDx-diagrams are dedicated to be used either as assumptions or commitments, unless they have the type "general", which allows usage in both assumption- and commitment-declarations. The individual diagrams are instantiated using declarations, whereat several diagrams can be grouped by a declaration. The semantics of a STDx-specification is a direct function of the semantics of the individual diagrams instantiated by the set of declarations the specification refers to. This is deeply related to a particular methodology of system verification. The construction of a requirement specification in terms of several (conjunctive) specification clauses leads naturally to a high degree of modularization. The verification of a STDx-specification consists of the verification of all diagrams in the set.

The layout of the user interface of the STDx-specification-manager reflects the logical structure of STDx-specifications.

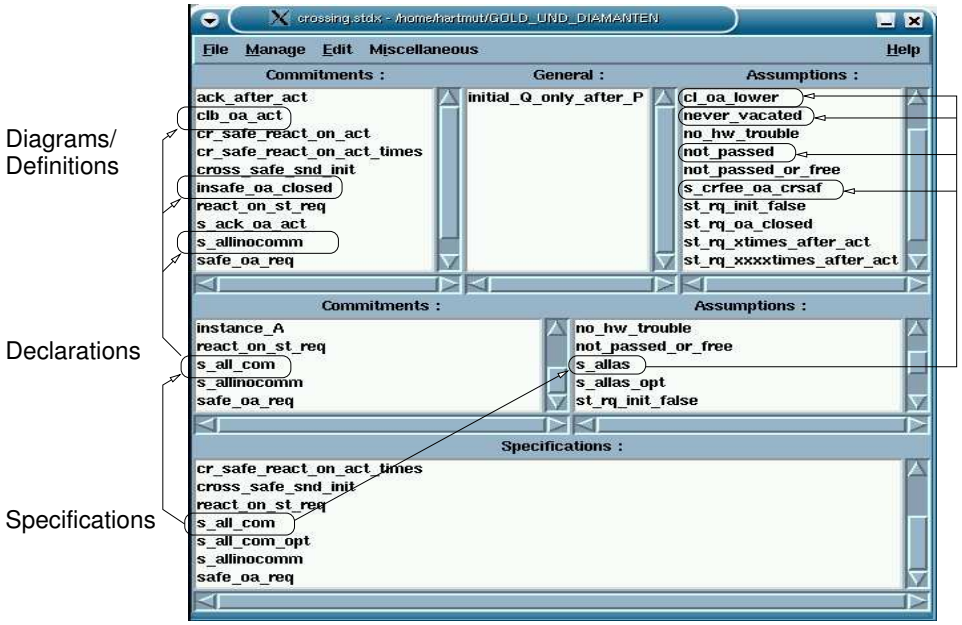


Figure 6.9: Timing Diagram Editor

The basic constructs of STDx-diagrams - symbolic events, waveforms, constraints, and activation concepts - are informally introduced and explained in detail in subsection 6.5.1. In order to formally define an interpretation according to the intuition of moving a front through the diagrams, subsection 6.5.2 provides formal definitions of specifications, declarations, diagrams and their building constructs. In subsection 6.5.3, preparation of STDx-specifications for the application of diagram translation into TSA is described. A formal description of this translation (unwinding into TSA) follows in subsection 6.5.4. The section is concluded with an overview of related work in subsection 6.5.5 .

6.5.1 Diagrams

Figure 6.10 shows an example diagram. The diagram consists of two waveforms, one for the interface object `ACTIVATE_CROSSING_SND_F` and the other one for `ACK_REC_F`. The two parallel vertical lines between waveform-names and the waveforms specify initial activation of the diagram, which will be described in more detail later in this section. If not initially `not(ACK_REC_F) and not(ACTIVATE_CROSSING_SND_F)` holds, then the diagram is violated. Otherwise, the diagram is fulfilled only if either `not(ACK_REC_F) and not(ACTIVATE_CROSSING_SND_F)` holds forever or if `ACK_REC_F` becomes true exactly 3 steps after `ACTIVATE_CROSSING_SND_F` has become true. Any other order of observations regarding `ACK_REC_F` and `ACTIVATE_CROSSING_SND_F` would violate the diagram, for example `ACK_REC_F` becoming true without previous observation of `ACTIVATE_CROSSING_SND_F` etc.

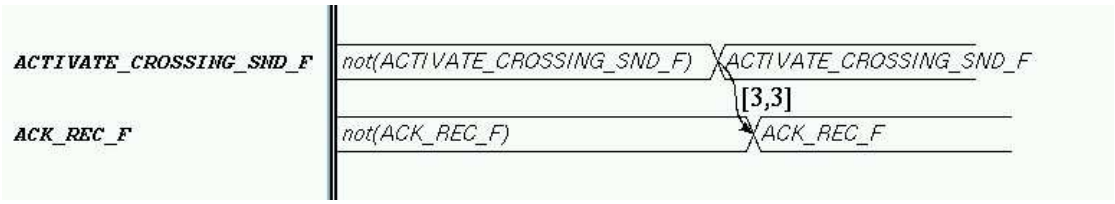


Figure 6.10: Symbolic event labeled with expressions

We will in the remainder of this subsection informally explain the graphical and textual constructs of which a diagram consists.

Symbolic Events and Waveforms

A symbolic waveform specifies a required (or expected) sequence of symbolic events, each of which defines a particular change of value of one or more observables to which the diagram refers. A single event is defined by three expressions: a *trigger* expression, an optional *stable* condition and an also optional *exit* condition. Their meaning is given by the unwinding interpretation: The interpretation of a diagram remains in a stable phase as long as all stable conditions of all events belonging to that phase are satisfied. The interpretation moves the front, if the trigger expression of a symbolic event right next to the front becomes true, i.e. the TSA constructed by unwinding will provide transitions for each combination of triggers that can legally become true for a particular phase. If neither the trigger nor the stable condition of one event are satisfied, the exit condition of the event determines whether the diagram is violated or canceled. Figure 6.11 shows an example event and the placement of the expressions.

Figure 6.11: Symbolic event labeled with expressions

Each waveform is identified by a name, which may be a symbolic name or the name of an interface object of the model to which the diagram refers. In the latter case, 'incomplete' assertions of the

form '`= value`' or '`/= value`' are allowed, which are completed using the name of the interface object as left-hand-side operand of the comparison operator.

Sometimes a waveform is ended with the trigger '`false`' for the last event. This means, that the last event can never occur, since no system state satisfies '`false`'. The intended meaning is, that the last phase before the final event is required to last 'forever'.

If stable and trigger condition of one event are non-exclusive, the event is a non-deterministic event. Nondeterministic events are graphically marked by an underlying grey rectangle.

Constraints

Without explicit constraints, no temporal relationship between events on different waveforms is specified. Different kinds of constraints can be used to require ordering or timing conditions among events:

Symmetric Constraints are used for specification of a distance between two events. They can be used to require simultaneity, to exclude simultaneity or to specify a concrete distance in terms of steps or super-steps. Only a distance is specified without requiring a particular order of the events referred to by the constraint.

Asymmetric Constraints : In contrast to symmetric constraints, asymmetric constraints relate a source-event to a target-event.

- **Precedence Constraints** specify an order of the events they refer to. In general, the order is of the form '*the target event may not be observed before (only after) the source event*'.
- **Leadsto Constraints** specify a causality relation regarding the referred events. Basically, leadsto constraints require that '*if the source event is observed, the target event must be observed eventually after activation of the diagram*'. Hence, leadsto constraints in their pure form express rather a temporal implication than an ordering of the concerned events.
- **Combined Constraints** are a combination of a precedence constraint portion and a leadsto constraint portion. They specify a causality relation as well as an ordering of the referenced events.

STDx-constraints are either *qualitative* or *quantitative* constraints. Qualitative constraints specify a principle temporal relation between two symbolic events, without requiring a concrete timing. This is denoted graphically by *symbolic* interval annotations of the constraint. In contrast, quantitative constraints are annotated with concrete time-intervals. For quantitative constraints it has to be distinguished whether the timing-interval refers to *steps* or to *super-steps* of the model. Graphically, quantitative step constraints are depicted by normal (thin) lines, while super-step constraints are depicted by bold lines.

Constraints are either *mandatory* or *possible*:

- Mandatory constraints express a requirement: it is an error if a run of the system violates a mandatory constraint. Mandatory constraints are graphically depicted by solid lines.
- In contrast to the mandatory interpretation, possible constraints specify expectations about the temporal relation of two events. If a possible constraint is violated by a run of the

system, the diagram is canceled, i.e. exited in a state of premature acceptance. For graphical distinction from mandatory constraints, possible constraints are drawn using dashed lines.

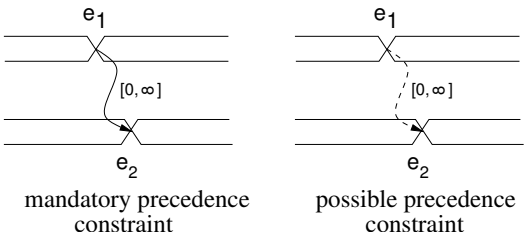


Figure 6.12: Mandatory and Possible Precedence Constraints

The concrete interpretation of possible and mandatory constraints depends also on the usage of the diagram:

Mandatory constraints in assumption diagrams are used to express requirements regarding the environment of the (sub-)system, whereas a possible constraint expresses expectations of the environment regarding the (sub-)system.

In contrast, mandatory constraints in commitment diagrams are used to express requirements regarding the (sub-)system, whereas possible constraints express expectations w.r.t. the environment.

Consequently, in particular for *asymmetric* constraints the usage of possible and mandatory constraints is logically restricted:

In assumptions, mandatory asymmetric constraints shall not refer to target events specifying output behavior of the (sub-) system, whereas possible asymmetric constraints shall not refer to target events specifying inputs from the environment.

The contrary should be adhered to by commitment diagrams: mandatory asymmetric constraints shall not refer to target events specifying inputs form the environment, whereas possible asymmetric constraints shall not refer to target events specifying output behavior of the (sub-) system.

Symmetric Constraints

Symmetric constraints (Figure 6.13) express a *symmetric* temporal relation between two events. A *simultaneous* constraint (symbolic interval $[0, 0]$) specifies that whenever event e_1 is observed, e_2 has to to be observed simultaneously and vice versa. Even though the interval annotation seems to denote a concrete interval, simultaneous constraints only specify 'real' simultaneity and thus belong to the qualitative constraints. Notice that there exists no super-step variant of simultaneous constraints.

In contrast to simultaneous constraints, *conflict* constraints (symbolic interval $(0, \infty]$) forbid the observation of both events at the same point in time. Again, the conflict constraint is a qualitative one and there exists no super-step variant.

In order to refer to steps or super-steps quantitatively, a *distance* constraint with a concrete timing-interval ($[n, m]$) can be used:

Whenever one of the two constrained events is observed, a distance constraint requires the other event to be observed within an temporal interval specified by a lower ($n \in \mathbb{N}_0$) and an upper bound ($m \in \mathbb{N}$) after the observation of the first event. Distance constraints exist in two variants, one referring to steps and the other one referring to super-steps of the model.

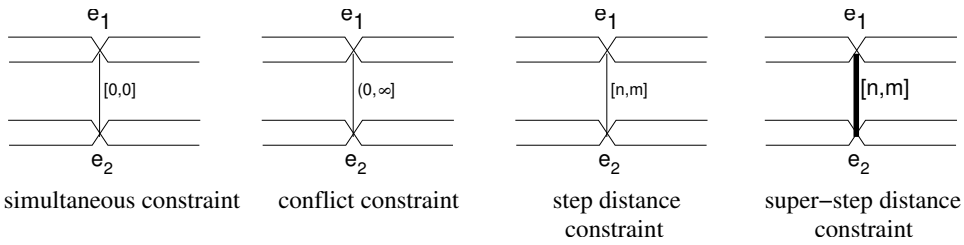


Figure 6.13: Symmetric Constraints

Precedence constraints

Precedence constraints are used to define an ordering of events. They do not require the target event to occur, but if the target event occurs it has to occur - depending on the interval annotation of the constraint - not before or strictly after the source event of the constraint.

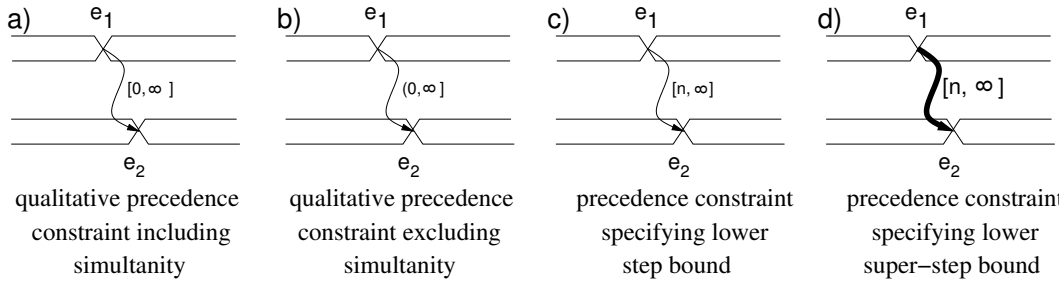


Figure 6.14: Precedence Constraints

Constraint (a) in figure 6.14 specifies event e_2 to be observed - if e_2 will be observed at all - not before observation of e_1. Depending on whether the lower bound 0 is (a) included in the symbolic interval or (b) excluded from the interval, e_2 may or may not happen simultaneously with e_1. Precedence constraints can also be used with a non-negative lower bound $n \in \mathbb{N}$ (as shown in (c) referring to steps and (d) referring to super-steps). Quantitative constraint (c) specifies that, if e_2 is observed, e_1 must have been observed at least n steps before e_2, whereas quantitative constraint (d) requires that e_1 must have been observed at least n super-steps before e_2 . Thus, the lower bound specifies a minimal distance of the target event from the source event, whereat the events have to be observed in the specified order. In order to emphasize the fact that e_2 is not required to be observed at all, other upper bounds than the ∞ symbol as *included upper bound* are not permitted for precedence constraints.

Leadsto constraints

Leadsto constraints require the target event to occur if the source event occurs - without restricting the lower bound of the interval. I.e. the target event may also occur before the source event.

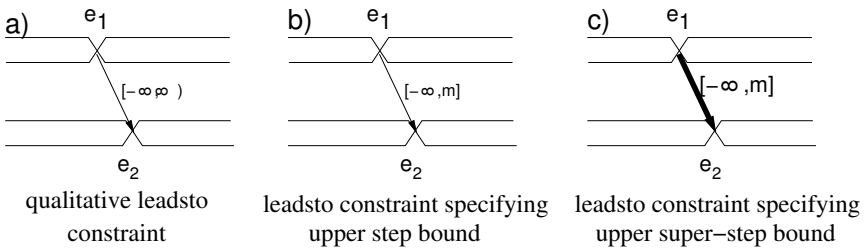


Figure 6.15: Leadsto Constraints

A quantitative leadsto constraint (figure 6.15 (a)) requires that if event e_1 is observed, event e_2 must be observed eventually after activation of the diagram. Leadsto constraints can also be specified with an quantitative upper bound, as depicted in 6.15 (b) referring to steps and (c) referring to super-steps. If event e_1 occurs, event e_2 must occur eventually after activation of the diagram but at most $m \in \mathbb{N}$ steps (super-steps) after e_1.

Combined constraints

Combined constraints combine the ordering properties of precedence and the causality portion of leadsto constraints. The precedence portion specifies the ordering of source and target event, while the leadsto portion requires the target event to occur. Examples of combined constraints are shown in figure 6.16. Again, combined constraints can be used with qualitative and quantitative interval annotations.

Qualitative combined constraints ((a) and (b)) : If event e_1 is observed, e_2 *must be observed eventually not before (after) observation of* e_1. If the lower bound of the constraint interval is 0, it can be (a) included in the interval or (b) excluded from the interval in order to accept or permit simultaneity.

Quantitative combined constraints ((c) and (d)) : combined constraints can also be used with a concrete quantitative interval $[n, m]$, where $n \in \mathbb{N}_0$and $m \in \mathbb{N}$. (c) specifies that if event e_1 is observed, e_2 has to be observed at least n steps but at most m steps after observation of e_1. In contrast, (d) refers to super-steps instead of steps.

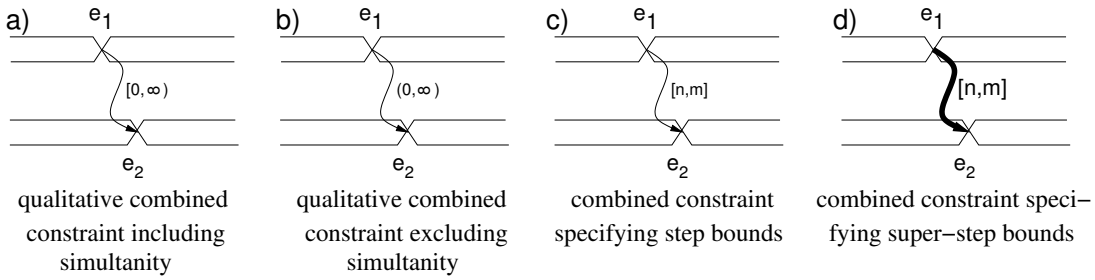


Figure 6.16: Combined Constraints

Activation modes

A system has to satisfy a requirement specified by a diagram whenever the diagram is activated. We distinguish two different activation modes: *initial* and *iterative*. An initial diagram is activated only once at system start. In contrast, iterative diagrams are activated again and again along a system run, whenever the activation condition evaluates to true and no other incarnation of the same diagram is already active. Only one instance of an iterative diagram can be activated at any instance of time (cf. figure 6.4 of section 6.3). The activation condition is built from the trigger expressions of the first (left most) events of all waveforms. The conjunction of all these expressions forms the starting state of the diagram. To further restrict the activation, an additional condition can be specified by an expression, which is then regarded in combination with the activation condition (explained below).

Regarding older version of Symbolic Timing Diagrams, it has been criticized by users that *invariant* activation in particular of assumption diagrams was counter-intuitive:

This particular problem of invariant activation has been referred to as 'self-activation', which means the following: while a diagram with invariant activation mode is activated and being matched by the behavior following the activation point, *further* activations (instances) of the same diagram might occur. These further instances are often violated by the runs satisfying the first instance. This is unexpected in almost all cases, since the intention of the designer has been to express a requirement only in the context of a situation where the start pulse has happened just before. The problem can be remedied using the new concept of iterative activation.

Iteratively activated diagrams first have to be completely worked off before a new instance can be activated. The variant of Symbolic Timing Diagrams presented in this thesis, hence only provides initial and iterative activation. Even though, invariant activation could technically be supported for commitments, we will not consider invariant activation in the context of this work.

Graphically, initial activation is depicted by a double vertical between the waveform names and the waveforms, while iterative activation is depicted by a single vertical line.

Activation Context and Activation Exception

As stated above, the activation condition of a diagram is build by conjunction of the trigger expressions of the left-most events of all waveforms of a diagram. Sometimes this may be not restrictive enough. STDx therefore offers the concept off an additional optional activation context. Instead of adding additional waveform(s) in order to further restrict the activation of a diagram, an arbitrary expression can be entered as activation context for iterative diagrams. If such an activation context is specified by the user for an iterative diagram, the diagram is activated only if the conjunction of activation condition plus the activation context expression evaluates to true (and no earlier instance of the diagram is currently active).

For initial diagrams instead of an activation context an activation exception can be specified. If the activation condition of an initial diagram evaluates to false in step 0, but a user defined activation exception becomes true, the diagram is prematurely satisfied without being activated at all. Activation exceptions can be used e.g. for initial case distinctions: By specifying assumptions with mutual exclusive activation exceptions different initializations of a system can be considered; by specifying commitments with mutual exclusive activation exceptions, alternative reactions dependent on different initial values can be specified.

6.5.2 Building STDx-specifications from Diagrams and Declarations

For the specification of reactive systems it is quite often necessary to assume some behavior of the environment. A correct behavior of the system or of a sub-system can only be guaranteed provided that the environment does not violate these assumptions. STDx-diagrams can be used not only to specify properties of a system but also to specify such assumptions about the environment. For this purpose, diagrams are determined to be either assumptions, commitments or general diagrams. The latter can be instantiated by commitment as well as by assumption declarations.

Based upon the basic diagrams, STDx-specifications are derived in two steps. Diagrams are instantiated in declarations. *Commitment declarations* can instantiate commitment or general diagrams, while *assumption declarations* can instantiate general or assumption diagrams. *Parameters* (see below) of the instantiated diagrams are mapped to concrete expressions by the declaration. A specification is built by choosing exactly one commitment declaration (instantiating arbitrary many diagrams) and arbitrary many assumption declarations.

The interpretation of a STDx-specification is that the commitment declaration has to be valid for the system under consideration provided that the assumptions are guaranteed by the environment.

For example STDx-specification `s_all_com` in figure 6.9 refers to the commitment declaration `s_all_com` and the assumption declaration `s_allas`. The commitment declaration `s_all_com` instantiates the commitment diagrams `clb_oa_act`, `insafe_oa_closed` and `s_allinocomm`. Assumption declaration `s_allas` instantiates the assumption diagrams `cl_oa_lower, never_vacated, not_passed, s_crfree_oa_crsaf,` and `strq_and_act_init` (not visible in figure 6.9). This specification will be considered is part of an application example for compositional verification and will be explained in detail in section 8.3.3.

According to the hierarchical organization of STDx-specifications we first formally define STDx-specifications in definition 6.27, because specifications define the scope of all instantiated diagrams. In definition 6.29, a formal definition is given for STDx-declarations, which instantiate the individual diagrams in the scope of STDx-specifications. After some additional formal definitions regarding waveforms and constraints, STDx-diagrams will be formally defined in definition 6.34.

Specification Variables

Three kinds of specification variables can be used in Symbolic Timing Diagrams: Last, rigid and flexible specification variables:

- *Last variables* represent the value of particular observables in the previous step. A last-variable is introduced for a specification by the declaration:

  $$< \texttt{new\_name} > = last(< \texttt{observable} >)$$

  where `new_name` is the name of the last-variable to be introduced and observable is an interface object of the interface referred to by the specification.

- *Rigid variables* can be assigned any value of their domain in step 0, but do not change their value thereafter. Rigid variables can for example be used as indices for arrays if a specification captures a property for all elements of an array.
  A rigid variable is introduced for a specification using the declaration

  $$< \texttt{new\_name} >: - < \texttt{type} >;$$

or alternatively

$$< \texttt{new\_name} >: -\texttt{type\_of}(< \texttt{observable} >);$$

where `type` is a data-type in the scope of the specification,and `new_name` is the name of the rigid variable which will be added to the specification. Using the alternative declaration, a rigid variable `new_name` of same data-type as `observable` is declared.

- *Flexible variables* can take different values of their domain in every step. The declaration of a flexible variable introduces a fresh input to the system, which can be referred to in the specification. For each flexible variable also a last variable is introduced, which contains the value of the variable in the last step. Flexible variables can for example realize counters in a specification. A flexible variable is introduced by the declaration

  $$< \texttt{new\_name} >: \sim < \texttt{type} >;$$

  or alternatively

  $$< \texttt{new\_name} >:\sim \texttt{type\_of}(< \texttt{observable} >);$$

  where - again - `type` is a data-type in the scope of the specification.

We will only make use of last-variables in the application examples of section 8.3. Examples for the application of rigid and flexible specification variables can be found in [SAC99] and [ABC+99].

Definition 6.27 (STDx-Specification)
A STDx-specification is a tuple $spec = (intf(A), name, ass\_decls, comm\_decl, specvars)$, where

- $intf(A)$ is the interface declaration of (sub-)system A, to which $spec$ refers (cf. section 5.4)
- $name$ is the name of the specification
- ass_decls is a possibly empty set of assumption declaration names. For each assumption declaration $adecl$ with $adecl.name \in ass\_decls$ it is required that $adecl.context = assumption$. Declarations will formally be defined in definition 6.29.
- $comm\_decl$ is a single commitment declaration name. For the referred commitment declaration $cdecl$ with $cdecl.name = comm\_decl$ it is required that $cdecl.context = commitment$.
- $specvars$ is a possibly empty set of specification variables.

If $specvars \neq \emptyset$, $spec$ refers to an interface of A which is extended with $specvars$. This interface extension of $intf(A)$ has to be provided by $spec$ for the scope of all diagrams referred to by $spec$. Let $intf_{\circledS}(A) := intf(A) \cup specvars$ denote the extension[15] of $intf(A)$ with $specvars$. ■

[15] Technically, $intf(A)$ is *injected* with $specvars$, and the model representation of (sub-)system A is extended with new variables according to the declarations of $specvars$. An injector-tool is integrated with the STVE, which

1. for *last variables* extends the model with a new variable which always keeps the last value of the variable to which the declaration of the *last variable* refers
2. for *rigid variables* extends the model with a *fresh input* as well as *a new variable*, which is initialized by the input in step 0, and keeps this initial value for the remaining computation
3. for *flexible variables* extends the model with a *fresh input*, which can change its value at every step *plus a last variable,* which keeps the last value of this input in every step

Templates using Formal Parameters

In order to enable reuse of diagrams, parameters can be used instead of concrete expressions. These parameters are mapped to concrete expressions when instantiating a diagram in declarations, permitting the usage of one diagram in several contexts. This way, different specifications can be derived from one diagram. Figure 6.17 shows an example of an `initial_Q_onlyafter_P` template.

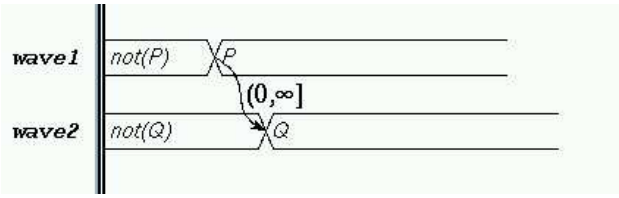


Figure 6.17: An Initial Only-After Diagram Template

In a declaration instantiating the template of figure 6.17 in a concrete context, the formal parameters `P` and `Q` are bound to concrete expressions referring to observables of the interface: For example, the mapping:

```
P => ACTIVATE_CROSSING_REC_F ;
Q => ACK_SND_F ;
```

instantiates the diagram such that initially `ACK_SND_F` is specified to be observed only after observation of `ACTIVATE_CROSSING_REC_F`.

Some temporal schemes are often used for the specification of system properties. By using diagrams with parameters instead of concrete expressions, the diagram has to be drawn only once and can be reused several times. Consequently, a set of often used diagrams can be imported from a predefined library.

In order to define valid annotations for symbolic events of diagrams formally, we have to extend the definition of predicates[16] to predicates that also permit the usage of formal parameters and specification variables.

Definition 6.28 (Parametrized Predicates)

Given a set pd of parameter names and a set of variables $\mathcal{V}_{\textcircled{S}}$. Then, predicates $\beta \in PPred_{\mathcal{V}_{\textcircled{S}} \cup pd}$ ranging over $\mathcal{V}_{\textcircled{S}}$ and pd are built according to :

$$\beta := \alpha \,|\, true \,|\, \neg\beta \,|\, (\beta) \,|\, \beta_1 \wedge \beta_2 \text{ , where}$$

α is a parameter name or an atomic proposition w.r.t. $\mathcal{V}_{\textcircled{S}}$. Lee $false{:=}\neg true$ and $\beta_1 \vee \beta_2{:=}\neg(\neg\beta_1 \wedge \neg\beta_2)$ be abbreviations as usual. In order to support expressions in a more STATEMATE-like notation, instead of $\neg$ also `not`, `and` instead of $\wedge$,and `or` instead of $\vee$can be used in event annotations. ■

Definition 6.29 (STDx-Declaration)

A STDx-declaration is a tuple $decl{:=}(name, context, diagrams, pm)$, where

- *name* is the name of the declaration

[16] Recall the definition of $Pred_{\mathcal{V}}$ from definition 6.2 of section 6.3

- $context \in \{assumption, commitment\}$ regulates the permitted utilization of $Decl$ in specifications
- $diagrams$ is a set of Symbolic Timing Diagrams names. If $context = assumption$ then for each diagram d with $d.name \in diagrams$ it is required that $d.context \in \{general, assumption\}$. If $context = commitment$ then for each diagram d with $d.name \in diagrams$ it is required that $d.context \in \{general, commitment\}$. Diagrams will formally be defined in definition 6.34.
- For a set pd of parameter names, $pm \subseteq pd \times Pred_{intf_{\circledS}(A)}$ is a mapping of the parameters of the diagrams $d \in diagrams$ to expressions ranging over the set of variables $\mathcal{V}_{\circledS}$ in $intf_{\circledS}(A)$. Let PM denote the set of possible replacement mappings.

■

In order to refer to the assumption and commitment diagrams instantiated by a STDx-specification via reference to declarations, we introduce for simplicity the following sets:

Definition 6.30 (Diagrams of a STDx-specification)
For a given STDx-specification $spec = (intf(A), name, ass\_decls, comm\_decl, specvars)$, let $ad(spec)$ denote the set of diagrams instantiated by the STDx-declarations a with $a.name \in ass\_decls$:

$$\begin{aligned} ad(spec) \quad := \quad & \{diagram \quad | \quad \exists a \in declarations : a.name \in ass\_decls \\ & \wedge diagram.name \in a.diagrams\} \end{aligned}$$

Accordingly, let $cd(spec)$ denote the set of diagrams instantiated by the STDx-declaration referred to by $comm\_decl$:

$$\begin{aligned} cd(spec) \quad := \quad & \{diagram \quad | \quad \exists c \in declarations : c.name = comm\_decl \\ & \wedge diagram.name \in c.diagrams\} \end{aligned}$$

■

Before we can define Symbolic Timing Diagrams as the core of the formalism, we first have to introduce waveforms and constraints formally:

Definition 6.31 (Symbolic Waveform)
A symbolic waveform is a tuple $wvf{:=}(name, SE_{wvf}, \rightarrow_{wvf}, stable, trigger, exit, det)$ w.r.t. interface declaration $intf_{\circledS}(A)$ of system A and a possibly empty set of parameter names pd, where

- $\mathcal{V}_{\circledS}$ is the set of variables in interface $intf_{\circledS}(A)$
- $name$ is the name of the waveform. $name$ can either be the name of an interface object from $intf(A)$ or a symbolic name.
- SE_{wvf} is the set of symbolic events of waveform wvf.

- $\rightarrow_{wvf}$: $SE_{wvf} \rightarrow SE_{wvf} \cup \{\top_{wvf}\}$ is a successor function, which associates each event $e \in SE_{wvf}$ of waveform wvf with a successor event $e' \in SE_{wvf} \cup \{\top_{wvf}\}$, whereat $\top_{wvf}$ (top-event) is an auxiliary construct for the unwinding algorithm, such that the last event of wvf has the successor $\top_{wvf}$, while $\top_{wvf}$ itself has no successor. $\rightarrow_{wvf}$ defines a total order on the events $SE_{wvf} \cup \{\top_{wvf}\}$.
  Let the reflexive and transitive closure of $\rightarrow_{wvf}$ be denoted by $\leq_{wvf}$.

- $stable : SE_{wvf} \cup \{\top\} \rightarrow (PPred_{\mathcal{V}_{(\mathbb{S})} \cup pd} \cup \{\varepsilon\})$,
  $trigger : SE_{wvf} \cup \{\top\} \rightarrow PPred_{\mathcal{V}_{(\mathbb{S})} \cup pd}$, and
  $exit : SE_{wvf} \cup \{\top\} \rightarrow (PPred_{\mathcal{V}_{(\mathbb{S})} \cup pd} \cup \{\varepsilon\})$ are mappings which associate a symbolic event with a *stable* condition, a *trigger* condition and an *exit* condition, respectively. Since *stable* and *exit* conditions can be left unspecified by the user, both conditions can be mapped to the empty predicate ε. In this case a meaningful default has to be chosen for the respective condition before applying the unwinding algorithm (defaults will be defined in definition 6.36)
  The associated conditions are predicates ranging over the objects of $intf_{(\mathbb{S})}(A)$. In particular, let

  - $stable(\top) := true$,
  - $trigger(\top) := false$
  - $exit(\top) := false$

- $det : SE_{wvf} \cup \{\top\} \rightarrow \{deterministic, non\_deterministic\}$ is an attribute, which determines whether an event will be treated deterministically or non-deterministically by the unwinding algorithm, i.e. whether *stable* and *trigger* conditions are to be treated to be mutually exclusive or if they can both be true at the same time. Let $det(\top) := deterministic$

Let $Waveforms_{\mathcal{V}_{(\mathbb{S})}}$ denote the set of waveforms w.r.t $intf_{(\mathbb{S})}(A)$

■

In general, a Symbolic Timing Diagram consists of more than only one waveform:

Definition 6.32 (Bundle of waveforms)
A bundle of waveforms BW over a set of variables $\mathcal{V}$ is a set of waveforms, such that their sets of symbolic events are mutually disjoint. Given waveforms wvf_i, wvf_j:
$wvf_i, wvf_j \in BW, i \neq j \Rightarrow SE_{wvf_i} \cap SE_{wvf_j} = \emptyset$
The set of events SE_{BW} belonging to a bundle of waveforms BW is defined by the union of the sets of events of the waveforms belonging to bundle BW:

$$SE_{BW} := \bigcup_{wvf \in BW} (SE_{wvf} \cup \{\top_{wvf}\})$$

■

While the symbolic events SE_{wvf} along a single waveform wvf are totally ordered according to $\rightarrow_{wvf}$, no order is defined among symbolic events of different waveforms. Constraints can be used, in order to introduce an ordering:

Definition 6.33 (Constraints)
For a bundle of waveforms BW, let the set of feasible constraints $Constr$ be defined by:
$Constr \subseteq type \times class \times SE_{BW} \times scale \times \mathbf{T} \times SE_{BW}$, where

- $type \in \{distance, precedence, leadsto, combined\}$
- $class \in \{possible, mandatory\}$
  Possible constraints express expectations, while mandatory constraints express requirements about order or timing of the constrained events.
- $scale \in \{step, superstep\}$
  $scale$ determines to which concept of time the constraint refers. If $scale = step$, then the constraint refers to steps - *quantitative* constraints with $scale = step$ hence refer to series of δ-delays. If $scale = superstep$, then the constraint refers to the virtual model-time, which assumes that steps are performed with δ-delays and time advances only after the model has reached a *stable status*.
  For *qualitative* constraints, only $scale = step$ is supported.
- $\mathbf{T}$ is an interval, which can either be symbolic or specify concrete lower of upper bounds for the specified temporal relation of the regarded source and target events. according to the rules described informally in section 6.5.1. Table 6.2 lists the legal interval annotations of constraints.
  In general, qualitative constraints are supported only in combination with $scale = step$.

Constraint type	Legal Intervals	
precedence	qualitative: $[0, \infty], (0, \infty]$ quantitative: $[n, \infty], n \in \mathbb{N}$	
combined	qualitative: $[0, \infty), (0, \infty)$ quantitative: $[n, m], n \in \mathbb{N}_0, m \in \mathbb{N}, n \leq m$	
leadsto	qualitative: $[-\infty, \infty)$ quantitative: $[-\infty, n], n \in \mathbb{N}$	
distance	qualitative: $[0, 0]$ qualitative: $(0, \infty]$ quantitative: $[n, m], \quad n \in \mathbb{N}_0, m \in \mathbb{N}, n \leq m$ $[n, \infty] \qquad n \in \mathbb{N}$	'simultaneous' 'conflict' 'separation'

Table 6.2: Legal Interval Annotation of Constraints

■

We will use the don't care symbols '-' in the following if the value of an element or part of an element of the tuple is irrelevant. For example, let $(precedence, -, e_1, \{-, \infty], e_2)$ denote a *possible* or *mandatory* precedence constraint with source event e_1 and target event e_2. Interval $\{-, \infty]$ denotes an arbitrary interval $[n, \infty]$, with $n \in \mathbb{N}_0$ or $(0, \infty]$.

Notice, that *distance* constraints are symmetric, no order of source and target event, but only a temporal distance between them, is specified. Thus, $(distance, -, e_1, -, e_2)$ is equivalent to $(distance, -, e_2, -, e_1)$.

Using the above definitions, Symbolic Timing Diagrams can now formally be defined:

Definition 6.34 (Symbolic Timing Diagrams)
A Symbolic Timing Diagram for the interface $intf(A)$ of system A is a tuple $TD{:=}(intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, pd)$, where

- $intf_{\circledS}(A)$ is the interface of system A, with which the diagram is associated. Let $\mathcal{V}_{\circledS}$ denote the variables of $intf_{\circledS}(A)$
- $name$ is the name of the diagram
- $context \in \{assumption, general, commitment\}$ determines the permitted utilization of TD in STDx-specifications
- $actmode \in \{initial, iterative\}$ is the activation mode of the diagram,
- BW is a bundle of waveforms over interface $intf_{\circledS}(A)$,
- $e_{act} \in (PPred_{\mathcal{V}_{\circledS} \cup pd})$ is the activation condition of the diagram: $e_{act}{:=}\bigwedge_{wvf \in BW \forall e \in SE_{wvf}, \nexists e' : e' \rightarrow_{wvf} e} trigger(e)$ (The activation condition of TD is the conjunction of the triggers of all left-most symbolic events of all waveforms)
- $e_{ac} \in (PPred_{\mathcal{V}_{\circledS} \cup pd} \cup \{\varepsilon\})$ is the optional activation context of an iterative diagram
- $e_{ae} \in (PPred_{\mathcal{V}_{\circledS} \cup pd} \cup \{\varepsilon\})$ is the optional activation exception of an initial diagram
- $constr$ is a set of constraints for the bundle of waveforms BW
- pd is a possibly empty set set of parameter names.

■

Since the language STDx has been designed for capturing specification requirements in an intuitive and comfortable manner, there are some constructs aiming at comfort, which have to be instantiated and normalized before unwinding can be applied. Optional constructs, such as exit and stable conditions of symbolic events can be left unspecified by the user. Before applying the unwinding algorithm, a preprocessing has to provide useful defaults for these unspecified constructs. Furthermore, some high level constructs, such as parameters of diagrams have to be instantiated according to the parameter mappings provided by the instantiating declarations.

6.5.3 Preparation of STDx-Specifications for Application of Unwinding

As stated above, combined constraints represent combinations of precedence and leadsto constraints. Thus, in order to keep unwinding as simple as possible, each combined constraint is split up into a precedence constraint representing the ordering portion and a leadsto constraint representing the respective timing requirement portion of the combined constraint. Consequently, the unwinding algorithm only has to regard precedence and leadsto constraints as well as the symmetric constraints.

Preparation substitutes formal parameters of a diagram template by the actual expressions as provided by the instantiating declaration. Also the usage in assumption or commitment place is

fixed by instantiation before applying unwinding. Even though a diagram can be defined to be used 'general' (in assumption-declarations as well as in commitment-declarations) the concrete STDx-specification hierarchy referring to declarations instead of referring to diagrams directly restricts the usage of diagrams to either assumption or commitment.

Stable and exit conditions can be left unspecified by the user, in which case defaults have to be used: If the stable condition of an event is not explicitly specified, the condition is set to the trigger condition of the previous event of the waveform.

The choice of a default for exit conditions depends on the waveform as well as on the kind of the diagram. If a waveform of a commitment is associated with an input of the system, exit conditions default to true. The contrary default is chosen for assumption diagrams: for waveforms associated with inputs, unspecified exit conditions are set to false by default. If the waveform is associated with an output of the system or has only a symbolic name[17], unspecified exit conditions are set to false for commitment diagrams and to true for assumption diagrams. These default rules have the effect that assumptions are treated as restrictive as possible: If an input does not conform to an assumption, the assumption is violated. Since assumptions must not restrict outputs of the system, an output not conforming to an assumption diagram prematurely exits the assumption in an accepting state. On the other hand, if an input does not conform to a commitment, the commitment is prematurely exited by default. In contrast, an output not conforming to a commitment is treated as a violation.

Activation exceptions of initial diagrams are assigned the default '`false`' if they were left unspecified by the user, whereas by default for the activation context of an iterative diagram always '`true`' is chosen.

We formalize these preparations with the following definitions.

Definition 6.35 (Parameter Substitution)

Given a set of variables $\mathcal{V}$ and a possibly empty set of parameter names pd, as well as a parameter mapping $pm \subseteq pd \times Pred_{\mathcal{V}_{\textcircled{S}}}$

Let $psubst : PPred_{\mathcal{V}_{\textcircled{S}} \cup pd} \times 2^{pm \times Pred_{\mathcal{V}_{\textcircled{S}}}} \rightarrow Pred_{\mathcal{V}_{\textcircled{S}}}$ denote the substitution of parameter names in $\beta \in PPred_{\mathcal{V}_{\textcircled{S}} \cup pd}$ by predicates according to pm.

■

Definition 6.36 (Normalization of Waveforms)

For a set of waveforms $Waveforms_{\mathcal{V}_{\textcircled{S}}}$ w.r.t. interface $intf_{\textcircled{S}}(A)$, and a parameter mapping $pm \in PM$ we define two transformations

$norm\_ass\_wvf : Waveforms_{\mathcal{V}_{\textcircled{S}}} \times PM \rightarrow Waveforms_{\mathcal{V}_{\textcircled{S}}}$ and

$norm\_comm\_wvf : Waveforms_{\mathcal{V}_{\textcircled{S}}} \times PM \rightarrow Waveforms_{\mathcal{V}_{\textcircled{S}}}$.

Given a waveform $wvf = (name, SE_{wvf}, \rightarrow_{wvf}, stable, trigger, exit, det)$, let

- $wvf' := norm\_ass\_wvf(wvf)$, s.t.
  $wvf' := (name, SE_{wvf}, \rightarrow_{wvf}, stable', trigger, exit', det)$, where
  $name, SE_{wvf}, \rightarrow_{wvf}, trigger$, and det are the same as in wvf, and $\forall e \in SE_{wvf}$

[17] In this case, the waveform is treated like an output waveform.

$$stable'(e) := \begin{cases} false & \text{if } stable(e) = \varepsilon \wedge \nexists e' \in SE_{wvf'} : e' \rightarrow_{wvf} e \\ stable'(e') & \text{if } stable(e) = \varepsilon \wedge e' \in SE_{wvf'} \wedge e' \rightarrow_{wvf} e \\ psubst(stable(e), pm) & \text{otherwise} \end{cases}, \quad (6.18)$$

and

$$exit'(e) := \begin{cases} false & \text{if } (exit(e) = \varepsilon) \wedge \\ & (\exists p \in intf_{\circledS}(A) : name = name(p) \wedge mode(p) = in) \\ true & \text{if } (exit(e) = \varepsilon) \wedge \\ & \Big((\exists p \in intf_{\circledS}(A) : name = name(p) \wedge mode(p) = out) \\ & \vee (\nexists p \in intf_{\circledS}(A) : name = name(p))\Big) \\ psubst(exit(e), pm) & \text{otherwise} \end{cases} \quad (6.19)$$

- $wvf' := norm\_comm\_wvf(wvf)$, s.t.
  $wvf' := (name, SE_{wvf}, \rightarrow_{wvf}, stable', trigger, exit', det)$, where
  $name, SE_{wvf}, \rightarrow_{wvf}$, $trigger$, and det are the same as in wvf, and $\forall e \in SE_{wvf}$ $stable(e)$ is defined as in (6.18), and

$$exit'(e) := \begin{cases} true & \text{if } (exit(e) = \varepsilon) \wedge \\ & (\exists p \in intf_{\circledS}(A) : name = name(p) \wedge mode(p) = in) \\ false & \text{if } (exit(e) = \varepsilon) \wedge \\ & \Big((\exists p \in intf_{\circledS}(A) : name = name(p) \wedge mode(p) = out) \\ & \vee (\nexists p \in intf_{\circledS}(A) : name = name(p))\Big) \\ psubst(exit(e), pm) & \text{otherwise} \end{cases} \quad (6.20)$$

■

The following definition defines formally the split-up of combined constraints into a precedence and a leadsto portion. Additionally, for all symmetric constraints also the equivalent constraint is added to the set of constraints in order emphasize the symmetric nature of these constraints.

Definition 6.37 (Normalization of Constraints)
Given a set of constraints $constr \subseteq Constr$ for a bundle of waveforms BW. Let $norm\_constr : 2^{Constr} \rightarrow 2^{Constr}$ be a mapping of the set of constraints with
$constr' := norm\_constr(constr)$, s.t.

1. Each precedence and leadsto constraint $c \in constr$ is also in $constr'$:
   $\Big((c = (precedence, -, -, -, -, -) \in constr)$
   $\vee (c = (leadsto, -, -, -, -, -) \in constr)\Big) \quad \Rightarrow (c \in constr')$

2. For symmetric constraints $c \in constr : \; c \in constr'$ as well as the symmetric equivalent constraint c' are in $constr'$:

   a) $(c = (distance, mp, e_1, tm, [n, m], e_2) \in constr)$
   $\Rightarrow \Big((c \in constr') \wedge (c' = (distance, mp, e_2, tm, [n, m], e_1) \in constr')\Big)$,
   where $n \in \mathbb{N}_0, m \in \mathbb{N}$, $mp \in \{mandatory, possible\}$ and $tm \in \{step, superstep\}$

   b) $(c = (distance, mp, e_1, step, (0, \infty], e_2) \in constr)$
   $\Rightarrow \Big((c \in constr') \wedge (c' = (distance, mp, e_2, step, (0, \infty], e_1) \in constr')\Big)$,
   where $mp \in \{mandatory, possible\}$, and

   c) $(c = (distance, mp, e_1, step, [0, 0], e_2) \in constr)$
   $\Rightarrow \Big((c \in constr') \wedge (c' = (distance, mp, e_2, step, [0, 0], e_1) \in constr')\Big)$,
   where $mp \in \{mandatory, possible\}$.

3. Combined constraints are split-up into a precedence and a leadsto portion:
   $(c = (combined, mp, e_1, tm, [n, m], e_2) \in constr)$
   $\Rightarrow \Big((c' = (precedence, mp, e_1, tm, [n, \infty], e_2) \in constr')$
   $\wedge (c'' = (leadsto, mp, e_1, tm, [-\infty, m], e_2) \in constr')\Big)$,
   where $n \in \mathbb{N}_0, m \in \mathbb{N}$, $mp \in \{mandatory, possible\}$ and $tm \in \{step, superstep\}$

■

Now, we can collect the preparations and normalizations applied to diagrams before unwinding them into TSA in a single definition:

Definition 6.38 (Instantiation of STDx diagrams)
Given a diagram $d{:=}(intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, pd)$ and a parameter mapping pm. Let $inst\_ass\_td$ and $inst\_comm\_td$ be transformations on d, s.t.

- $d' = inst\_ass\_td(d, pm)$, with
  $d'{:=}(intf_{\circledS}(A), name', context', actmode, BW', e_{act}, e'_{ac}, e'_{ae}, constr', pd')$, where
  - $name'$ is an access-path containing the name of the specification and of the declaration instantiating d, as well as the name of d.
  - $intf_{\circledS}(A), actmode, e_{act}$ are the same as in d
  - $context'{:=}assumption$
  - $BW'{:=}\{wvf' | \exists wvf \in BW : wvf' = norm\_ass\_wvf(wvf, pm)\}$
  - $e'_{ac}{:=}\begin{cases} psubst(e_{ac}, pm) & \text{if } e_{ac} \neq \varepsilon \\ true & \text{otherwise} \end{cases}$
  - $e'_{ae}{:=}\begin{cases} psubst(e_{ae}, pm) & \text{if } e_{ae} \neq \varepsilon \wedge (actmode = initial) \\ false & \text{otherwise} \end{cases}$
  - $constr'{:=}norm\_constr(constr)$
  - $pd'{:=}\emptyset$ (Parameters have been instantiated according to pm)

- $d' = inst\_comm\_td(d, pm)$, with
  d':=$(intf_{\circledS}(A), name', context', actmode, BW', e_{act}, e'_{ac}, e'_{ae}, constr', pd')$, where
  - $intf_{\circledS}(A)$, $name$, $actmode$, e_{act}, $e_{ac'}$, $e_{ae'}$,$constr'$, pd' are defined the same way as for $inst\_ass\_td$.
  - $context'$:=$commitment$
  - BW':=$\{wvf' | \exists wvf \in BW : wvf' = norm\_comm\_wvf(wvf, pm)\}$

■

In an instantiated diagram, all predicates belonging to a symbolic event are either specified by the user or set to a default according to definition 6.36. For instantiated diagrams determinism of waveforms can formally defined by:

Definition 6.39 (Deterministic Waveform)
A waveform wvf of an instantiated diagram is *deterministic*, iff

- $\forall e \in SE_{wvf} : (trigger(e) \Rightarrow \neg stable(e))$
- $\forall e \in SE_{wvf} : (trigger(e) \Rightarrow \neg exit(e))$

■

Definition 6.40 (Instantiation of STDx Declarations)
Given a STDx-declaration $decl = (name, context, diagrams, pm)$. Let $inst\_ass\_decl$ and $inst\_comm\_decl$ be transformations, s.t.

- $decl'$:=$inst\_ass\_decl(decl)$, with $decl' = (name', context, diagrams', \emptyset)$, where
  $name'$ is an access-path consisting of the name of the instantiating specification and the name of $decl$
  $diagrams'$ refers to the instantiated diagrams
  $\{d' | \exists d \in diagrams : d.name \in decl.diagrams \wedge d' = inst\_ass\_td(d, pm)\}$
  Since the parameter mappings have been applied to the instantiated diagrams, the instantiated declaration $decl'$ defines no parameter mappings.
- $decl'$:=$inst\_comm\_decl(decl)$, with $decl' = (name', context, diagrams', \emptyset)$, where
  $name'$ is an access-path consisting of the name of the instantiating specification and the name of $decl$
  $diagrams'$ refers to the instantiated diagrams
  $\{d' | \exists d \in diagrams : d.name \in decl.diagrams \wedge d' = inst\_comm\_td(d, pm)\}$

■

Definition 6.41 (Instantiation of STDx-Specifications)
Given a STDx-specification $spec = (intf(A), name, ass\_decls, comm\_decl, specvars)$.
Let $inst\_spec$ be a transformation of $spec$, s.t.
$spec'$:=$inst\_spec(spec)$, with $spec' = (intf_{\circledS}(A), name, ass\_decls', comm\_decls', specvars)$, where
$intf_{\circledS}(A)$ refers to the interface $intf(A)$ extended with $specvars$ according to footnote 15 on page 159.
$name, specvars$ are the same as in $spec$, and

- ass_decls' refers to the instantiated declarations
  $\{decl' | \exists decl \in declarations \wedge decl.name \in spec.ass\_decl \wedge decl' = inst\_ass\_decl(decl)\}$
- $comm\_decl'$ refers to the instantiated declaration $decl'$for which
  $\exists decl \in declaration \wedge decl.name = comm\_decl \wedge decl' = inst\_comm\_decl(decl)$

■

In the remainder of this work, we consider only instantiated STDx-specifications. Hence, all diagrams are assumed to be instantiated according to the above definitions.

6.5.4 Unwinding of Symbolic Timing Diagrams

In this subsection we present an unwinding algorithm which generates TSA representations for instantiated diagrams. By vitually moving a frontier from left to right through the diagram according to the ordering induced by the waveforms and constraints the unwinding automaton is generated step by step. The unwinding algorithm starts with an identification of classes of events which have to happen simultaneously or which have to happen in a strict order. Each set of events that have to happen simultaneously represents a valid *phase* of unwinding the diagram. Unordered events or sets of events have to be represented by alternative interleavings in the automaton.

The unwinding algorithm only considers the core diagram. Activation mode *actmode* and activation condition e_{act} as well as the optional activation context e_{ac} and activation exception e_{ae} are not regarded by the algorithm. They come into play again later when providing the resulting automaton with activation control.

We start the description of the algorithm with the definition of a phase:

Definition 6.42 (Phases)
A set $\zeta \subseteq SE_{BW}$ which contains exactly one symbolic event of each waveform is called a *phase* of the bundle of waveforms BW:

$$Phases_{BW} := \left\{ \zeta \subseteq SE_{BW} \cup \{\top_{wvf}\} | \forall wvf \in BW \exists^1 e \in \zeta : e \in SE_{wvf} \cup \{\top_{wvf}\} \right\}$$

■

Definition 6.43 (Partial Order of Phases) The successor functions $\rightarrow_{wvf}$ on the individual waveforms induce a *partial order relation* on BW:

$$\leq_{BW} \subseteq Phases_{BW} \times Phases_{BW}, \text{ with}$$

$$\zeta_i \leq_{BW} \zeta_j, \text{ iff } \forall e_i \in \zeta_i \forall e_j \in \zeta_j \ : \\ (\exists wvf \in BW : e_i \in SE_{wvf} \cup \{\top_{wvf}\}, e_j \in SE_{wvf} \cup \{\top_{wvf}\}) \Rightarrow (e_i \leq_{wvf} e_j)$$

Minimal and maximal elements of the bundle regarding $\leq_{BW}$ are defined straightforwardly:

$$\zeta_0 := \bigcup_{wvf \in BW} \{e_0 \in SE_{wvf} | e_0 \text{ is minimal w.r.t. } \leq_{wvf}\}$$

and

$$\top_{BW} := \bigcup_{wvf \in BW} \{\top_{wvf}\}$$

.

■

In contrast to the individual successor functions $\rightarrow_{wvf}$, a phase ζ may have different alternative successors according to the partial order relation $\leq_{BW}$. Hence, alternative interleavings have to be considered for possible successor phases according to $\leq_{BW}$. The unwinding algorithm for Symbolic Timing Diagrams is based on this definition of partially ordered phases.

Definition 6.44 (Successors of a Phase)
A phase ζ divides the set SE_{BW} into the two disjoint subsets:

- $future(\zeta) := \{e' \in SE_{BW} \mid e' \notin \zeta \wedge \exists wvf \in BW : \exists e \in \zeta : e \leq_{wvf} e'\}$ (all events located right from phase ζ)
- $past(\zeta) := SE_{BW} \backslash future(\zeta)$ (all events located left from phase ζ - and including ζ)

We write $\zeta_1 \rightarrow_\zeta \zeta_2$ iff :

$$\zeta_1 \neq \zeta_2 \wedge \zeta_1 \leq_{BW} \zeta_2$$

and

$$\begin{aligned} &\forall e_i \in \zeta_1 \forall e_j \in \zeta_2 \quad : \quad (\exists wvf \in BW : e_i, e_j \in SE_{wvf} \cup \{\top_{wvf}\}) \\ &\Rightarrow (e_i \leq e_j \wedge \nexists e_k \in SE_{wvf} : e_i \leq_{wvf} e_k \leq_{wvf} e_j) \vee (e_i = e_k = e_j)) \end{aligned}$$

(if $e_i \in \zeta_1$ and $e_j \in \zeta_2$ are symbolic events of the same waveform, then either $e_i = e_j$ or $e_i \rightarrow_{wvf} e_j$)

■

The successor relation $\rightarrow_\zeta$ on the set of phases defines for a phase the set of possible direct successor phases. In general, no order is determined for two phases $\zeta_i \neq \zeta_j$ with same predecessor ζ_k regarding $\rightarrow_\zeta$ unless constraints define an additional ordering of ζ_i and ζ_j on the set of phases. $\rightarrow_\zeta$ only determines possible moves of the front, while constraints further restrict these possible moves or specify time bounds for permitted moves.

Definition 6.45 (Unwinding-TSA of a Diagram)
The Timed Symbolic Automaton (cf. definition 6.6)

$$TSA(TD) = (\mathcal{V}_{\circledS}, S, s_0, C_{step} \cup C_\tau, T, F)$$

for a symbolic timing diagram

$$TD = (intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, \emptyset)$$

consists of:

1. $S \subseteq Phases_{BW} \cup \{\zeta_{exit}\}$. The set of states correspond to the *phases* of TD plus an unique exit state, which is entered on violation of *possible* constraints and by exit conditions of symbolic events. Since ζ_{exit} is a newly introduced state, ζ_{exit} is not affected by any constraint.
2. $s_0 = \zeta_0$, where ζ_0 is the minimal phase w.r.t $\leq_{BW}$ according to definition 6.43

3. $\mathcal{V}_{\circledS} \subseteq intf_{\circledS}(A)$ is the set of variables in interface $intf_{\circledS}(A)$ of system A, with which TD is associated.

4. C_{step} is a set of clocks referring to model steps. C_τ is a set of clocks referring to super-steps of the model.
   A clock $z_e \in C_{step}$ as well as a clock $zz_e \in C_\tau$ is introduced - and is associated with - in advance for each symbolic event $e \in SE_{BW}$; Most of these clocks will later be eliminated according to TSA normalization, because they will not be referred to in any timing constraint of $TSA(TD)$.

5. The construction of transition relation T is the major task of the unwinding algorithm. The rules for construction of T will be explained below (in definition 6.48).

6. The set F of fair states is derived from the *leadsto* and *distance* constraints[18]. No progress is enforced if no target event of a *mandatory leadsto* or *'separation (distance) with upper bound'* constraint is pending. Furthermore, canceling the diagram - entering the exit state - due to exit conditions or due to violation of possible constraints has to be interpreted as premature acceptance. Thus,

$$
\begin{aligned}
F = \quad & \{\zeta \mid \forall \zeta_1 \in (past(\zeta) \cup \zeta) \quad \forall \zeta_2 \in future(\zeta): \\
& \quad \forall e_1 \in \zeta_1 \forall e_2 \in \zeta_2 : \\
& \quad \nexists(leadsto, mandatory, e_1, step, [-\infty, m], e_2) \in constr, m \in \mathbb{N} \\
& \quad \wedge \ \nexists(leadsto, mandatory, e_1, superstep, [-\infty, m], e_2) \in constr, m \in \mathbb{N} \\
& \quad \wedge \ \nexists(leadsto, mandatory, e_1, step, [-\infty, \infty), e_2) \in constr \\
& \quad \wedge \ \nexists(distance, mandatory, e_1, step, [-, m], e_2) \in constr, m \in \mathbb{N} \\
& \quad \wedge \ \nexists(distance, mandatory, e_1, superstep, [-, m], e_2) \in constr, m \in \mathbb{N} \ \ \} \\
& \cup \{\zeta_{exit}\}
\end{aligned}
$$

■

The set of transitions is derived from unwinding the diagram w.r.t. constraints and exit conditions:
$T = T_{unwind} \cup T_{exitconds} \cup T_{exitorder} \cup T_{exitdistancetime} \cup T_{exitprogresstime} \cup T_{stuttering}$,where the individual sets are:

- T_{unwind} - transitions corresponding to '*normal*' unwinding the bundle of waveforms BW w.r.t. to *mandatory* constraints. This subset of transitions captures the '*normal*', required changes of the observables in $intf_{\circledS}(A)$ according to the partial order of $Phases_{BW}$ plus the additional ordering and timing relations of events as required by the *mandatory* constraints of TD.

- $T_{exitconds}$ - transitions capturing *exit* conditions of symbolic events.

- $T_{exitorder}$ - transitions provided for *ordering violations of possible* constraints.

[18] Here an important subtlety of *distance* constraints must be emphasized:
A *distance* constraint with timing-interval $[n, m]$ requires event e_2 to be observed at least n steps (or super-steps) and at most m steps (or super-steps) after observation of e_1 and vice versa. If one of the constrained events is observed, then it is r*equired that the other event is observed.* In contrast, a *distance* constraint with timing-interval $[n, \infty]$ only requires a minimal distance without requiring the second event to be observed at all. Hence, *distance* constraints with interval $[n, \infty]$ have no influence on 'progress enforcement', while *distance* constraints with interval $[n.m]$ require progress.

- $T_{exitdistancetime}$ - transitions capturing *distance timing violations of possible* constraints. $T_{exitdistancetime}$ provides transitions to the exit state ζ_{exit} for events, which are observed *earlier* than specified by some *possible* constraint.

- $T_{exitprogresstime}$- transitions resulting from progress timing violations of *possible* constraints. $T_{exitprogresstime}$provides transitions to the exit state ζ_{exit} for events, which are observed *later* than specified by some *possible* constraint.

- $T_{stuttering}$ - transitions - *self loops* - allowing the TSA to re-enter the actual state - and letting time pass - if the actual observation does not trigger a state change.

The transition relation T is constructed in two phases. In the first phase $T_{\_} = T_{unwind} \cup T_{exitconds} \cup T_{exitorder} \cup T_{exitdistancetime} \cup T_{exitprogresstime}$ is constructed. Based upon $T_{\_}$ the transition relation T is completed by adding the self loops $T_{stuttering}$ to the transition relation. For the construction of the transition relation, we need a definition of a *maximum*-successor of a phase w.r.t. all waveforms in a bundle of waveforms.

Definition 6.46 (Maximum Successor of a Phase)
Given a phase $\zeta \in Phases_{BW}$. Let $\zeta_{succ} : Phases_{BW} \rightarrow_{\zeta} Phases_{BW}$ denote a *maximum*-successor which associates a phase ζ with a successor ζ', such that for all events $e \in \zeta$, ζ' contains event e', with $e \rightarrow_{wvf} e'$ according to the order on waveform wvf to which e belongs:

$$\zeta_{succ}(\zeta) := \bigcup_{wvf \in BW} \left\{ e' \in SE_{wvf} | \exists e \in SE\zeta_{wvf} : e \in \zeta \wedge e \rightarrow_{wvf} e' \right\}$$

■

For the construction of the transition relation of $TSA(TD)$ we introduce the following abbreviation:

Definition 6.47 (Linger - Predicate)
Given a symbolic waveform wvf w.r.t. interface declaration $intf_{\textcircled{S}}(A)$ according to definition 6.31.

Let $linger : SE_{wvf} \cup \{\top\} \rightarrow Pred_{\mathcal{V}_{\textcircled{S}}}$ be a mapping that associates a linger condition with each symbolic event. Let

$$linger(e) := \begin{cases} stable(e) \wedge \neg trigger(e) & \text{if } det(e) = deterministic \\ stable(e) & \text{otherwise} \end{cases}$$

■

Definition 6.48 (Construction of Transition Relation for *TSA(TD)*)

Construction of $\mathbf{T_{unwind}}$ (normal unwinding) :
For each phase $\zeta_1 \in Phases_{BW}$:
A transition $t = (\zeta_1, enable, \zeta_2, clocks, timing)$ from phase ζ_1 is added to T_{unwind} for each successor phase $\zeta_2 \in Phases_{BW}$ of ζ_1 according to $\rightarrow_{\zeta}$, iff

ζ_2 adheres to the following conditions:

$$\forall e_1, e_2 \in \zeta_1 : (\exists(distance, -, e_1, step, [0,0], e_2) \in constr \wedge (e_1 \notin \zeta_2)) \Rightarrow e_2 \notin \zeta_2 \tag{6.21}$$

(ζ_2 does not violate a *possible* or *mandatory* simultaneous constraint)

$$\forall e_1, e_2 \in \zeta_1 : (\exists(distance, -, e_1, step, (0,\infty], e_2) \in constr \wedge (e_1 \notin \zeta_2)) \Rightarrow e_2 \in \zeta_2 \tag{6.22}$$

(ζ_2 does not violate a *possible* or *mandatory* conflict constraint)

$$\forall e_1, e_2 \in \zeta_1 : (\exists(distance, -, e_1, -, [n,-], e_2) \in constr, n \in \mathbb{N} \wedge (e_1 \notin \zeta_2)) \Rightarrow e_2 \in \zeta_2 \tag{6.23}$$

(ζ_2 does not violate a *possible* or *mandatory* (step or super-step) separation constraint)

$$\begin{aligned} \forall e_1, e_2 \in \zeta_1 : \quad & ((\exists(precedence, -, e_1, step, (0,\infty], e_2) \in constr \\ & \vee \exists(precedence, -, e_1, -, [n,\infty], e_2) \in constr, n \in \mathbb{N}) \\ & \wedge (e_1 \notin \zeta_2)) \Rightarrow e_2 \in \zeta_2 \end{aligned} \tag{6.24}$$

(ζ_2 does not violate a *possible* or mandatory (step or super-step) precedence constraint)

Let $UnwSucc(\zeta_1)$ denote the set of successor phases of ζ_1 according to $\rightarrow_\zeta$, which adhere to conditions (6.21) - (6.24).
For $\zeta_2 \in UnwSucc(\zeta_1)$, $t = (\zeta_1, enable, \zeta_2, clocks, timing) \in T_{unwind}$ is build as follows:

1. $enable{:=} \bigwedge_{e_i \in \zeta_1 \wedge e_i \notin \zeta_2} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta_1 \cap \zeta_2)} linger(e_j)$, which is the conjunction of propositions associated with the *trigger* predicates of the unwound events and the *linger* predicates of the events in $\zeta_1 \cap \zeta_2$. The transition from ζ_1 to successor phase ζ_2 is enabled only if the *trigger* predicates of all events $e \in \zeta_1 \wedge e \notin \zeta_2$ evaluate to true.

2. The clocks associated with all events in ζ_1 are reset, whose trigger predicates determine the state change by transition t.
   $clocks{:=}\{z_e \in C_{step} | e \in \zeta_1 \backslash (\zeta_1 \cap \zeta_2)\} \cup \{zz_e \in C_\tau | e \in \zeta_1 \backslash (\zeta_1 \cap \zeta_2)\}$.

3. the timing predicate regards all clocks which are reset for some source event of a constraint in $past(\zeta_1)$ with target event in ζ_2 :
   $timing{:=}$

$$
\begin{aligned}
&\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (leadsto, -, e, step, [-\infty, m], e') \in constr, m \in \mathbb{N}} z_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (distance, -, e, step, [-, m], e') \in constr, m \in \mathbb{N}} z_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (distance, -, e, step, [n, -], e') \in constr, n \in \mathbb{N}} z_e \geq n \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (precedence, -, e, step, [n, \infty], e') \in constr, n \in \mathbb{N}} z_e \geq n \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (leadsto, -, e, superstep, [-\infty, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (distance, -, e, superstep, [-, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (distance, -, e, superstep, [n, -], e') \in constr, n \in \mathbb{N}} zz_e \geq n \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists (precedence, -, e, superstep, [n, \infty], e') \in constr, n \in \mathbb{N}} zz_e \geq n
\end{aligned}
$$

Construction of $\mathbf{T_{exitconds}}$ (capturing *exit* conditions of symbolic events):
For each phase $\zeta_1 \in Phases_{BW}$:
A transition $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing)$ from ζ_1 to the exit state ζ_{exit} is added to $T_{exitconds}$ for the dedicated successor phase $\zeta_{succ}(\zeta_1)$ of ζ_1 according to $\rightarrow_\zeta$, with:

1. $enable$:= $\bigvee_{e \in \zeta_1} unstab(e) \wedge exit(e)$, where

$$
unstab(e) := \begin{cases} \neg stable(e) \wedge \neg trigger(e) & \text{if } det(e) = deterministic \\ \neg stable(e) & \text{otherwise} \end{cases}
$$

.

2. no clocks have to be reset: $clocks$:=$\emptyset$

3. the timing predicate has to regard all upper bound constraints related to clocks, which were reset in $past(\zeta_1)$ and are constraint by upper bounds in $future(\zeta_1)$:
$timing$:=

$$
\begin{aligned}
&\bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(leadsto, mandatory, e, step, [-\infty, m], e') \in constr, m \in \mathbb{N}} z_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(distance, mandatory, e, step, [-, m], e') \in constr, m \in \mathbb{N}} z_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(leadsto, mandatory, e, superstep, [-\infty, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m \\
\wedge\ &\bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(distance, mandatory, e, superstep, [-, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m
\end{aligned}
$$

Construction of $\mathbf{T_{exitorder}}$ (capturing violations of *possible ordering* constraints):

For each phase $\zeta_1 \in Phases_{BW}$:
A transition $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing)$ from ζ_1 to the exit state ζ_{exit} is added to $T_{exitorder}$ for each successor $\zeta_2 \in Phases_{BW}$ of ζ_1 according to $\rightarrow_\zeta$, iff ζ_2 violates a *possible* constraint in at least one of the following ways:

$$\exists e_1, e_2 \in \zeta_1 : \exists(distance, possible, e_1, step, [0, 0], e_2) \in constr \wedge (e_1 \in \zeta_2 \wedge e_2 \notin \zeta_2) \tag{6.25}$$

(two events are expected to happen simultaneous but are observed at different times)

$$\begin{aligned}
\exists e_1, e_2 \in \zeta_1 : \quad &(\exists(distance, possible, e_1, step, (0, \infty], e_2) \in constr \\
&\vee \exists(distance, possible, e_1, -, [n, -], e_2) \in constr, n \in \mathbb{N}) \\
&\wedge (e_1 \notin \zeta_2) \wedge (e_2 \notin \zeta_2)
\end{aligned} \tag{6.26}$$

(two events are expected not to be observed simultaneously (separated by a *possible* step or super-step lower separation bound) but both are observed when entering ζ_2)

$$\begin{aligned}
\exists e_1, e_2 \in \zeta_1 : \quad &(\exists(precedence, possible, e_1, step, (0, \infty], e_2) \in constr \\
&\vee \exists(precedence, possible, e_1, -, [n, \infty], e_2) \in constr, n \in \mathbb{N}) \\
&\wedge (e_1 \notin \zeta_2) \wedge (e_2 \notin \zeta_2)
\end{aligned} \tag{6.27}$$

(two events are expected to be observed in a strict order (separated by a *possible* step or super-step lower precedence bound) but both are observed when entering ζ_2)

Let $SuccExOrd(\zeta_1)$ denote the set of successor phases of ζ_1 according to $\rightarrow_\zeta$, which adhere to at least one of the conditions (6.25), (6.26), and (6.27).

For $\zeta_2 \in SuccExOrd(\zeta_1)$, $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing) \in T_{exitorder}$ is build as follows:

1. $enable$:= $\bigwedge_{e_i \in \zeta_1 \wedge e_i \notin \zeta_2} trigger(e_i) \wedge \bigwedge_{e_j \in \zeta_1 \cap \zeta_2} linger(e_j)$.
2. no clocks are reset : $clocks$:=$\emptyset$.
   Since transition t enters ζ_{exit}, no clocks have to be considered an more. Thus, no clocks have to be reset.

3. the timing predicate has to regard all upper bound constraints related to clocks which were reset in $past(\zeta_1)$ and are constraint by upper bounds in $future(\zeta_1)$. Furthermore all lower bound constraints have to be regarded which refer to clocks that were reset in $past(\zeta_1)$ and are constraint by lower bounds for entering events $e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2)$. :
$timing$:=

$$\bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(leadsto, mandatory, e, step, [-\infty, m], e') \in constr, m \in \mathbb{N}} z_e \leq m$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(distance, mandatory, e, step, [-, m], e') \in constr, m \in \mathbb{N}} z_e \leq m$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists(distance, mandatory, e, step, [n, -], e') \in constr, n \in \mathbb{N}} z_e \geq n$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists(precedence, mandatory, e, step, [n, \infty], e') \in constr, n \in \mathbb{N}} z_e \geq n$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(leadsto, mandatory, e, superstep, [-\infty, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists(distance, mandatory, e, superstep, [-, m], e') \in constr, m \in \mathbb{N}} zz_e \leq m$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists(distance, mandatory, e, superstep, [n, -], e') \in constr, n \in \mathbb{N}} zz_e \geq n$$
$$\wedge \bigwedge_{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \exists(precedence, mandatory, e, superstep, [n, \infty], e') \in constr, n \in \mathbb{N}} zz_e \geq n$$

Construction of $\mathbf{T_{exitdistance}}$ (capturing violation of *lower bound distances* as specified by possible constraints):
For each phase $\zeta_1 \in Phases_{BW}$:
A transition $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing)$ from ζ_1 to ζ_{exit} is added to $T_{exitdistance}$ for each successor phase $\zeta_2 \in Phases_{BW}$ of ζ_1 according to $\rightarrow_\zeta$, iff ζ_2 can possibly violate a lower bound distance specification of a *possible* constraint in the following ways:

$$\exists e_1 \in past(\zeta_1), \exists e_2 \in \zeta_2 : \exists(distance, possible, e_1, -, [n, -], e_2) \in constr \ , n \in \mathbb{N} \qquad (6.28)$$

($e_2 \in \zeta_2$ is the target event of a *quantitative possible* separation constraint, which specifies a step or super-step lower bound. The added transition will only be enabled if the clock associated with e_1 is less than n when e_2 is observed)

$$\exists e_1 \in past(\zeta_1), \exists e_2 \in \zeta_2 : \exists(precedence, possible, e_1, -, [n, \infty], e_2) \in constr, n \in \mathbb{N} \qquad (6.29)$$

($e_2 \in \zeta_2$ is the target event of a *quantitative possible* precedence constraint, which specifies a step or super-step lower bound. The added transition will only be enabled if the clock associated with e_1 is less than n when observing e_2)

Let $SuccExLowBd(\zeta_1)$ denote the set of successor phases of $\zeta_1$according to $\rightarrow_\zeta$, which adhere to conditions (6.28) or (6.29).
For $\zeta_2 \in SuccExLowBd(\zeta_1)$, $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing) \in T_{exitdistance}$ is build as follows:

1. $enable$:= $\bigwedge_{e_i \in \zeta_1 \wedge e_i \notin \zeta_2} trigger(e_i) \wedge \bigwedge_{e_j \in \zeta_1 \cap \zeta_2} stable(e_j)$. Transition t has to be taken if ζ_2 can be entered earlier than expected.
2. no clocks are reset : $clocks$:=$\emptyset$. Since ζ_{exit} is entered, no clock have to be considered any more.
3. the $timing$ constraint has to guarantee that t can only be taken as long as a lower bound of some possible distance or precedence constraint with target event in $\zeta_2 \backslash (\zeta_1 \cap \zeta_2)$ can be violated. If all relevant clocks have exceeded the respective lower bound, t will be disabled:
   $timing$:=
$$\bigvee_{\substack{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \\ \exists (distance, possible, e, step, [n, -], e'), n \in \mathbb{N}}} z_e < n$$
$$\vee \bigvee_{\substack{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \\ \exists (precedence, possible, e, step, [n, \infty], e'), n \in \mathbb{N}}} z_e < n$$
$$\vee \bigvee_{\substack{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \\ \exists (distance, possible, e, superstep, [n, -], e'), n \in \mathbb{N}}} zz_e < n$$
$$\vee \bigvee_{\substack{e \in past(\zeta_1), e' \in \zeta_2 \backslash (\zeta_1 \cap \zeta_2) : \\ \exists (precedence, possible, e, superstep, [n, \infty], e'), n \in \mathbb{N}}} zz_e < n$$

Construction of $\mathbf{T_{exitprogresstime}}$ (capturing violation of *upper bound distances* as specified by possible constraints):
For each phase $\zeta_1 \in Phases_{BW}$:
A transition $t = (\zeta_1, enable, \zeta_{exit}, clocks, timing)$ from ζ_1 to exit state ζ_{exit} is added to $T_{exitprogresstime}$ for each successor phase $\zeta_2 \in Phases_{BW}$ of ζ_1 according to $\rightarrow_\zeta$, if ζ_2 can violate some upper bound distance specification of a *possible* constraint in the following ways:

$$\begin{array}{l} \exists e_1 \in past(\zeta_1), \exists e_2 \in future(\zeta_1) : \\ \exists (distance, possible, e_1, -, [-, m], e_2) \in constr, m \in \mathbb{N} \end{array} \tag{6.30}$$

(some source event of a *quantitative possible* distance constraint - which specifies a step or super-step upper bound - is in $past(\zeta_1)$, whereas the target event of this constraint is in $future(\zeta_1)$. The added transition will only be enabled if the clock associated with e_1 has

become greater than m without observing e_2)

$$\begin{array}{l}\exists e_1 \in past(\zeta_1), \exists e_2 \in future(\zeta_1) : \\ \exists (leadsto, possible, e_1, -, [-\infty, m], e_2) \in constr, m \in \mathbb{N}\end{array} \tag{6.31}$$

(some source event of a *quantitative possible* leadsto constraint - which specifies a step or super-step upper bound - is in $past(\zeta_1)$, whereas the target event of this constraint is in $future(\zeta_1)$. The added transition will only be enabled if the clock associated with e_1 has become greater than m without observing e_2)

1. $enable$:= $\bigwedge_{e' \in \zeta_1 \wedge e' \notin \zeta_2} linger(e')$. Transition t has to be taken if ζ_2 can only be entered later than expected.
2. no clocks are reset : $clocks$:=$\emptyset$. Since ζ_{exit} is entered, no clocks are of interest any longer.
3. timing :
   $timing$:=
$$\begin{array}{ll} & \bigvee_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists (leadsto, possible, e, step, [-\infty, m], e')} z_e > m \\ \vee & \bigvee_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists (distance, possible, e, step, [-, m], e')} z_e > m \\ \vee & \bigvee_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists (leadsto, possible, e, superstep, [-\infty, m], e')} zz_e > m \\ \vee & \bigvee_{e \in past(\zeta_1), e' \in future(\zeta_1) : \exists (distance, possible, e, superstep, [-, m], e')} zz_e > m \end{array}$$

Up to this point, the transition relation consists of transitions for all events of which the bundle of waveforms consists. All constraints of the timing diagram are taken into account.

To be able to take a transition in every step also if the actual observation does not trigger a state change of the automaton, the transition relation is completed with self loops. The meaning of $T_{stuttering}$ is that $TSA(TD)$ can re-enter its currently active state as long as nothing happens that enforces a state change.

The transition relation T of $TSA(TD)$ is built from $T_{-} \cup T_{stuttering}$, with:

Construction of $\mathbf{T_{stuttering}}$ (self-loops):

For each phase $\zeta_1 \in Phases_{BW}$:

A transition $t = (\zeta_1, enable, \zeta_1, clocks, timing)$ is added to $T_{stuttering}$, where

- $enable$:= $\bigwedge_{e \in \zeta_1} linger(e)$.
- Stuttering is permitted, when none of the transitions to successor states ζ_{succ} w.r.t. T_{-} is taken. This is the case if all propositions for the *linger* predicates associated with the events of the actual phase evaluate to true.

- no clocks need to be reset: $clocks = \emptyset$. Clocks of the automaton are associated with events. No event is observed.
- no timing has to be considered: $timing = true$. Constraints are already covered by outgoing transitions. If $\zeta_1 \in F$, then the automaton can take $(\zeta_1, enable, \zeta_1, \emptyset, true) \in T_{stuttering}$ infinitely often. Otherwise, there is either an outgoing transition $(\zeta_1, -, \zeta_{exit}, -, -) \in T_-$ or no outgoing transition of ζ_1 is enabled.
- for ζ_{exit} , $(\zeta_{exit}, true, \zeta_{exit}, \emptyset, true) \in T_{stuttering}$ is added.

■

Lemma 6.11 (Unwinding a Diagram yields a Partially Ordered TSA)
$TSA(TD)$ is a Partially Ordered TSA according to definition 6.15.

■

Proof 6.11
Follows from the partial order and finiteness of $Phases_{BW}$ according to definition 6.43 and construction rules of definition 6.48, which strictly adhere to the successor relation $\rightarrow_\zeta$.

□

For the proof of the following lemma, we need an association of a particular transition in $TSA(TD)$ with the phase of diagram TD , for which the unwinding algorithm has introduced the transition in $TSA(TD)$.

Definition 6.49 (Association of TSA-Transition with Unwinding Phase)
Given the unwinding-TSA $TSA(TD) = (\mathcal{V}_{\circledS}, S, s_0, C_{step} \cup C_\tau, T, F)$ of Symbolic Timing Diagram $TD = (intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr)$.
For transition $t \in T$, let $phase(t) : T \rightarrow Phases_{BW}$ denote the phase associated with the target state of t, i.e. the phase $\zeta \in Phases_{BW}$ for which t has been introduced.

■

Lemma 6.12 (Determinism of *TSA(TD)*)
Given a Symbolic Timing Diagram $TD = (intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, \emptyset)$. Then $TSA(TD) = (\mathcal{V}_{\circledS}, S, s_0, C_{step} \cup C_\tau, T, F)$ (according to definitions 6.45 and 6.48) is a deterministic POTSA, if the following conditions are satisfied:

- $\forall wvf \in BW : wvf \text{ is deterministic}$ (6.32)
- $\forall e \in SE_{BW} : det(e) = deterministic$ (6.33)
- $\not\exists(precedence, possible, -, -, [n, \infty], -) \in constr, n \in \mathbb{N}$ (6.34)
- $\not\exists(distance, possible, -, -, [n, \infty], -) \in constr, n \in \mathbb{N}$ (6.35)
- $\not\exists(distance, possible, -, -, [-, m], -) \in constr, m \in \mathbb{N}$ (6.36)
- $\not\exists(leadsto, possible, -, -, [-, m], -) \in constr, m \in \mathbb{N}$ (6.37)

■

Proof 6.12

1. $T_{exitdistance} = \emptyset$.
   Follows from conditions (6.34) and (6.35).

2. $T_{exitprogresstime} = \emptyset$.
   Follows from conditions (6.36) and (6.37).

3. $\forall \zeta, \zeta_1, \zeta_2 \in S : \forall t_1 = (\zeta, enable_1, \zeta_1, -, -) \in T_{unwind} \forall t_2 = (\zeta, enable_2, \zeta_2, -, -) \in T_{unwind} : t_1 \neq t_2 : \nexists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
   By construction of T_{unwind}: $t_1 \neq t_2 \Rightarrow \zeta_1 \neq \zeta_2$ and thus:

   - $enable_1 = \bigwedge_{e_i \in \zeta \wedge e_i \notin \zeta_1} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap \zeta_1)} linger(e_j)$, and
   - $enable_2 = \bigwedge_{e_i \in \zeta \wedge e_i \notin \zeta_2} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap \zeta_2)} linger(e_j)$.

   The assertion follows from pairwise disjointness of $SE_{wvf}, SE_{wvf'}, \forall wvf, wvf' \in BW$ according to definition 6.32 and because wvf, wvf' are deterministic according to condition (6.32).

4. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta_{exit}, -, -) \in T_{exitorder} \forall t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitorder} :/ \exists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
   Likewise, the assertion follows from construction of $T_{exitorder}$ and from pairwise disjointness of $SE_{wvf}, SE_{wvf'}, \forall wvf, wvf' \in BW$ according to definition 6.32 as well as from determinism of wvf, wvf'.

5. $\forall \zeta \in S : \exists^1 t = (\zeta, enable, \zeta, \emptyset, true) \in T_{stuttering}$.
   Trivial. Follows immediately from construction of $T_{stuttering}$w.r.t $\zeta_{succ}(\zeta)$ for all $\zeta \in S$

6. $\forall \zeta \in S : \exists^1 t = (\zeta, enable, \zeta_{exit}, \emptyset, timing) \in T_{exitconds}$.
   Trivial. Follows immediately from construction of $T_{exitconds}$ w.r.t. $\zeta_{succ}(\zeta)$,$\forall \zeta \in S$.

7. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta', -, -) \in T_{unwind} \forall t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitorder} : \nexists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
   $\forall \zeta \in S$: $UnwSucc(\zeta) \cap SuccExOrd(\zeta) = \emptyset$. This follows immediately from definition of $UnwSucc(\zeta)$ and $SuccExOrd(\zeta)$:

   - condition (6.21) $\Leftrightarrow$ $\neg$condition (6.25)
   - conditions (6.22)$\wedge$(6.23)$\Leftrightarrow$ $\neg$condition (6.26)
   - condition (6.24)$\Leftrightarrow$ $\neg$condition(6.27)

   Hence, $\forall \zeta' : \exists t_1 = (\zeta, -, \zeta', -, -) \in T_{unwind} :/ \exists t_2 = (\zeta, -, \zeta_{exit}, -, -) \in T_{exitorder}$ with $phase(t_2) = \zeta'$ and vice versa.

   Consequently:
   $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta', -, -) \in T_{unwind} \Rightarrow \forall t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitorder} : \zeta' \neq phase(t_2)$, and
   $\forall t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitorder} : \zeta' = phase(t_2) \Rightarrow \nexists t_1 = (\zeta, enable_1, \zeta', -, -) \in T_{unwind}$.

In particular holds: $\forall t_1 \in T_{unwind} \forall t_2 \in T_{exitorder} : (phase(t_1) \backslash (\zeta \cap phase(t_1))) \neq (phase(t_2) \backslash (\zeta \cap phase(t_2)))$. According to definition 6.48:

- $enable_1 = \bigwedge_{e_i \in \zeta \wedge e_i \notin \zeta'} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap \zeta')} linger(e_j)$
- $enable_2 = \bigwedge_{e_i \in \zeta \wedge e_i \notin phase(t_2)} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap phase(t_2))} linger(e_j)$

Since $\zeta \rightarrow_\zeta \zeta'$ and $\zeta \rightarrow_\zeta phase(t_2)$, we have:

$$\forall e' \in \zeta' \forall e'' \in phase(t_2) : (\exists wvf \in BW : e', e'' \in SE_{wvf}) \Rightarrow ((e' \rightarrow_{wvf} e'') \vee (e'' \rightarrow_{wvf} e')).$$

Because $\zeta' \neq \zeta$ and $phase(t_2) \neq \zeta$ and $\zeta' \neq phase(t_2)$, there exist at least one waveforms wvf, for which ζ' and ζ'' differ. Since wvf is deterministic (cf. definition 6.31), and since $\forall wvf, wvf' \in BW : SE_{wvf}$ and $SE_{wvf'}$ are pairwise disjoint according to definition 6.32, $enable_1$ and $enable_2$ are mutual exclusive and hence $\nexists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.

8. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta', -, -) \in T_{unwind}, t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitconds} : \nexists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
   Follows from the definition of T_{unwind} and $T_{exitconds}$:
   - $enable_1 := \bigwedge_{e_i \in \zeta \wedge e_i \notin \zeta'} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap \zeta')} linger(e_j)$
   - $enable_2 := \bigvee_{e \in \zeta} unstab(e) \wedge exit(e)$, where
     $unstab(e) := \begin{cases} \neg stable(e) \wedge \neg trigger(e) & det(e) = deterministic \\ \neg stable(e) & \text{otherwise} \end{cases}$

   Hence, $t_1 \in T_{unwind}$ can only be enabled if for all events $e \in \zeta, e \notin \zeta'$ it holds $\exists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models trigger(e)$, while for all events $e \in (\zeta \cap \zeta')$ holds: $\exists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models stable(e)$. In contrast, t_2 in $T_{exitconds}$ can only be enabled if at least for one event $e \in \zeta$ it holds $\exists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models \neg trigger(e) \wedge \neg stable(e)$ (by condition (6.33)).

9. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta', -, -) \in T_{unwind}, t_2 = (\zeta, enable_2, \zeta, -, -) \in T_{stuttering} : \nexists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
   Trivial. Follows immediately from the definition of T_{unwind} and $T_{stuttering}$.

10. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta_{exit}, -, -) \in T_{exitorder}, t_2 = (\zeta, enable_2, \zeta_{exit}, -, -) \in T_{exitconds} : / \exists \sigma \in \Sigma_{\mathcal{V}_{\circledS}} : \sigma \models (enable_1 \wedge enable_2)$.
    Follows from the definition of $T_{exitorder}$ and $T_{exitconds}$:
    - $enable_1 := \bigwedge_{e_i \in \zeta \wedge e_i \notin phase(t_1)} trigger(e_i) \wedge \bigwedge_{e_j \in (\zeta \cap phase(t_1))} linger(e_j)$
    - $enable_2 := \bigvee_{e \in \zeta} unstab(e) \wedge exit(e)$, where
      - $unstab(e) := \begin{cases} \neg stable(e) \wedge \neg trigger(e) & det(e) = deterministic \\ \neg stable(e) & \text{otherwise} \end{cases}$

Hence, $t_1 \in T_{exitorder}$ can only be enabled if for all events $e \in \zeta, e \notin phase(t_1)$ holds $\exists \sigma \in \Sigma_{\mathcal{V}_{(S)}} : \sigma \models trigger(e)$, while for all events $e \in (\zeta \cap phase(t_1))$ $\exists \sigma \in \Sigma_{\mathcal{V}_{(S)}} \sigma \models stable(e)$ holds. In contrast, t_2 in $T_{exitconds}$ can only be enabled if $\exists \sigma \in \Sigma_{\mathcal{V}_{(S)}}$ s.t. for one event $e \in \zeta$ holds $\sigma \models (\neg trigger(e) \wedge \neg stable(e))$ (by condition (6.33)).

11. $\forall \zeta \in S : \forall t_1 = (\zeta, enable_1, \zeta_{exit}, -, -) \in T_{exitorder}, t_2 = (\zeta, enable_2, \zeta, -, -) \in T_{stuttering} :/ \exists \sigma \in \Sigma_{\mathcal{V}_{(S)}} : \sigma \models (enable_1 \wedge enable_2)$.
Follows form the definition of $T_{exitorder}$ and $T_{stuttering}$: In order to enable a transition $t_1 \in T_{exitorder}$ at least the *trigger* condition $trigger(e)$ of some event $e \in \zeta \wedge e \notin phase(t_1)$ has to be satisfied by some $\sigma \in \Sigma_{\mathcal{V}_{(S)}}$, while $t_2 \in T_{stuttering}$ can only be enabled, if $\sigma \models \neg trigger(e)$, $\forall e \in \zeta$.

12. $\forall \zeta \in S : t_1 = (\zeta, enable_1, \zeta_{exit}, -, -) \in T_{exitconds}, t_2 = (\zeta, enable_2, \zeta, -, -) \in T_{stuttering} : \nexists \sigma \in \Sigma_{\mathcal{V}_{(S)}} : \sigma \models (enable_1 \wedge enable_2)$.
Follows form the definition of $T_{exitconds}$ and $T_{stuttering}$: In order to enable a transition $t_1 \in T_{exitconds}$ at least the *stable* condition $stable(e)$ of some event $e \in phase(t_1)$ would have to be violated by the same valuation $\sigma \in \Sigma_{\mathcal{V}_{(S)}}$, which satisfies $\forall e \in \zeta : \sigma \models stable(e)$ in order to enable $t_2 \in T_{stuttering}$.

□

Notice, that renouncement of quantitative possible constraints means no lack of expressiveness: instead of using *quantitative possible* constraints in commitments, explicit assumption diagrams using appropriate *quantitative mandatory* constraints can be used; and instead of using *quantitative possible* constraints in assumptions, commitment diagrams using appropriate *quantitative mandatory* constraints can be used.

Lemma 6.13 (Global Constrainedness of *TSA(TD)*)
Given a $TD = (intf_{(S)}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, \emptyset)$. $TSA(TD)$ *is globally constrained, iff*

- $\forall c = (leadsto, mandatory, -, -, \mathbf{T}, -) \in constr : \mathbf{T} = [-\infty, m], m \in \mathbb{N}$ (6.38)
- $\nexists c = (leadsto, possible, -, -, [-\infty, m], -) \in constr, m \in \mathbb{N}$ (6.39)
- $\nexists c = (distance, possible, -, -, [-\infty, m], -) \in constr, m \in \mathbb{N}$ (6.40)

■

Proof 6.13
According to definition 6.45, states $\zeta \in S$ are fair iff:

$$
\begin{aligned}
F = \quad & \{\zeta \mid \forall \zeta_1 \in (past(\zeta) \cup \zeta) \quad \forall \zeta_2 \in future(\zeta): \\
& \forall e_1 \in \zeta_1 \forall e_2 \in \zeta_2 : \\
& \nexists (leadsto, mandatory, e_1, step, [-\infty, m], e_2) \in constr, m \in \mathbb{N} \\
& \wedge \nexists (leadsto, mandatory, e_1, superstep, [-\infty, m], e_2) \in constr, m \in \mathbb{N} \\
& \wedge \nexists (leadsto, mandatory, e_1, step, [-\infty, \infty), e_2) \in constr \\
& \wedge \nexists (distance, mandatory, e_1, step, [-, m], e_2) \in constr, m \in \mathbb{N} \\
& \wedge \nexists (distance, mandatory, e_1, superstep[-, m], e_2) \in constr, m \in \mathbb{N} \quad \} \\
\cup & \{\zeta_{exit}\}
\end{aligned}
$$

Transitions without upper bounds in their *timing* constraints are only added to the transition relation of $TSA(TD)$ for *quantitative possible* constraints specifying an *expected* upper bound on some clock ($T_{exitprogresstime}$: in definition 6.48) and for '*staying*' in some state because the actual observation enforces no state change ($T_{stuttering}$: in definition 6.48) as well as for *qualitative mandatory* leadsto constraints (requiring liveness in a pure form without upper time bound - according to T_{unwind}: in definition 6.48). *Quantitative possible* constraints are excluded from consideration due to conditions (6.39) and (6.40).

Except for premature acceptance due to exits according to transitions in $T_{exitconds}$ or $T_{exitorder}$ or $T_{exitdistance}$ always all symbolic events $e \in SE_{BW}$ are considered in some order permitted by the diagram TD along all paths through $TSA(TD)$.

- In $T_{exitconds}$ - by definition - all upper bound constraints related to clocks, which were reset in $past(\zeta_1)$ and are constrained by upper bounds in $future(\zeta_1)$ are respected.
- In $T_{exitorder}$ - by definition - all upper bound constraints related to clocks which were reset in $past(\zeta_1)$ and are constrained by upper bounds in $future(\zeta_1)$ are respected.
- Transitions $t \in T_{exitdistance}$ guarantee that t can only be taken as long as a lower bound of some possible distance or precedence constraint with target event in $\zeta_2 \backslash (\zeta_1 \cap \zeta_2)$ can be violated. If all relevant clocks have exceeded the respective lower bound, t will be disabled. Hence $t \in T_{exitdistance}$ are upper bounded.

Hence, it remains to be proved that condition (6.38) guarantees global constrainedness:

On entering any state ζ'' in $TSA(TD)$, all timing constraints[19] associated with $e'' \in \zeta''$ along every path to ζ'' in $TSA(TD)$ must be satisfied. Construction of T_{unwind} guarantees, that all transitions entering ζ'' specify an upper bound timing constraint on either $z_{e''}$ or $zz_{e''}$ iff e'' has been the target event of a quantitative mandatory (step or super-step) leadsto constraint. Hence, for unfair states according to 6.45, always upper bound constraints for all possible successor states are part of the *timing* predicates along every path to a state containing target events of upper bounded leadsto constraints.

□

Definition 6.50 (Unwinding Representation of Symbolic Timing Diagram ($\mathcal{U}(TD)$))

The unwinding representation $\mathcal{U}(TD)$ of a symbolic timing diagram

$$TD = (intf_{\circledS}(A), name, context, actmode, BW, e_{act}, e_{ac}, e_{ae}, constr, \emptyset)$$

(cf. definition 6.34) is a POTSA_{AC} according to definition 6.24:

$$\mathcal{U}(TD) := (actmode', e_{acond}, e_{aexcept}, \mathcal{A}), \text{ where}$$

- $actmode' := actmode$ is the activation mode for activating $\mathcal{A}$
- $e_{acond} := \begin{cases} e_{act} \wedge e_{ac} & \text{if } actmode = iterative \\ e_{act} & \text{if } actmode = initial \end{cases}$
  is the activation condition for activating $\mathcal{A}$

[19] Recall, that each event $e \in SE_{BW}$ is associated with two clocks, $z_e \in C_{step}$ and $zz_e \in C_\tau$ which are reset for observations $\sigma \in \Sigma_{\mathcal{V}_{\circledS}}$ satisfying $trigger(e)$ if the actual state of $TSA(TD)$ is ζ, with $e \notin \zeta$ and if $\zeta \rightarrow \zeta'$, with $e \in \zeta'$ (e is expected to happen next).

- $e_{aexcept}$:=$\begin{cases} false & \text{if } actmode = iterative \\ e_{ae} & \text{if } actmode = initial \end{cases}$
  is the activation exception for $\mathcal{A}$. Recall from definition 6.24 that $e_{aexcept}$ is relevant only for *initial* POTSA$_{AC}$
- $\mathcal{A}$:=$TSA(TD)$ is the unwinding-TSA $TSA(TD)$ according to definitions 6.45 and 6.48.

According to definition 6.26 we write $\Omega(\mathcal{U}(TD)) \circ f$ to denote the SMI-observer module encoding $\mathcal{U}(TD)$ with dedicated fairness condition output f. If $TSA(TD)$ is globally constrained, we write $\Omega(\mathcal{U}(TD)) \triangleright o$ to denote the SMI-observer module encoding $\mathcal{U}(TD)$ with dedicated stepwise non failure acceptance output o.

■

Finally, we define the semantics of a STDx-specification based upon the languages accepted by the observer modules obtained from the unwinding representations (according to the algorithms and results of section 6.3) of all diagrams belonging to a STDx-specification.

Definition 6.51 (Semantics of a STDx-specification)
Let $spec = (intf(A), name, ass\_decls, comm\_decls, specvars)$ be a STDx-specification. The semantics of *spec* is defined by the timed observation sequences (c.f. 5.6), i.e. by the languages accepted by the unwinding representations of the individual diagrams:

$$\mathcal{L}(spec) := \left\{ ts \mid \left(ts \notin \bigcap_{a \in ad(spec)} \mathcal{L}(\mathcal{U}(a)) \right) \vee \left(ts \in \bigcap_{c \in cd(spec)} \mathcal{L}(\mathcal{U}(c)) \right) \right\}$$

■

Recall from section 6.3 that for deterministic POTSA$_{AC}$ CTL and LTL model checking, respectively, can be applied in order to determine whether a system fulfills a given specification. Theorem 6.3 established that for explicitly constrained deterministic POTSA$_{AC}$ verification using invariants is applicable.

Definition 6.52 (Verification using Deterministic STDx-specifications)
Let $\mathcal{C} = (V, \Theta, \rho, E)$ be a CSTS and $spec = (intf(A), name, ass\_decls, comm\_decls, specvars)$ be a STDx-specification. If all diagrams of *spec* are deterministic, verification can be applied by constructing the parallel composition

$$\mathcal{C} \quad ||_{\Omega} \underset{a \in ad(spec)}{} \Omega(\mathcal{U}(a)) \circ f_a \quad ||_{\Omega} \underset{c \in cd(spec)}{} \Omega(\mathcal{U}(c)) \circ f_c$$

and checking the acceptance formula

$$\bigwedge_{a \in ad(spec)} \mathtt{GF}(f_a) \Rightarrow \bigwedge_{c \in cd(spec)} \mathtt{GF}(f_c)$$

If all unwinding-TSA of all diagrams referred to by *spec* are deterministic and globally constrained, then verification can be applied by constructing:

$$\mathcal{C} \quad ||_{\Omega} \underset{a \in ad(spec)}{} \Omega(\mathcal{U}(a)) \triangleright o_a \quad ||_{\Omega} \underset{c \in cd(spec)}{} \Omega(\mathcal{U}(c)) \triangleright o_c$$

and checking the acceptance formula

$$\bigwedge_{a \in ad(spec)} \mathtt{G}(o_a) \Rightarrow \bigwedge_{c \in cd(spec)} \mathtt{G}(o_c)$$

■

Complexity Issues

It must be emphasized, that STDx is a non-canonical specification formalism, i.e. a requirement can often be specified in many different ways. Also, it depends on the user as well on the particular requirement, whether the requirement is specified using a single diagram or with a set of diagrams. As a rule of thumb, the complexity of the resulting TSA depends on the degree of independence of the symbolic events in a diagram. If the symbolic events of various waveforms are temporally unrelated (i.e. not ordered or bound by constraints), then unwinding has to regard a relatively large amount of possible interleavings and hence the TSA has to represent all phases according to the degree of freedom incorporated by the diagram. In order to get an idea of the worst case complexity of the resulting TSA, we assume a diagram with n symbolic waveforms with one symbolic event on each (after activation). Then there exist 2^n possible phases, which have to be represented as states in the TSA with $(2^n - 1) + \sum_{i=0}^{n} \left(\left(\begin{array}{c} n \\ i \end{array} \right) (2^{n-i} - 1) \right)$ transitions (where the first $(2^n - 1)$ transitions belong to $T_{exitconds}$ and the rest are transitions of T_{unwind}). In contrast, the TSA of a diagram with n symbolic events along one waveform - which are hence totally ordered - requires n states and $2n$ transitions (where n transitions belong to $T_{exitconds}$ and n transitions belong to T_{unwind}).

Due to the treatment of parameters and the instantiation of specifications, observers can not be shared among different declarations and specifications, even though they are generated from the same diagrams. Hence, complexity of the representation of a requirement specification comprised of several STDx-specifications depends on the grouping of diagrams in declarations. Notice, that is is not possible to instantiate the same diagram template twice with different parameter mappings within the same declaration. The most concise representation can be obtained by grouping several diagrams within one declaration, while formalizing of a requirement specification comprised of several STDx-specifications will in general lead to a more complex representation due to redundant references to assumption diagrams.

Conclusion

In this section it has been shown how requirement specifications formalized using the graphical specification formalism STDx can be applied for verification of embedded systems. Graphical specifications with an intuitive reading and a formal semantics are unwound and observers are constructed, for which an efficient verification procedure exists.

- The consequence of lemmata 6.12 and 6.13 is that invariance checking is applicable for verification using STDx, if no quantitative possible constraints (determinism) and no unbounded leadsto constraints (global constrainedness) are used in requirement specifications. As stated, possible quantitative constraints can be avoided in commitments by formulating appropriate assumptions using mandatory quantitative constraints and vice versa.
  Concerning unbounded leadsto constraints, we claim that in the specification and verification

of embedded reactive systems, there always exists a concrete upper time bound for required reactions. The observer construction for STDx is capable of capturing bounded liveness requirements. Because the time model for conceptual models should not resemble physical time and because it should be possible to specify a concrete upper bound for reaction deadlines, we claim that specification of unbounded progress is not a necessity. Hence renunciation of unbounded liveness is an acceptable price to be paid for applicability of invariance checking, instead of using LTL model checking with fairness constraints.

- Using SMI observer modules generated from TSAs obtained from unwinding diagrams, iterative activation can easily be realized. This remedies the problems arising with invariant activation, where several contradictory instances of a diagram can be active at the same time, which often led to counter-intuitive error-paths for older versions of Symbolic Timing Diagrams.

- The distinction between step and super-step constraints allows to specify the input/output behavior of internal activities of a system in terms of model-time as well as in terms of δ-delays. The interfaces of internal activities in general contain fast as well as slow inputs. For reactions it has often to be distinguished whether an activity reacts to stimuli in a step oriented way, i.e. in terms of δ-delays, or in terms of model-time, i.e. a certain amount of super-steps after the stimulus. Since a system is synchronously composed of such internal activities also in the asynchronous execution semantics, being able to quantitatively specify time bounds w.r.t. both steps and super-steps is a prerequisite for compositional verification of real-time requirements for systems according to the asynchronous semantics.
  In section 7.3 a proof-rule for compositional verification is presented. In section 8.3.2, an example for a compositional proof of a real-time requirement is presented.

- Unfortunately, the restriction of STDx-specifications of referring to only one commitment declaration limits the use of template diagrams, because only one parameter mapping can be specified for the instantiation of commitment diagrams. This implies no severe restriction for component proofs, because each requirement can be verified in a separate proof. For compositional proofs the restriction is a real limitation, as will become apparent in section 8.3.3. In order to share assumptions for several instances of a diagram template with different parameter mappings it would be necessary allow specifications to refer to more than one commitment declaration. Hence, a topic for future work enhancing the formalism should be the support of multiple references to commitment declarations by STDx-specifications.

- Recall from section 5.4 that STDx-specifications can be associated with architectures in the same way as behavior descriptions. Architectures - serving as containers - are bound uniquely to component instances by configurations as the central building blocks. Different views, such as behavioral description and requirement specifications, are associated with different configurations referring to the same entity. This concept of defining bindings of components with particular views by separate configurations has the advantage of fixing the structural description independently of a concrete view to the components. SSL itself does not describe behavior, in particular communication is not delayed but instantaneous. Hence, structure descriptions can always be interpreted as the synchronous parallel composition of their components.
  Using configurations, a hierarchical structure can be exploited for structured handling of proof

obligations and verification results in the STVE. The structure description serves as basis for design navigation, proof obligation generation and management of verification results. This will be explained in section 7.4.

6.5.5 Related Work

Symbolic Timing Diagrams have already a long history at the research institute OFFIS. Invented and first presented in the early 1990ies [DS93], many enhancements and changes have been applied to syntax, graphical representation and semantics of Symbolic Timing Diagrams during more than a decade of ongoing development.

A first description of a formal semantics of - an early version of - STD was given by Rainer Schlör and Werner Damm in [DS93] . A full semantics definition for this version of STD can be found in [DJS95]. For this early version only qualitative constraints have been supported. A structured organization using the three layers of diagrams, declarations and specifications has not been part of this early graphical formalism; only textual specification clauses were used to associate commitments with assumptions. Also, specification variables and templates with formal parameters have been introduced later.

The semantics has been defined by unwinding the single diagrams into partially ordered symbolic automata (POSA). POSA are basically Büchi Automata with symbolic expressions as transition triggers. For POSA a translation to linear temporal logic formulae has been defined. The complete foundation and description of the constructs and algorithms applied in this translation can be found in [Sch00].

A first real time version (RTSTD) of STD has been developed in the context of the ESPRIT project SACRES. This version together with a brief semantic definition has been presented by Konrad Feyerabend and Bernhard Josko in [FJ97]. In contrast to STD, unwinding of RTSTD has been defined on the basis of Timed Automata as unwinding representation. From these variant of timed automata TPTL^c formulae have been generated (cf. section 6.3). Since one global clock is assumed in TPTL^c, RTSTD can consequently only refer to one sort of clock. Although the global clock can refer to an arbitrary time domain- not necessarily discrete time, RTSTD have been applied - to the best of our knowledge - always with the assumption of "next step=next time", due to the lack of a TPTL^c tableaux generation.

Within the ESPRIT-project FORMAT the powerful verification environment CVE for VHDL-based hardware designs was built employing major state-of-the-art methods and using STD as graphical specification formalism.

The CVE-verification environment was originally developed by Siemens; later on - in the VFORMAT project - it was extended and marketed by the company Abstract-Hardware under the product name CheckOff.

During a cooperation of several years with the research center OFFIS, this company also provided a graphical design capture tool for STD-specification development. Another implementation of a graphical editor for STD has been developed on top of Tcl/Tk in the context of our master thesis.

In the ESPRIT project VFORMAT - a successor project of FORMAT - the extended Symbolic Timing Diagrams (STDx) formalism has been developed. STDx is a conservative extension of the visual formalism STD. The semantics is similar to the development of [FJ97]; the main difference is the definition of constraint-priority (violations of possible constraints have priority over the violation of strong constraints) and the treatment of symbolic events in a deterministic way by default. Moreover, for STDx the restriction has been made that quantitative-timing constraints always refer

to step clocks. Thus, it has been possible to define the semantics of STDx using the next time-operator of LTL. This ensured still a tractable verification complexity, provided that the numbers occurring in constraint intervals are reasonably small.

The first design of a user interface for STDx was again pursued by Abstract-Hardware within the course of the VFORMAT project. Later on, the Tcl/Tk based STD editor developed at OFFIS has also been adapted and extended in order to support STDx.

Unfortunately, STDx has never been documented with all language constructs in any document. Even though STDx has been referred to in a series of publications, there exists no reference for the unwinding algorithm before the modifications applied in the context of this work.

As stated above, in contrast to the unwinding algorithms for RTSTD and STD, violation of possible constraints has priority over mandatory constraints. In combination with quantitative constraints this induces some difficulties regarding determinism and adherence to upper bound constraints. The major contribution of this work has been in two aspects:

- The variant of STDx presented here permits quantitative constraints regarding two different concepts of time by distinguishing clocks counting steps and clocks counting super-steps.
- The semantics of our STDx variant is defined in terms of observer modules instead of linear temporal logic. This, for the first time, permits realization of an iterative activation in a satisfactory way.

Kathryn Fisler presented in her PhD thesis [Fis96] another variant of timing diagrams (TDL) that allows to formalize non-regular languages. In general the semantics of TDL requires a more general class of languages than regular languages due to the use of variables in time-bound annotations, but a certain regular sub-class of TDL can be translated into CTL - a later publication also presents a LTL semantics [Fis00]. TDL is embedded in a framework of six hardware design notations (heterogeneous hardware language-HHL). Like STDx, TDL supports different activation modes of diagrams; diagrams can either be activated invariantly or iteratively. Real time constraints of TDL refer to steps of the model according to the assumption ”next time=next step”. In contrast to STDx, TDL assumptions are not separate explicit diagrams but are expressed as part of the diagram.

Nina Amla et al. presented another variant of timing diagrams called ’(Synchronous) Regular Timing Diagrams’ (SRTD) [AEN99, AEKN00]. An SRTD is specified by a number of waveforms which describe changes of values w.r.t. a given clock. Activation is realized by splitting the diagram into a pre- and a postcondition fragment, where the precondition triggers the activation of the postcondition fragment. Furthermore several instances of a diagram may either be activated overlapping or may only be activated according to a non-overlapping semantics. The semantics definition is oriented towards a decompositional verification method: a deterministic finite automaton for the complement of each individual waveform and each sequential dependency between waveforms is created, as well as a non-deterministic finite $\omega-$automaton realizing the activation of the individual waveform automata. If all complement (waveform-) automata reject the computation after activation, then the computation adheres to the property specified by the diagram. Real time constraints refer to steps of the model.

A further interesting approach to visual specificatio ns can be found in the work of Cheryl Kleuker (formerly Dietz) [Die96, DD97] about Constraint Diagrams. There, waveforms are partitioned into intervals which are annotated with conditions describing system states. Timing requirements between intervals of different waveforms as well as regarding the duration of individual intervals can be expressed using real-time constraints. Assumptions and commitments are specified within the

same diagrams, whereat commitments are graphically denoted by surrounding boxes. Semantically, Constraint Diagrams are based on a subset of the interval temporal logic Duration Calculus.

Constraint Diagrams have mainly been used within a transformational approach for the construction of correct systems by refinements, but have been applied to verification as well. Reachability based model checking for timed automata on the basis of Constraint Diagrams has been presented by Henning Dierks and Marc Letrari in [DL02]. There, an automatic construction of so-called test automata from Constraint Diagrams has been described, with which verification using the model checker Uppaal has been applied.

Finally, the work of Jochen Klose has to be mentioned [Klo03]. Even though, Live Sequence Charts (LSC) seem to be different to STDx at first glance, the formalism uses very similar TSA as semantical basis of its unwinding algorithm. In [Klo03], LSCs refer only to steps of a model, whereas referring also to super-step counters would be an advantage even for specifications at system-level. Hence, in particular the considerations regarding two different kinds of clocks are of interest also for the formalism of LSCs. The translation of TSA into specification observers (cf. section 6.3) could be applied also for TSA obtained from unwinding of LSCs, and thus improve the real-time capabilities of LSCs.

7 Verification Techniques for Complex Embedded Systems

In this chapter we focus on the verification techniques for complex embedded systems integrated with the STVE. In the previous chapter we have described the formalization of requirements for the on-the-fly techniques of robustness analyses, formal debugging, and pattern-based verification and the construction of synchronous observers from Timed Symbolic Automata obtained from unwinding of Symbolic Timing Diagrams. According to the placement of the different techniques in the development process, the STVE is comprised of two major components, a so-called verification manager offering basis functionality and on-the-fly verification and a proof-manager for specification verification using STDx. Even though verification-manager and proof-manager are separate integration platforms, they share some optimization and approximation techniques in their respective verification applications.

This chapter is organized as follows: section 7.1 presents the overall structure of the STVE and the interaction of its parts. In section 7.2, the optimization and approximation techniques integrated with STVE are presented. The principle of compositional verification using STDx is described in section 7.3. Section 7.4 concretizes the presentation of compositional verification by relating the basic techniques to the concrete structure representation using SSL. There also the proof-management as integrated with the proof-manager is described. The chapter concludes with an overview of other approaches to compositional verification.

7.1 Structure of the Statemate Verification Environment

Figure 7.1 gives an overview about the overall integration of the STVE. Solid lines in the figure denote data-flow, while dotted lines depict flow of control.

The verification-manager is closely integrated with the STATEMATE user interface. Using a plug-in mechanism of the STATEMATE tool set, the verification-manager can be invoked directly from the main window by simply pressing a button (①). The verification-manager is the central control user interface of the STVE. This GUI offers invocation of the Statemate to SMI/SSL compilation (②) , construction and execution of robustness analyses, formal debugging checks (cf. section 6.1) and observer-pattern-based verification tasks (cf. section 6.4).

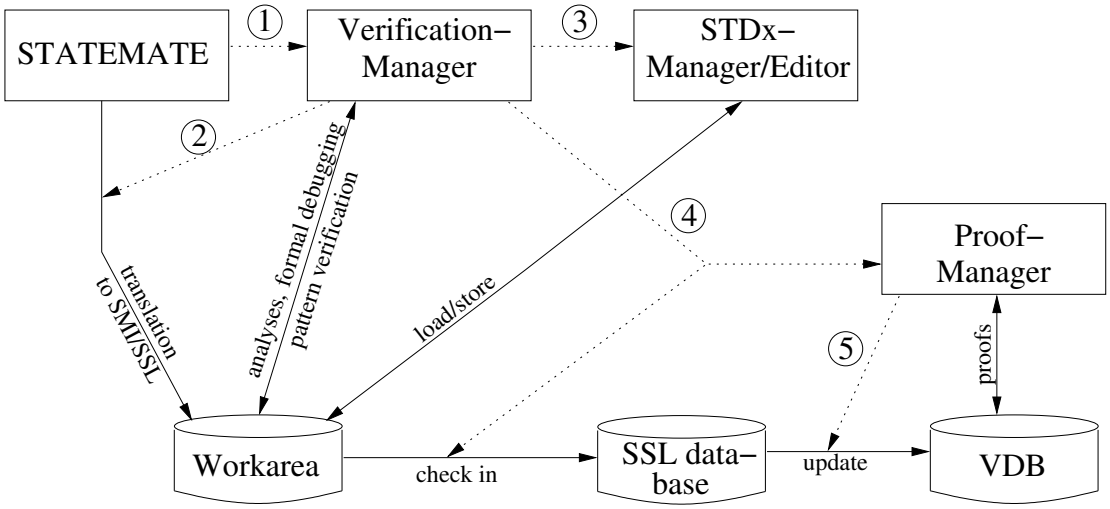


Figure 7.1: The Overall Picture of Integration

The STDx-manager (cf. section 6.5) can be invoked (③) from the verification-manager for each individual sub-activity of the design in order to create or edit specifications for verification using the proof-manager. Up to this point, all data are stored in the user's workarea. In order to perform requirement verification for STDx specification, the design representation has to be checked into an SSL database, and the proof-manager has to be invoked. Again check-in for the design representation as well as for STDx specifications can be controlled using the verification manager (④). If there exists already an active incarnation of the proof-manager, this incarnation is notified about the check-in (also ④). Upon notification - as well as upon start - the proof-manager compares the contents of the SSL data-base with the mirrored contents in its verification data base (VDB) and performs the necessary update operations on the VDB (⑤).

The Verification-Manager

By the compilation of a STATEMATE design into its SMI/SSL representation, for each activity a directory is created containing all files belonging to the semantical representation [Bro99]. The verification-manager offers creation of proof-obligations for robustness analyses, formal debugging checks and observer-pattern-based verification (cf. section 6.4). Proof-obligations are created and the corresponding proof-tasks are executed in the workarea in dedicated directories located in the directory associated with the chosen activity, s.t. for each individual analysis, debugging or pattern-proof a separate directory is used. Construction of proof-obligations is realized using a pure push-button technique. For analyses the potential conflicts are offered - after a preparation phase - in a selection list, s.t. the user can define a proof-obligation by simple graphical selection. For robustness analyses and drive-to-state debugging checks a multiple selection is supported, in the remaining cases only single definitions are possible.

Instances of observer-pattern are created by selection from a list, assigning the instance a name and mapping the formal parameters to concrete expressions in a dedicated dialog. For entering expressions - to be mapped to the formal parameters of a pattern or to define a drive-to-property check - a list of the variables, states and events in the scope of the selected activity is offered which

permits easy graphical selection.

User defined pattern instances are stored in a separate directory and can be shared between different proof-obligations. This allows to define e.g. assumptions only once and to reuse these definitions in various proofs.

All verification related operations are managed by the user interface, hiding by default all control aspects from the user. Results are recorded in reports, no further book-keeping of results or dependence management is supported. For each of the proof-obligations an executable script is generated. Hence, results can be re-established on demand after modifications in the design. Proof-obligations can be redefined, copied and, of course, deleted. Proof-tasks can be re-executed any-time, with or without modification of the proof-obligation or the proof-configuration.

Witness sequences are collected and managed by the verification manager. Part of this management is the translation of witnesses into Statemate simulation control programs. Hence, the user can easily animate witnesses and errorpaths by using the Statemate simulator. Additionally, witnesses can be displayed as set of waveforms.

Optimizations and abstractions can be enabled or disabled, using an easy-to-use graphical configuration dialog. Propositional abstraction and automatic abstraction refinement are offered permitted only for observer-pattern based proofs. These abstractions perform over-approximations of the model-behavior and should hence not be combined with falsification-oriented checks such as in particular formal debugging.

For all checks offered by the verification-manager, the verification engine can be chosen to be either the VIS model checker or the Prover plug-in based bounded model checker. Furthermore, for each check performed using the VIS model checker it can be chosen whether CTL model checking or invariance checking is to be used. For each check using the bounded model checker an lower unroll-bound as well as an upper unroll-bound can be specified. These user-settings are managed by the verification-manager as configurations of the proof-scripts. Hence, a proof-obligation can be executed with various different configurations. This allows the advanced user - without bothering him with details of control aspects - to gain significant control about the applied techniques at an appropriate abstract level.

For the unexperienced user all configurations are predefined with useful defaults, to which the user settings can always be reset.

Summarizing, the verification-manager integrates all verification activities that are offered for application during development of a STATEMATE model with an easy-to-handle graphical user interface. Application of the offered techniques to a STATEMATE model under development does only require a little bit of expert knowledge.

But the verification-manager bridges also the gap to the more ambitious approach of applying specification verification using STDx. Therefore the verification-manager offers control over the check-in of the model representation as well as the STDx specifications into the SSL data-base, that serves as entry point to verification facilities integrated with the proof-manager. The STDx-manager with which STDx specification can be created and edited and which controls creation and editing of diagrams can be invoked for each activity representation in the structural view to the model. The proof-manager can be invoked in order to perform specification verification.

The Proof-Manager

Once invoked, the proof-manager checks the contents of the SSL data-base for modification w.r.t. the contents of an existing VDB. If there exists no VDB yet, the proof-manager creates one according

to the contents of the SSL data-base. Otherwise, validity of existing proof-results is computed according to the description in section 7.4. The VDB basically mirrors the contents of the SSL data-base in a particular directory structure. Symbolic links in the file-systems permit navigation of the design according to the structure description. The VDB provides the data-basis for the integrated file-system oriented verification tools developed at the C.v.O.- University of Oldenburg and by OFFIS, as well as the model checking engines.

The proof-manager offers proof-obligation construction for component proofs , derivation of STDx specifications from other STDx specifications referring to the same SSL entity declaration, as well as for compositional proofs.

Proof-obligations are constructed according to user-selection of SSL configurations. According to user selection, the proof-manager constructs executable proof-scripts for the selected proof-obligations, which can be invoked immediately or with an arbitrary delay by the user and can later be re-run if the proof-result has been invalidated due to modification of the (sub-)component or STDx specification. If invoked, a proof-script automatically performs all necessary operations in order to verify whether the (sub-)component to which the script refers fulfills the corresponding STDx specification. Due to the famous mechanism of `make`, only the necessary operations have to be re-executed in case of an invalidation, in order to re-establish the proof-result. Interactive selections, such as variable selections for propositional abstraction are stored as data associated with a proof-obligation and can be reused in re-executions of proof-tasks.

The verification of designs often lead to complex proof states. Parts of the design have already been proven, other parts still have to be verified. According to the compositional proof rule, dependences among proofs have to be regarded for compositional verification using STDx. A compositional conclusion about a top-level specification can only be drawn if an implication involving sets of STDx specifications for sub-components can be proven to be a tautology, and if all sub-component specification referred to in the premise of the implication are satisfied by the sub-components. Hence, in contrast to the on-the-fly techniques of analyses, formal debugging and pattern-based verification, compositional proofs do not depend only on a single proof-task; but moreover depend on the set of sub-component proofs for STDx specifications referred to in the premises of the hierarchical conclusion.

During the various iteration phases of the development, parts of the design which have been proven correct may be changed by the developers. Since fulfillment of the involved sub-component specifications is essentially part of the compositional proof rule, book keeping of proof-results and dependence management have to be part of an automated environment for compositional reasoning. The implementation of dependence management has to keep track of the proof-results and to hierarchically invalidate dependent proofs when parts of the design or relevant sub-component specifications are modified. On the other hand, proof results which are independent from a modification have to remain valid and only these proof-tasks have to be re-executed which were affected by modifications in order to re-establish a compositional proof. Therefore all proofs are under control of a proof-manager.

The most complex proof situation, of course, arises during verification of hierarchical designs. Compositional proofs require at least two steps of verification activities. In the first step a a selection from the STDx specifications of a structural specification configuration is checked against a selected STDx specification of the considered activity. In the second step the selection of STDx specifications of its sub-activities has to be proved for their behavioral - or again decomposed - representations. This second task consists of several proof-tasks each checking a sub-activity against its STDx specifications. Due to dependence management and book-keeping of results, these steps

can be performed in any order. Hence, a top-down strategy as well as a bottom-up strategy can be applied, and also critical-first verification can be performed this way. Regardless, which of these strategies is preferable, the proof-manager ensures that in the end a compositional conclusion is justified only if all sub-tasks of the proof have been applied successfully.

Besides construction of proof-obligations and execution control for the corresponding proof-tasks, the proof-manager provides a complete overview about the proof-states of all parts of the design. In order to provide navigation of the design and to maintain the different views to the (sub-)components, the compositional representation of the design and the formal specifications referring to its (sub-)components are kept in a SSL design-database.

Witnesses for failed proofs can be directly visualized under control by the proof-manager as a set of waveforms. Furthermore, they can be exported to the workarea and run in the STATEMATE simulator.

7.2 Optimizations and Abstractions in the Verification of Statemate Models

Model checking has proven to be fairly well applicable in practice for models of non-trivial size. Many successful applications of model checking to industrial size models have been reported over the past years (for an overview cf. e.g. [CWA+96]). State-of-the-art model checkers are able to handle models of impressing complexity [BCM+90]. Although the symbolic representation of models using ROBDDs can efficiently handle models orders of magnitude larger than the explicit representation can, the inherent complexity of the model checking problem limits its feasibility in general.

To alleviate this problem, many optimization and abstraction techniques have been proposed and some have been applied very successfully in practice [CGP99, BCC+99].

In this section, we will briefly overview some techniques to cope with complexity problems in the verification of STATEMATE models. While some of these techniques are tailored to specific problems arising in the verification of asynchronous STATEMATE models, others are of rather basic nature and make no use of specific features of a particular representation.

Cone of Influence Reduction(COI) : Reactive systems usually consist of more than only one single function. Several computations reacting to subsets of the inputs each influence only a subset of the outputs. Computations running in parallel may interfere with each other but often this is not the case. Therefore, the main idea behind Cone of Influence reduction is that not all parts of a model contribute to the validity a particular specification. Only these inputs and variables can have an impact on the validity of a requirement, which are transitively connected to variables which are referred to in the specification [CGP99].

For verification purposes, the model can be reduced to only these parts which contribute to the property without doing harm to the validity of the specification. A functional dependency analysis determines which variables and control structures are potentially relevant to the validity of a specification. Computation paths irrelevant for the validity of a particular specification can be eliminated from the model. This COI reduction is an exact approximation of the model, i.e. both truth and falsehood of a specification are preserved.

COI reduction can be computed on several levels of model representation. E.g. COI reduction is part of state-of-the-art model checkers like VIS [VIS96a] or SMV [McM93], where it is

performed automatically and invisibly to the user as part of every verification task. Of course, the maximal advantage can be obtained from applying COI reduction already on a high level description of the model. By applying COI reduction to the SMI representation, we avoid the construction of an un-reduced finite state machine representation for the model checker.

Relaxed Cone of Influence Reduction(RCOI) [Bie03] : While COI reduction often yields drastic reductions of complexity when verifying synchronous STATEMATE models, the technique does not show this efficiency in the verification of asynchronous STATEMATE models. The problem with the asynchronous semantics is that all computations of the model depend on stabilization of the asynchronous model: The model is "glued together" by dynamic stabilization. New inputs are read and timer variables are modified only in stable states. On the other hand, the model becomes stable only if no transition can be taken without reading new inputs or observing a timeout event.
Following this observation, a variable v (or state) of an asynchronous model may belong to the COI of a specification for one of two reasons:

- v (in)directly influences variables which are referred to in the specification. Then, v has a functional dependence to a variable of the specification and would also belong to the COI in the synchronous semantics,
- v has no functional dependency with the variables of the specification and belongs to the COI only due to stabilization detection.

Now, RCOI reduction exploits this distinction by only considering the functional dependences and abstracting from the stabilization of an asynchronous model. Provided that the model always eventually becomes stable, the validity of a specification only depends on the functional dependencies of variables. Note that RCOI does not preserve super-step divergence of a model. If a diverging computation for a variable - not belonging to the functional cone of influence - is eliminated by RCOI reduction, the reduced model may stabilize, whereas the un-reduced model is divergent. Hence, it must be guaranteed that those portions of a model which are eliminated by RCOI reduction do not cause the model to diverge. For this purpose a dedicated stabilization check for asynchronous models if offered by the STVE. This check is required for an asynchronous model in order to justify application of RCOI reduction. Only provided that the model always stabilizes, RCOI is an exact approximation of an asynchronous model for a given specification[1].
RCOI reduction was formally introduced in [Bie03]. There also a proof of soundness as well as many experimental results can be found.

Propositional Abstraction : COI and RCOI reductions often drastically reduce the model complexity w.r.t. a specification. Since both reductions preserve the functional dependencies in a model M for specification ϕ, the reduced model may still contain complex computations. This complexity often remains too high for successful verification.
Although the COI consists of only these variables which may have an impact on the validity of ϕ, for some of them their concrete values may be of no relevance regarding ϕ.
If e.g. ϕ must hold for arbitrary values of a variable x, it may have no impact on the validity

[1] Since RCOI does not preserve the length of a super-step, RCOI is only applicable to specifications not referring to step counters - or `X`−operators.

of ϕ if x is assigned arbitrary values instead of the result of a complex computation. By abstracting from concrete computations for x, the complexity of model M can possibly be reduced drastically. If the concrete values for some variable x do not influence the validity of ϕ, also the computation of the values need not be considered for verification of ϕ: After e.g. assigning x with a new input instead of a computed value, COI reduction can eliminate the computation from the abstract model.
Often not only application of model checking suffers from the complexity of the exact model, but also the construction of the finite state machine representation for the model checker is impossible, e.g. if the model contains infinite data-objects like variables of real number domains. Thus, in practice abstractions are useful only if applicable to a syntactical representation of a system before building its semantical representation M.
Propositional abstraction - as offered by the STVE [BDW00, BBD+99] - provides a mechanism to automatically compute abstraction M_A of M w.r.t. a user selected set of variables. The user selected variables are treated abstractly and only their influence on other objects is maintained in M_A.
Propositional abstraction is offered with two different abstraction granularities in the STVE:

- The more fine grained abstraction simply *frees* x, by making x an input. All assignments to x are eliminated.
- The coarser abstraction - referred to as *strong* - abstracts from variable x of model M by *approximative existential quantification* - in a pure syntactic fashion. In order to approximate $\exists x.b$, each condition b in which x occurs is transformed. This is achieved by replacing the least boolean term of b containing x is by either $true$ or $false$, depending on the polarity of x's occurrence. For example, let x be a variable to be abstracted from, then $(y = 7) \wedge (x = z)$ is replaced by $(y = 7) \wedge true$, $(y = 7) \wedge \neg(x = z)$ is replaced by $(y = 7) \wedge \neg false$, $\neg((y = 7) \wedge \neg(x = z))$ is replaced by $\neg((y = 7) \wedge \neg true)$, and so on. Approximative existential quantification possibly enables more transitions for every step, than in the un-abstracted model. In order to be able to also stay in a state for which no transition is enabled in the concrete model, non-exclusive self loops are added to the source states of transitions whose triggers are affected by applying abstraction. Staying in the source state or taking an outgoing transition can now non-deterministically be chosen.
  As in the fine grained abstraction, assignments to x are eliminated. If x occurs on the right hand side of an assignments to a non-abstract variable y, the entire right hand side expression of the assignment is replaced by a fresh input.
  By eliminating abstracted variables entirely from the model, the coarse abstraction is also suitable for models containing infinite objects, such as real variables. When abstracting from all variables of infinite domains, the resulting abstract model M_A is a finite model.

Propositional abstraction is an over-approximation in both offered granularities, whereat strong abstraction over-approximates freeing. Over-approximations are only applicable for verification of universal specifications, i.e. specifications which express requirements for all runs of a model. If an over-approximated abstract model fulfills a universal requirement, then also the concrete model fulfills the requirement. Since over-approximations preserve only truth of universal specifications, in case of a violated universal specification, no conclusion to the concrete model is possible.

Details regarding the above abstraction techniques and their realization can be found in the dissertation of Tom Bienmüller [Bie03]. Also presented there is an application methodology for *automatic abstraction refinement*. In this approach COI computation and propositional abstraction are combined in order to automatically compute a series of abstractions. In an automated verification process - starting with a very coarse propositional abstraction - the abstraction is iteratedly refined unless the requirement is fulfilled by one of the abstract models of the series. Otherwise, the iteration terminates if no further abstraction can be found and the un-abstracted model also violates the requirement.
This automatic abstraction refinement verification process is integrated with the STVE and offered in the context of pattern verification.

Counter Abstraction of Asynchronous Stabilization RCOI abstracts from stabilization dependences among the variables of an asynchronous model in order to compute its functional cone of influence for a given specification, but the reduced model still contains the dynamic stabilization detection.
A counter which triggers stabilization of super-steps with a user-defined length can be used to abstract from dynamic stabilization. Using counter abstraction, stabilization depends no longer on internal computations of the model, while reading of inputs and modifying timer variables remains dependent on stabilization. After computing all reactions on inputs and timeouts, the model performs idle steps until the next counter-triggered stabilization, if the counter is chosen large enough. In order to determine an upper bound for the length of super-steps, the verification environment offers a dedicated stabilization check. In general, such an upper bound can *only be established on top-level* of an asynchronous STATEMATE model. If the interface of a model contains *fast* inputs, they can change their values and the model reacts to these changes independent of counter-based stabilization. Thus, counter abstraction should not be applied to sub-component models considered in isolation
Provided that specification of interest does not count steps of the model and the counter is defined according to an existing - and verified - upper bound, and provided that the model has no fast inputs, counter abstraction is an exact approximation. Otherwise, counter abstraction is just an approximation: the model stabilizes disregarding changes to fast inputs. Since the un-abstracted model does not stabilize when any transition is enabled, a run of the abstract model is not necessarily a run of the concrete model.

Freezing of Inputs When verifying open embedded systems, one often needs to express assumptions about the environment in order to focus on particular functionality of the model. The STVE offers different techniques to express assumptions about the environment of the model under verification. For many use-cases, assumptions about the environment are of rather simple character. For example, one often wants to disable all inputs modeling error indications from external devices, in order to examine the model under *normal conditions*[2]. In this case, where particular inputs are assumed to have fixed values, freezing can be applied.
Freezing sets inputs to user-defined constant values. The impact of this modification on internal computations is analyzed by a data-flow analysis and those parts of the model which

[2] In STATEMATE, model simulations are often driven by user-defined simulation panels. Since these panels are not part of the model, they are omitted in the SMI representation. Hence, inputs connected to a panel appear as free inputs in the SMI representation.
Often panels are also used to preselect values for inputs of the model which are understood by the system designer as model parameters rather than as inputs.

have become unreachable due to the modification are eliminated. The abstract model has strictly less runs than the concrete model. Freezing is an under-approximation, i.e. each run of the abstract model is always also a run of the concrete model.

Approximations of Non-deterministic Choices with Deterministic Choices The execution semantics of STATEMATE permits non-deterministic choice of enabled transitions. In order to preserve non-determinism in the translation of STATEMATE models into SMI, all outgoing transitions of a state are represented as non-deterministic choices, regardless if some of these transitions - or all of them - are mutual exclusive. An elaborate determinization procedure - partly interpreting the transition triggers using BDDs - transforms the non-deterministic choice into a smaller choice of the potentially conflicting transitions as well as a deterministic choice for all mutually exclusive transitions. Whether the remaining non-deterministic choices are real ones or at most one guard can be become true at each step can be decided only by an own verification task. Since non-determinism is resolved by introducing one input per possible decision, non-determinism is a source of verification complexity. By simply transforming all non-deterministic choices of a model into deterministic ones, an *under-approximation* can be applied. Each run of the under-approximated model is also a run of the concrete model.

All the techniques listed so far attempt to reduce verification complexity by model transformations. Instead of applying the verification task to the concrete model, an abstract model is constructed and verification is applied to this abstract model. In section 4.10 abstractions have been classified regarding preservation of falsehood and truth of verification result w.r.t. the concrete model. For each of the techniques listed above we have stated whether its application is an under- or over-approximation, or does neither preserve validity nor violation of specifications in general. It depends on this classification whether this technique is applicable to a particular verification use-case: In general, *under-approximations are appropriate for falsification*, i.e. application of verification aimed at obtaining a witness trace for reachability of a particular configuration of the model. If this specified configuration is reachable in the under-approximated model, it is also reachable in the concrete model. In contrast to under-approximations, *over-approximations are appropriate when establishing satisfaction* of a specification. Over-approximations extend the possible behavior of a model. Hence, if the over-approximated model fulfills a requirement, the concrete model fulfills the requirement anyway. Abstractions neither preserving falsehood nor truth of verification results essentially do not fit in either use-case. In principle, results obtained using such an abstraction are inconclusive regarding the concrete model. At least for violated requirements, the result can be validated for the concrete model by *concretizing the witness trace* (this concretization is not performed for propositional abstraction[3])*:*

In order to justify application of an approximation, the STVE applies for each model transformation applied during preparation for verification the reverse modification to the witness trace. A witness trace obtained from verification of the abstract model, is used to simulate the concrete model according to the stimuli as recorded in the abstract trace. This simulation is only successful, if the concrete model

* (in case of counter abstraction) always stabilizes within the counter bound for the same inputs as recorded in the abstract trace ,

[3]Propositional abstraction abstracts from variables, which may have an impact also on the timing of reactions and on the synchronization between components of a model. In general, abstract witnesses obtained for the violation of requirements using propositional abstraction can not be concretized.

* (in case of COI and RCOI) computes the same values of variables in the cone of influence as recorded in the abstract trace,

If an abstract witness trace can not be concretized in this manner, the result of an verification task remains inconclusive. Otherwise, application of the particular approximation is justified.

In addition to application of model abstractions, also verification methodology according to the following recommendations can avoid undesired verification complexity:

Usage of appropriate Verification Engine In practice, bounded model checking is often much faster in finding violations of specifications than model checking or invariance checking are. On the other hand, bounded model checking is only complete w.r.t. a user defined depth k, i.e. bounded model checking can decide only whether a specification is valid for all runs of the model up to a length of k step, but is in general not able to decide whether a specification holds on all possible runs of arbitrary length.
The STVE is integrated with both a conventional model checker (the model checker VIS [VIS96a, VIS96b]), which is capable also of invariance checking, and a bounded model checker (based on the SAT solver Prover Plug-In, trademark of Prover Technologies AB in Sweden, the United States and other countries). By selecting the verification engine according to the expected result, the most appropriate verification engine for the particular use-case can be chosen. If in a particular use-case a violation of the specification is more likely than a true result, invocation of bounded model checking engine is suggested. This is obviously the case for formal debugging, but can also be advantageous for robustness analyses if the potential conflict is expected to be a real one. Otherwise, if the verification use-case requires a complete decision procedure, as for observer pattern-based verification, invariance checking is preselected. For the verification using STDx only model checking or invariance checking are applied.

Application of Verification to the least Scope A key advantage of the SMI/SSL representation according to the compositional semantics for Statemate models [DJHP97] is that sub-activities of a system can be considered in isolation. Due to the concept of interfaces - derived from analysis of data-flows - a sub-activity representation always takes all possible interactions with its environment into account. E.g. if a data-item can be written outside a particular component, this component is equipped with an input that enables the environment to interfere with local computations of the component. Hence, many properties can be checked for the component in isolation without considering the concrete representation of the surrounding system model. For example, whether two potentially conflicting transition can ever be enabled at the same step can often be decided on the sub-activity model to which both transitions belong. If this conflict can be ruled out for this scope, no further consideration of a more complex scope is necessary.

Compositional Verification Results of verification tasks for sub-activity models can be exploited hierarchically. In a modular way, model checking is applied to verify fulfillment of specifications for sub-activities of a system. A specification for the composed system is then derived from the specifications of its sub-activities using proof rules. Usually the size of a composed system is given by the product of the size of its components. Therefore deriving a system specification from the specifications of its sub-activities according to a proof-rule avoids considering the complex model representation of the entire system. For a compositional proof only the involved specifications and the structural mapping of the interfaces need to be considered. Hence, the

complexity of compositional proofs using tautology checking only depends on the complexity of the involved specifications. In particular, the complexity is independent of the model complexity.

7.3 Compositional Verification

Often, specification verification for entire systems suffers from the complexity of the system model. One idea to overcome this 'state explosion problem' is to verify specifications of sub-components of the system and then to conclude the validity of a system specification from fulfillment of the sub-component specifications. If it can be proved that the implication of the system specification by a set of fulfilled sub-component specifications is a tautology, the complexity of a verification task can be reduced to a number of localized, smaller verification tasks.

Each sub-component requirement of the system is specified in terms of assumption/commitment specifications, i.e. the respective sub-component guarantees its commitments only provided that its environment adheres to the assumptions.

A main task in assumption/commitment style compositional verification is the substitution or elimination of local assumptions. If a specification of sub-component B assumes correct behavior of sub-component A, and A guarantees that this assumption is fulfilled, then we would like to get rid of the assumption of B. In practice, when considering the composition of component A with component B, it is often necessary to assume properties of A for verification of B and vice versa. A severe problem arising in this situation is that A sometimes can not guarantee the assumptions of B without the guarantee of B to adhere to A's assumptions. A small example illustrates this situation:

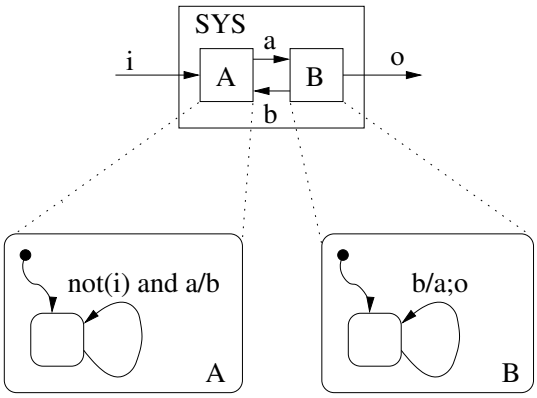


Figure 7.2: Simple Circular Dependency

Let SYS in figure 7.2 be an activity, which consists of the sub-activities A and B, where A reacts on event a with emission of event b if event i is absent. Statechart B reacts on event b with emission of events a and event o. Let $\mathcal{C}_A$ (and accordingly for B) denote the CSTS representation of A, and $\mathcal{K}(\mathcal{C}_A)$ the Kripke structure for $\mathcal{C}_A$ according to definition 5.5.

For statechart A, one can verify that the LTL assumption/commitment specification

$$\mathcal{K}(\mathcal{C}_A) \models \underbrace{(\mathtt{G}(\neg i) \wedge \mathtt{GF}(a))}_{=:a_A} \Rightarrow \underbrace{\mathtt{GF}(b)}_{=:c_A}$$

is a valid specification. For statechart B it can be shown that

$$\mathcal{K}(\mathcal{C}_B) \models \underbrace{\mathtt{GF}(b)}_{=:a_B} \Rightarrow \underbrace{\mathtt{GF}(a \wedge o)}_{=:c_B}.$$

Now we can prove that if SYS is never provided with event i, the commitment of A guarantees the assumption of B and vice versa. Hence, with $a_{SYS} := \mathtt{G}(\neg i)$ we obtain:

$$\models (a_{SYS} \wedge c_A) \Rightarrow a_B \quad \text{and}$$
$$\models (a_{SYS} \wedge c_B) \Rightarrow a_A$$

Using these tautologies, a naive proof-rule for the composed system could state (cf. [Jos93]):

$$\frac{\begin{array}{c}\mathcal{K}(\mathcal{C}_A) \models \underbrace{(\mathtt{G}(\neg i) \wedge \mathtt{GF}(a))}_{a_A} \Rightarrow \underbrace{\mathtt{GF}(b)}_{c_A} \\ \models (\underbrace{\mathtt{G}(\neg i)}_{a_{SYS}} \wedge \underbrace{\mathtt{GF}(a \wedge o)}_{c_B}) \Rightarrow \underbrace{(\mathtt{G}(\neg i) \wedge \mathtt{GF}(a))}_{a_A}\end{array} \qquad \begin{array}{c}\mathcal{K}(\mathcal{C}_B) \models \underbrace{\mathtt{GF}(b)}_{a_B} \Rightarrow \underbrace{\mathtt{GF}(a \wedge o)}_{c_B} \\ \models (\underbrace{\mathtt{G}(\neg i)}_{a_{SYS}} \wedge \underbrace{\mathtt{GF}(b)}_{c_A}) \Rightarrow \underbrace{\mathtt{GF}(b)}_{a_B}\end{array}}{\mathcal{K}(\mathcal{C}_{(A||B)}) \models (\underbrace{\mathtt{G}(\neg i)}_{a_{SYS}} \Rightarrow (\underbrace{\mathtt{GF}(b)}_{c_A} \wedge \underbrace{\mathtt{GF}(a \wedge o)}_{c_B}))}$$

Obviously, this naive proof-rule is unsound for *liveness* specifications regarding mutual assumptions. Circular dependencies - regarding liveness - can only be handled using induction techniques, as explained in detail for example in [Jos93] and [McM99]. Fortunately, the naive proof-rule is sound if only safety properties are considered [Pnu85, Jos93] - as it is the case for non-failure acceptance observers. Hence, the rule can be applied to compositional verification using Symbolic Timing Diagrams, which are restricted according to lemma 6.13, i.e. diagrams which are deterministic and globally constrained. Since every reference to the future of another component's behavior is limited with an upper bound, the verification procedure itself is inductive (we refer to the explanation of invariance checking in section 4.7). Notice, that soundness means that proof-obligations involving mutual assumptions can not be proved successfully. Thus, we have to require that circular dependences are broken up.

In this section we present a automatic technique for compositional reasoning about requirements based on a compositional rule. If the representation is reasonably small, fulfillment of the system specification can be concluded automatically from fulfillment of the component specifications. The presented proof rule may suffer from the complexity of the involved specification. If the complexity of the involved specifications becomes too large, user interaction is necessary in order to substitute local assumptions with local commitments of other sub-components. The proof of validity for these substitutions becomes part of the proof-obligation and is smoothly integratable with the automated proof-task.

The definitions and conclusions of this section do not hold for parallel composition of CSTS in general, but only provided that the considered CSTS adhere to fact 5.1(cf. page 68). Recall from section 5.2 that parallel composition of CSTS does in general not preserve consistency and viability.

In order to discuss some basic rules regarding the composition of CSTS w.r.t. verification using synchronous observers and to present the compositional proof rule, we first have to define a more concise notation than those that we have used in the previous sections:

Definition 7.1 (Concise Notation for Observers of Diagrams)
For a set D of deterministic and globally constrained diagrams let $\|\!\|_{d\in D} d$ denote the parallel composition of all observers obtained from the unwinding representations of $d \in D$:

$$\|\!\|_{d\in D} d \quad := \quad ||_{\Omega}\ _{d\in D} (\Omega(\mathcal{U}(d)) \triangleright o_d)$$

Accordingly, for diagrams d_1, d_2 let $d_1 \| d_2$ denote the parallel composition

$$d_1 \| d_2 := (\Omega(\mathcal{U}(d_1)) \triangleright o_{d_1}) || _{\Omega} (\Omega(\mathcal{U}(d_2)) \triangleright o_{d_2})$$

For CSTS $\mathcal{C} = (V, \Theta, \rho, E)$, set A_1 of assumption diagrams and set C_1 of commitment diagrams that all refer to the externally observable variables E of $\mathcal{C}$, let

$$\mathcal{C} \quad \models_{\Omega} \quad (\|\!\|_{a\in A_1} a, \|\!\|_{c\in C_1} c)$$

denote that $\mathcal{C}$ satisfies the conjunction of commitment diagrams in C_1 provided that the environment satisfies the conjunction of assumption diagrams in A_1, i.e. :

$$\begin{aligned}\mathcal{C} \quad \models_{\Omega} \quad (\|\!\|_{a\in A_1} a, \|\!\|_{c\in C_1} c) \quad :\Leftrightarrow \quad & \\ & \mathcal{K}\left(\mathcal{C} || _{\Omega} \left(||_{\Omega}\ _{a\in A_1} (\Omega(\mathcal{U}(a)) \triangleright o_a)\right) || _{\Omega} \left(||_{\Omega}\ _{c\in C_1} (\Omega(\mathcal{U}(c)) \triangleright o_c)\right)\right) \\ & \models \left(\bigwedge_{a\in A_1} \mathtt{G}(o_a) \Rightarrow \bigwedge_{c\in C_1} \mathtt{G}(o_c)\right)\end{aligned}$$

Accordingly, for sets A_1, A_2 of assumption diagrams and sets C_1, C_2 of commitment diagrams, let

$$\begin{aligned}\mathcal{C} \quad \models_{\Omega} \quad \left((\|\!\|_{a\in A_1} a_1, \|\!\|_{c\in C_1} c_1) \quad \wedge \ (\|\!\|_{a_2\in A_2} a_2, \|\!\|_{c_2\in C_2} c_2)\right) \quad :\Leftrightarrow \quad & \\ & \mathcal{K}\Bigg(\mathcal{C} || _{\Omega} \left(||_{\Omega}\ _{a_1\in A_1} (\Omega(\mathcal{U}(a_1)) \triangleright o_{a_1})\right) || _{\Omega} \left(||_{\Omega}\ _{c_1\in C_1} (\Omega(\mathcal{U}(c_1)) \triangleright o_{c_1})\right) \\ & || _{\Omega} \left(||_{\Omega}\ _{a_2\in A_2} (\Omega(\mathcal{U}(a_2)) \triangleright o_{a_2})\right) || _{\Omega} \left(||_{\Omega}\ _{c_2\in C_2} (\Omega(\mathcal{U}(c_2)) \triangleright o_{c_2})\right)\Bigg) \\ & \models \left(\bigwedge_{a_1\in A_1} \mathtt{G}(o_{a_1}) \Rightarrow \bigwedge_{c_1\in C_1} \mathtt{G}(o_{c_1})\right) \wedge \left(\bigwedge_{a_2\in A_2} \mathtt{G}(o_{a_2}) \Rightarrow \bigwedge_{c_2\in C_2} \mathtt{G}(o_{c_2})\right)\end{aligned}$$

Let furthermore for sets D_1, D_2 of diagrams $(\|\!\|_{d_1\in D_1} d_1) \quad \Rightarrow_{\Omega} \quad (\|\!\|_{d_2\in D_2} D_2)$ be a concise notation according to:

$$\left(\mathop{\|}_{d_1 \in D_1} d_1\right) \Rightarrow_\Omega \left(\mathop{\|}_{d_2 \in D_2} d_2\right) :\Leftrightarrow$$

$$\mathcal{K}\left(\left(\mathop{||_\Omega}_{d_1 \in D_1} (\Omega(\mathcal{U}(d_1)) \rhd o_{d_1})\right) ||_\Omega \left(\mathop{||_\Omega}_{d_2 \in D_2} (\Omega(\mathcal{U}(d_2)) \rhd o_{d_2})\right)\right)$$
$$\models \left(\mathop{\wedge}_{d_1 \in D_1} \mathtt{G}(o_{d_1}) \Rightarrow \mathop{\wedge}_{d_2 \in D_2} \mathtt{G}(o_{d_2})\right)$$

Let $(true)$ denote the situation that no observer is involved. Hence,

$$(true) \Rightarrow_\Omega (c) :\Leftrightarrow (\Omega(\mathcal{U}(c)) \rhd o_c) \models \mathtt{G}(o_c)$$

■

We first consider some basic features for the verification using synchronous observers:

Lemma 7.1 (Basic Rules)
The following rules apply to deterministic and globally constrained diagrams:

Basic Conjunction Rule I:

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$, assumption diagram a and commitment diagrams c_1, c_2 which all refer to the externally visible variables E of $\mathcal{C}$.

$$\frac{\mathcal{C} \models_\Omega (a, c_1) \qquad \mathcal{C} \models_\Omega (a, c_2)}{\mathcal{C} \models_\Omega (a, c_1 \| c_2)}$$

Basic Conjunction Rule II:

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$, assumption diagrams a_1, a_2 and commitment diagrams c_1, c_2 which all refer to the externally visible variables E of $\mathcal{C}$.

$$\frac{\mathcal{C} \models_\Omega (a_1, c_1) \qquad \mathcal{C} \models_\Omega (a_2, c_2)}{\mathcal{C} \models_\Omega (a_1 \| a_2, c_1 \| c_2)}$$

Weakening Rule

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$, assumption diagrams a_s, a_w and commitment diagrams c which all refer to the externally visible variables E of $\mathcal{C}$

$$\frac{\begin{array}{c}\mathcal{C} \models_\Omega (a_s, c) \\ (a_w) \Rightarrow_\Omega (a_s)\end{array}}{\mathcal{C} \models_\Omega (a_w, c)}$$

Embedding Rule

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E_{in} \cup E_{out})$, assumption diagram a and commitment diagram c which both refer to the external visible variables E of $\mathcal{C}$. Furthermore, let $\mathcal{C}_x = (V_x, \Theta_x, \rho_x, E_{in_x} \cup E_{out_x})$ be a second CSTS, such that $E_{out} \cap E_{out_x} = \emptyset$.

$$\frac{\mathcal{C} \models_\Omega (a, c)}{(\mathcal{C} || \mathcal{C}_x) \models_\Omega (a, c)}$$

Assumption Elimination Rule

Given a CSTS $\mathcal{C} = (V, \Theta, \rho, E)$ *, assumption diagrams* a_1, a_2 *and commitment diagram* c *which all refer to the external visible variables* E *of* $\mathcal{C}$*.*

$$\frac{\mathcal{C} \models_\Omega (a_1 \| a_2, c) \quad (a_1) \Rightarrow_\Omega (a_2)}{\mathcal{C} \models_\Omega (a_1, c)}$$

■

Proof 7.1

Basic Conjunction Rule I: Follows immediately from definitions 7.1, 6.25 and 5.7:
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c_1))) \bigwedge$
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c_2))) \bigwedge$
$\Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a)) \Rightarrow (ts \in (\mathcal{L}(\mathcal{U}(c_1) \cap \mathcal{L}(\mathcal{U}(c_2))))$

Basic Conjunction Rule II: Follows immediately from definitions 7.1, 6.25 and 5.7:
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_1)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c_1))) \bigwedge$
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_2)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c_2)))$
$\Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ((ts \in (\mathcal{L}(\mathcal{U}(a_1) \cap \mathcal{L}(\mathcal{U}(a_2))) \Rightarrow (ts \in (\mathcal{L}(\mathcal{U}(c_1) \cap \mathcal{L}(\mathcal{U}(c_2))))$
Notice, that $(\mathcal{L}(\mathcal{U}(a_1) \cap \mathcal{L}(\mathcal{U}(a_2))$ may be empty.

Weakening Rule: Follows immediately from definitions 7.1, 6.25 and 5.7 and lemma 6.9:
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_w)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(a_s))) \bigwedge$
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_s)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c)))$
$\Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ((ts \in (\mathcal{L}(\mathcal{U}(a_w) \cap \mathcal{L}(\mathcal{U}(a_s))) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c)))$,
because $\mathcal{L}(\mathcal{U}(a_w) \subseteq \mathcal{L}(\mathcal{U}(a_s)$.
Notice, that the weakening rule also holds, if $\mathcal{L}(\mathcal{U}(a_w) = \emptyset$.

Embedding Rule: Trivial.

Assumption Elimination Rule: Follows immediately from definitions 7.1, 6.25 and 5.7 and lemma 6.9 :
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_1)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c))) \bigwedge$
$\forall ts \in TComps(\mathcal{C}) : ((ts \in \mathcal{L}(\mathcal{U}(a_2)) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c)))$
$\Rightarrow \quad \forall ts \in TComps(\mathcal{C}) : ((ts \in (\mathcal{L}(\mathcal{U}(a_1) \cap \mathcal{L}(\mathcal{U}(a_2))) \Rightarrow (ts \in \mathcal{L}(\mathcal{U}(c)))$

□

According to the embedding rule and the basic conjunction rule II, the parallel composition of two non-interfering CSTS satisfies specifications, which are fulfilled by the individual transition systems independent of the parallel composition:

Lemma 7.2 (Composition Rule)
Given $\mathcal{C}_1 = (V_1, \Theta_1, \rho_1, E_1)$ *and* $\mathcal{C}_2 = (V_2, \Theta_2, \rho_2, E_2)$ *representing sibling Statemate activities* A_1 *and* A_2 *according to the compositional semantics, s.t.* $E_{1_{out}} \cap E_{2_{out}} = \emptyset$*. For assumption diagram* a_1 *and commitment diagram* c_1 *which both refer to the external visible variables* E_1 *of* $\mathcal{C}_1$ *and assumption diagram* a_2 *and commitment diagram* c_2 *which both refer to the external visible variables* E_2 *of* $\mathcal{C}_2$ *holds (provided that* a_1, a_2, c_1, c_2 *are deterministic and globally constrained):*

$$\frac{\mathcal{C}_1 \models_\Omega (a_1, c_1) \qquad \mathcal{C}_2 \models_\Omega (a_2, c_2)}{(\mathcal{C}_1 || \mathcal{C}_2) \models_\Omega ((a_1 \| a_2), (c_1 \| c_2))}$$

■

Proof 7.2
The proof of **Composition Rule** follows from the **Embedding Rule** and **Basic Conjunction Rule II** of lemma 7.1.

□

Here is, where circular dependences come into play - the question is : is $a_1 \wedge a_2$ satisfiable? Are a_1 and a_2 mutual assumptions about $\mathcal{C}_2$ and $\mathcal{C}_1$, respectively, or is at least one of a_1 and a_2 an assumption about the environment of $(\mathcal{C}_1 || \mathcal{C}_2)$. The composition rule holds, even though $(a_1 \| a_2)$ might assume a miracle. If w.l.o.g. fulfillment of a_1 depends on validity of c_2, then the composition rule can be satisfied even though a_1 is violated because c_2 does not hold. In order to derive a system property from the local commitments, we hence need a stronger rule than this composition rule. In general, the local assumptions need not be independent from the local commitments of the other sub-components. The following compositional proof rule allows the conclusion of a system property from local specifications.

Definition 7.2 (Compositional Proof Rule)
Let $\mathcal{C}_1, \mathcal{C}_2$ be two CSTS. Let $spec_1, spec_2$ be symbolic timing diagram specifications, s.t. $spec_1$ refers to the observables of $\mathcal{C}_1$ and $spec_2$ refers to the observables of $\mathcal{C}_2$. Assumed that all considered diagrams are deterministic and globally constrained, we define our compositional proof rule for a specification $spec$ of the composed system by[4]:

$$\frac{\begin{array}{l}\mathcal{C}_1 \models_\Omega \left(\|_{a_1 \in ad(spec_1)} a_1, \|_{c_1 \in cd(spec_1)} c_1 \right) \qquad \mathcal{C}_2 \models_\Omega \left(\|_{a_2 \in ad(spec_2)} a_2, \|_{c_2 \in cd(spec_2)} c_2 \right) \\ \models \left(\left(\left(\|_{a_1 \in ad(spec_1)} a_1 \right) \Rightarrow_\Omega \left(\|_{c_1 \in cd(spec_1)} c_1 \right) \right) \wedge \left(\left(\|_{a_2 \in ad(spec_2)} a_2 \right) \Rightarrow_\Omega \left(\|_{c_2 \in cd(spec_2)} c_2 \right) \right) \right. \\ \qquad \left. \wedge \left(\|_{a \in ad(spec)} a \right) \right) \Rightarrow_\Omega \left(\|_{c \in cd(spec)} c \right)\end{array}}{(\mathcal{C}_1 || \mathcal{C}_2) \models_\Omega \left(\|_{a \in ad(spec)} a, \|_{c \in cd(spec)} c \right)}$$

■

Lemma 7.3 (Soundness of Compositional Proof Rule)
The compositional proof rule defined in definition 7.2 is sound.

■

Proof 7.3 The structure of the premise reflects the fact, that the subcomponents have been proven to fulfill their commitments only provided that their respective assumptions hold. Let $\mathcal{L}_{prem}$ be the set of all timed observation sequences that are accepted by the premise:

[4] Let $ad(spec)$ refer to the assumption diagrams instantiated by $spec$ and $cd(spec)$ refer to the commitment diagrams instantiated by $spec$ as defined in definition 6.30.

$$\mathcal{L}_{prem} := \left\{ ts \mid \left(ts \notin \bigcap_{a_1 \in ad(spec_1)} \mathcal{L}(\mathcal{U}(a_1)) \vee ts \in \bigcap_{c_1 \in cd(spec_1)} \mathcal{L}(\mathcal{U}(c_1)) \right) \left(ts \notin \bigcap_{a_2 \in ad(spec_2)} \mathcal{L}(\mathcal{U}(a_2)) \vee ts \in \bigcap_{c_2 \in cd(spec_2)} \mathcal{L}(\mathcal{U}(c_2)) \right) ts \in \bigcap_{a \in ad(spec)} \mathcal{L}(\mathcal{U}(a)) \right\}$$

Note that this set is in general not the same as

$$\left\{ ts \mid \left(ts \in \bigcap_{c_1 \in cd(spec_1)} \mathcal{L}(\mathcal{U}(c_1)) \wedge ts \in \bigcap_{c_2 \in cd(spec_2)} \mathcal{L}(\mathcal{U}(c_2)) \wedge ts \in \bigcap_{a \in ad(spec)} \mathcal{L}(\mathcal{U}(a)) \right) \right\} \tag{7.1}$$

The contribution of a sub-component commitment to the acceptance of $\mathcal{L}_{prem}$ depends on whether its assumptions are fulfilled. $\mathcal{L}_{prem}$ is equal to set (7.1) only if all sub-component assumptions are fulfilled. Hence,

$$\left(\left(\left(\|_{a_1 \in ad(spec_1)} a_1 \right) \Rightarrow_\Omega \left(\|_{c_1 \in cd(spec_1)} c_1 \right) \right) \wedge \left(\left(\|_{a_2 \in ad(spec_2)} a_2 \right) \Rightarrow_\Omega \left(\|_{c_2 \in cd(spec_2)} c_2 \right) \right) \wedge \left(\|_{a \in ad(spec)} a \right) \right) \Rightarrow_\Omega \left(\|_{c \in cd(spec)} c \right)$$

can only be proven to be a tautology, if all *relevant* local assumptions are fulfilled by sub-component commitments or by top-level assumptions, because sub-component commitments can only contribute to the validity of the top-level commitment if their respective assumptions hold.

W.l.o.g., assume that some assumption $a_x \in ad(spec_1)$ of $\mathcal{C}_1$ is violated. Then the term

$$\left(\|_{a_1 \in ad(spec_1)} a_1 \Rightarrow_\Omega \|_{c_1 \in cd(spec_1)} c_1 \right)$$

is trivially satisfied and $\left\{ ts \in \bigcap_{c_1 \in cd(spec_1)} \mathcal{L}(\mathcal{U}(c_1)) \right\}$ does not restrict $\mathcal{L}_{prem}$. Moreover, assume that all assumptions in $ad(spec_2)$ of $\mathcal{C}_2$ are satisfied. Then $\mathcal{L}_{prem}$ would be equal to the set

$$\left\{ ts \mid \left(ts \in \bigcap_{c_2 \in cd(spec_2)} \mathcal{L}(\mathcal{U}(c_2)) \wedge ts \in \bigcap_{a \in ad(spec)} \mathcal{L}(\mathcal{U}(a)) \right) \right\}$$

The conjunction of all sub-component specifications with the top-level assumptions can only imply the top-level commitment, i.e. $\mathcal{L}_{prem} \subseteq \bigcap_{c \in cd(spec)} \mathcal{L}(\mathcal{U}(c))$, if all *relevant* local assumptions are satisfied.

□

This way, the compositional proof rule covers two verification tasks in one sweep. On the one hand it is checked whether the sub-component assumptions are fulfilled in the parallel composition of the components. On the other hand it is cheked whether the top-level commitment follows from a set of sub-component specifications.

Invariance checking is based on reachability computation, i.e. acceptance of all observers is considered stepwise; circular dependences within the premise of the implication will not lead to a *true*-result. If all local assumptions are fulfilled, by local commitments of other sub-components or by top-level assumptions, then fulfillment of the top-level commitment only depends on the top-level assumptions and the local sub-component commitments.

The compositional proof rule reflects the fact that fulfillment of a system requirement can only be concluded from fulfilled sub-component specifications. If local assumptions are violated or are mutually exclusive, then the implication in the premise of the proof rule will in general not hold. Only if a sub-component specification is entirely irrelevant for the proof, its local assumptions may be violated without having an effect on the result.

According to the definition of observers, all observables regarded by the individual specifications are inputs to the parallel composition of their observers. Hence, the overall implication can only be true, if these inputs are guessed according to the requirements of the sub-components, provided that their respective assumption are fulfilled. If one of the local assumptions is not satisfied, then the respective commitment need not hold, and hence will not contribute to the fulfillment of the top-level commitment.

The compositional proof rule handles the composition of two components. It has an obvious and natural generalization to the composition of n components, with $n \geq 2$. We omit a formal definition of this extension due to limited space.

Definition 7.3 (Assumption Substitution in Compositional Proof Rule)
W.l.o.g. let $c_x \in cd(spec_2)$ and $a_z \in ad(spec_1)$, s.t.

$$\models (c_x \Rightarrow_\Omega a_z)$$

Then, a_z can be substituted by c_x in the implication of the compositional rule and instead of the original implication it can be checked, whether

$$\begin{aligned}\Bigg(\Bigg(\Bigg(\Bigg(\Bigg(\Bigg\|_{a_1 \in (ad(spec_1) \setminus a_z)} a_1\Bigg) \| c_x\Bigg) \Rightarrow_\Omega \Bigg(\Bigg\|_{c_1 \in cd(spec_1)} c_1\Bigg)\Bigg) \\ \wedge \Bigg(\Bigg(\Bigg\|_{a_2 \in ad(spec_2)} a_2\Bigg) \Rightarrow_\Omega \Bigg(\Bigg\|_{c_2 \in cd(spec_2)} c_2\Bigg)\Bigg)\Bigg) \\ \wedge \Bigg(\Bigg\|_{a \in ad(spec)} a\Bigg)\Bigg) \Rightarrow_\Omega \Bigg(\Bigg\|_{c \in cd(spec)} c\Bigg)\end{aligned}$$

is a tautology.

■

Lemma 7.4 (Soundness of Assumption Substitution)
Assumption substitution according to definition 7.3 is sound.

■

Proof 7.4 Soundness follows immediately from the weakening rule of lemma 7.1.

If c_x and a_z are involved in a circular dependence ($a_x \Rightarrow_\Omega c_x$ and $a_z \Rightarrow_\Omega c_z$, where $c_z \Rightarrow_\Omega a_x$), then assumption substitution preserves the dependence.

If there exist sub-component specifications[5] $spec_x \neq spec_y$, s.t. $c_x \in cd(spec_x)$ and $c_y \in cd(spec_y)$, with $\models (c_x \Rightarrow_\Omega a_z)$ and $\models (c_y \Rightarrow_\Omega a_z)$, then it depends on the choice of c_x or c_y, whether assumption substitution breaks up or preserves circular dependences.

□

Assumption substitution can be applied not only using single local commitments. Of course, also top-level assumptions can be used as substitutes for local assumptions, and also conjunctions of local commitments and top-level assumptions.

Each of the implications used for substitution becomes part of the proof-obligation: Before checking whether the top-level commitments can be derived from the top-level assumptions and the local specifications after assumption substitution, it can automatically be checked whether the substitutions are justified. In the application examples in section 8.3 substitutions by single diagrams as well as by combinations of top-level assumptions and local commitments are applied.

This way, the user has to guess only substitutions, while their justification is automatically checked. Guessing useful substitutions remains an interactive task, which sometimes might require some effort and insight into the dependences among the specifications. In order to support the user in finding appropriate implications for substitution, the proof-obligation generator proposes a list of possible candidates. This list is generated simply as the cross-product of all top-level assumptions and all sub-component commitments with all sub-component assumptions, which do not refer to the same sub-component as the respective commitment.

Since assumption substitution remains an interactive task, it should be applied to a compositional proof only if necessary, i.e. if otherwise the technique suffers from complexity of the involved specifications. If no circular dependences have to be broken up and if the complexity of the entailed specifications can be kept reasonably small, verification according to the compositional proof rule can be applied fully automatically without assumption substitution.

7.4 Extending Verification to Complete Systems

The Role of SSL

Recall from chapter 5 that each of the activities of a STATEMATE design is associated with an entity declaration in the structural representation according to the compositional semantics. The different views to an activity are uniformly determined by SSL configurations, which associate simple or structural architectures with the corresponding entity. Simple architectures are used as containers, which can either bind a behavior representation to the entity or contain a set of STDx specifications for the referenced entity (we refer to definitions 5.11 - 5.14 (page 77 et seqq.)).

According to this representation, a proof-obligation of an activity can be formulated as a relationship among configurations of its representing entity. This way, a component proof is determined by the configuration, which associates a SMI representation to an entity and a STDx specification

[5] Recall, that multiple STDx-specifications can refer to the same sub-component. It is often the case, that a set of these specifications is involved in a compositional proof (cf. application examples in section 8.3).

of the set associated to the entity by a specification configuration, where both configurations have a *depth* of 0 (cf. definition 5.14 on page 79). Accordingly, the tautology proof of a compositional conclusion is determined by a selection from the STDx specification sets of a configuration C'' with $Ckind(C'') = specification$ and $depth(C'') = 1$ and a selection from the STDx specification set of configuration C''' with $Ckind(C''') = specification$ and $depth(C''') = 0$. The compositional conclusion hence relates C'', C''' and a structural configuration C' with $Ckind(C') = behavior$ and $depth(C') > 0$.

Because it is necessary that formal specification and behavioral representation of an activity refer to the same entity, the proof-manager associates all proofs with the entities of the design representation[6].

For illustration of these relations consider figure 7.3: Assume, for the case study of the radio-based signaling system that E is the entity representing `SYSTEM`, E_{11} is the interface representation of `TRAIN`, E_{12} stands for the interface of `COMMUNICATION` and finally E_{13} represents `CROSSING`'s interface. Then the compositional representation guarantees that configuration C - associating behavior description B with E - is equivalent to the decomposed view that is represented by configuration C' (Dashed lines in figure 7.3 refer to the the next-level configurations associated with the component instances by structural behavior configuration C', while dotted lines denote the next-level configurations associated with the component instances associated by structural specification configuration C'').

Uniform identification of STDx specifications plays a central role in the proof-management. The proof-management is geared to the specifications involved in the individual proofs, because it is sufficient that there exists a proof which establishes fulfillment of a STDx specification, regardless whether this proof is a component proof or if the specification is derived from other specifications.

[6] This is also reflected by the graphical user interface(GUI). The window for proof-obligation creation consists of three columns. In the firsts columns, all entities of the SSL data-base are listed. By selecting one entity, a new proof for this entity can be initiated. According to the user selection, all configurations of the selected entity are displayed in the second columns. These configurations provide behavioral representations (including decomposed representations) or formal specifications (also including hierarchical configurations). In a third column, all configuration of the selected entity are listed, which provide possible goals for proof-obligations. According to the selection of the user, proof-obligations and proof-scripts for component proofs, tautology checks (hierarchical w.r.t. design structure or simple implications of STDx specifications) or compositional conclusion rules are created.

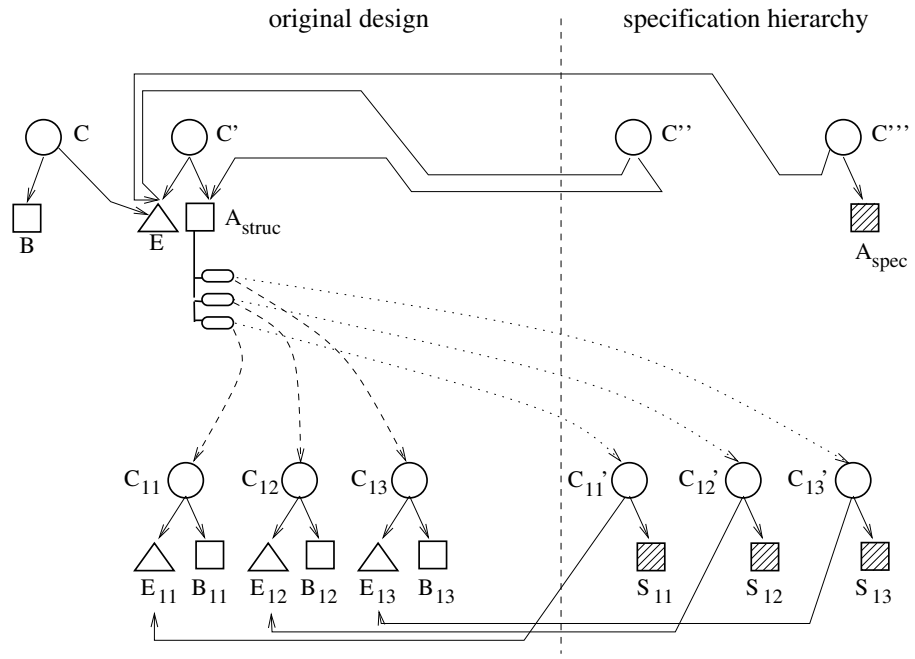


Figure 7.3: A Schematic SSL Hierarchy for Behavior and Specification

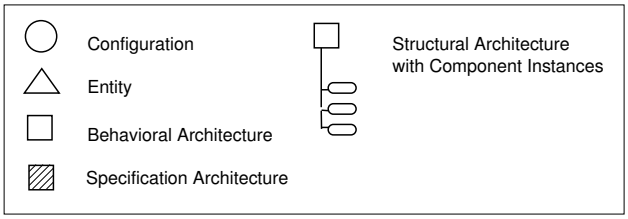


Figure 7.4: Legend

Definition 7.4 (STDx Specification Selection)
A STDx specification *spec* bound by an architecture A to some entity E of a SSL design representation is uniquely determined by the tuple $specsel{:=}(C, spec)$, where C is a configuration for which:

- $depth(C) = 0$
- $Ckind(C) = specification$
- $ent(C) = E$
- $arch(C) = A$ and $ent(arch(C)) = E$
- $spec \in Contents(arch(C))$

For $specsel{:=}(C, spec)$, let $ad_{sel}(specsel){:=}ad(spec)$ refer to all assumption diagrams instantiated by STDx specification *spec* of selection *specsel* and likewise let $cd_{sel}(specsel){:=}cd(spec)$ refer to the commitment diagrams of *spec*.

■

In the example of figure 7.3, a component proof-obligation for the entire `SYSTEM` can hence be defined by relating C to a STDx specification of C'''. Accordingly, a tautology proof-obligation as required for a compositional conclusion can be defined by relating a selection of the STDx specifications sets of the architecture to which C'_{11}, C'_{12}, and C'_{13} refer (C'_{11}, C'_{12} and C'_{13} are uniformly identified by C'') and a selection from the STDx bound to E by C'''.

In definition 5.15 (page 82), SSL scopes have been defined. Because C'_{11}, C'_{12}, and C'_{13} refer to different architectures of different entities, the STDx specifications in $Contents(arch(C'_{1i})$ for $i = 1, 2, 3$ also refer to the interfaces and therfore to the different scopes of entities E_{1i}. In particular, the scopes of E_{1i} differ from the scope of structural architecture A_{struc}, which comprises of the interface of E as well as of the local signals of $A_{struc.}$. In definitions 5.21 and 5.17 (pages 84, 83 , respectively, it has been considered, how the signals in the scope of an entity that is associated with a component instance is lifted to the scope of the instantiating structural architecture. There, for a structural configuration C_{struc} the composition of mappings

$$mapa \circ mapc : SigCompConf(C_{struc}) \rightarrow Sig(arch(C_{struc})) \tag{7.2}$$

has been defined, which maps the signal-names from the scopes of the component instance configurations referred to by C_{struc} to the signal-names in the scope of the structural architecture to which C refers. Exactly this mapping has to be applied to the STDx specification selections for C'_{11}, C'_{12} and C'_{13} (with $C_{struc}{:=}C''$ in (7.2)) before checking an implication according to compositional proof rule of definition 7.2. Since the semantics of a SSL structure description is the synchronous parallel composition of referenced components, mapping (7.2) is a *structure invariant*. By applying to every expression in the diagrams of the respective STDx specifications this structure invariant becomes part of the composition proof rule. According to the structure invariant the local interfaces to which the sub-component specifications of C'_{1i} refer are lifted to the scope of C''. Recall from section 5.4, that *mapa* as well as *mapc* are type-preserving mappings which comply with the promises of fact 5.1.

Definition 7.5 (Relating the Compositional Proof Rule to SSL)
Let $lift(C_{struc}, (C_x, td))$ denote the lifting of diagram td from the scope of C_x to the scope of C_{struc}, where $\exists ca \in Assocs(C_{struc}) : C_x = Conf(ca)$ by applying the above composed mapping (7.2) to every expression in $td \in Contents(arch(C))$.

Given configurations

- C, with $(depth(C) = 0) \wedge (Ckind(C) = behavior)$,
- C', with $(depth(C') = 1) \wedge (Ckind(C) = behavior)$, with $ent(C') = ent(C)$
- C'' with $(depth(C'') = 1) \wedge (Ckind(C'') = specification)$, with $ent(C'') = ent(C)$ and $arch(C'') = arch(C')$
- C''' with $(depth(C''') = 0) \wedge (Ckind(C''') = specification)$, with $ent(C''') = ent(C)$, as depicted in figure 7.3.

Let $(C''', spec)$ denote the top-level selection of a STDx specification to be verified by compositional verification.
W.l.o.g., let $selspec_1{:=}(C'_{11}, spec_1)$ and $selspec_2{:=}(C'_{12}, spec)$ be specification selections for subcomponent configurations, s.t. $\exists ca' \in Assocs(C'') : C'_{1i} = Conf(ca')$ for $i = 1, 2$ and $spec_i \in Contents(arch(C'_{1i}))$.

Furthermore, let C_{1i} , $i = 1, 2$ denote configurations associated with component instances of $arch(C')$, i.e. $C_{1i}{:}\exists ca \in Assocs(C') : C_{1i} \in Conf(ca)$. With $\mathcal{C}_i{:=}[\![Contents(arch(C_{1i}))]\!]$ (according to definition 5.11) , we rephrase the compositional proof rule:

$$\frac{\begin{array}{l}\mathcal{C}_1 \models_\Omega \left(\|_{a_1 \in ad_{sel}((C'_{11}, spec_1))} a_1, \|_{c_1 \in cd_{sel}((C'_{11}, spec_1))} c_1\right) \\ \mathcal{C}_2 \models_\Omega \left(\|_{a_2 \in ad_{sel}((C'_{12}, spec_2))} a_2, \|_{c_2 \in cd_{sel}((C'_{12}, spec_2))} c_2\right) \\ \models \left(\left(\left(\|_{a_1 \in, ad_{sel}((C'_{11}, spec_1))} lift(C'', a_1)\right) \Rightarrow_\Omega \left(\|_{c_1 \in cd_{sel}((C'_{11}, spec_1))} lift(C'', c_1)\right)\right)\right. \\ \quad \wedge \left(\left(\|_{a_2 \in ad_{sel}((C'_{12}, spec_2))} lift(C'', a_2)\right) \Rightarrow_\Omega \left(\|_{c_2 \in cd_{sel}((C'_{12}, spec_2))} lift(C'', c_2)\right)\right) \\ \qquad \left.\wedge \left(\|_{a \in ad_{sel}((C''', spec))} a\right)\right) \Rightarrow_\Omega \left(\|_{c \in cd_{sel}((C''', spec))} c\right)\end{array}}{\mathcal{C} \models_\Omega \left(\|_{a \in ad_{sel}((C''', spec))} a, \|_{c \in cd_{sel}((C''', spec)} c\right)}$$

By the promises of fact 5.3, $(\mathcal{C}_1||\mathcal{C}_2) = [\![Contents(arch(C_{11}))]\!]||[\![Contents(arch(C_{12}))]\!]$ is equivalent to $\mathcal{C}$, where $\mathcal{C}{:=}[\![Contents(arch(C))]\!]$

■

For checking whether the claimed implication is a tautology, technically all referenced observables in the observers for the lifted diagrams are represented by inputs. Since the observables to which the diagrams refer are represented by inputs in their individual observer modules, the parallel composition of all observers involved in the implication to be proven a tautology refers to all observables in the lifted scope only by inputs. This follows immediately from the definition of $||_\Omega$.

In this regard a problem w.r.t. violations for tautology checking should be mentioned. If the implication of the compositional proof-rule has can not be proven to be a tautology, the model checker generates a witness for the violation of the claim. In general, this witness does not provide insight into the reason for the violation, because the witness only shows, that due to un-satisfied local assumption(s) the entire implication is no tautology. It hence requires some intuition, to find out the reasons for the violation.

Proof-Management

Compositional conclusion rules have to take into account the equivalence of structural behavior configurations and simple behavior configurations. According to fact 5.3 (on page 80), there exists an equivalent behavior configuration for each structural behavior configuration. When determining whether all prerequisites of a compositional conclusion are fulfilled, every existing proof-obligation

for the involved sub-component specifications has to be regarded. Proofs for a (sub-)activity can be component proofs, derivations from another specification of the same activity or again a compositional proof. It has to suffice for proof-management that there exists any proof that establishes fulfillment of a particular specification, which is a prerequisite of a compositional proof for the instantiating activity.

For reasons of simplicity, we have not pictured further decompositions in figure 7.3, but a compositional conclusion rule proof has to take also proofs into account which are defined as relations between, for example, C_{13} and C'_{13} as well as for every relation between a configuration equivalent to C_{13} and C'_{13}.

Proof-obligations are the building blocks of proof-management. They either define an executable proof task for a proof of fulfillment of a particular STDx specification or conclusion rules for compositional reasoning using the results of other proof tasks.

Definition 7.6 (Proof-Obligations for SSL Description)
A proof-obligation is determined by a tuple $pobl{:=}(script, result, C_1, specsels, specsel, rpobl)$, where

- $script$ is a unique name of an executable proof-script associated with $pobl$. If no script shall be referred to, then $script{:=}\varepsilon$
- $result \in \{true, false, not\_proved, initiated\}$ is the result of the last execution of proof-script $script$.
- $specsel = (C_2, spec)$, where C_2 is a configuration with $Ckind(C_2) = specification$ and $depth(C_2) = 0$ and $spec$ is a STDx specification, s.t. $spec \in Contents(arch(C_2))$
- C_1 is a configuration, s.t. $ent(C_1) = ent(C_2)$
- $rpobl$ is a reference to another proof-obligation. If no proof-obligation is referred to, then $rpobl{:=}\varepsilon$
- $specsels$ is a (possibly empty) set of specification selections, s.t.:

$$
\begin{aligned}
(Ckind(C_1) = behaviour) \quad &\Leftrightarrow \quad (specsels = \emptyset) && (7.3)\\
(Ckind(C_1) = specification) & && (7.4)\\
\wedge(depth(C_1) = 0) \quad &\Rightarrow \quad \forall(C, spec) \in specsels : \\
& \quad (C = C_1) \wedge (spec \in Contents(arch(C_1))) \\
& \quad \wedge(rpobl = \varepsilon) \wedge (script \neq \varepsilon) \\
(Ckind(C_1) = specification) & && (7.5)\\
(depth(C_1) = 1) \quad &\Rightarrow \quad \forall(C, spec) \in specsels : \exists ca \in Assocs(C_1) : \\
& \quad (Conf(ca) = C) \\
& \quad \wedge(spec \in Contents(arch(Conf(ca)))) \\
& \quad \wedge(rpobl = \varepsilon) \wedge (script \neq \varepsilon)
\end{aligned}
$$

Then:

1. (7.3) is a component proof-obligation iff $(depth(C_1) = 0) \wedge (rpobl = \varepsilon) \wedge (script \neq \varepsilon)$
2. (7.3) is a component-proof conclusion rule iff $(depth(C_1) = 0) \wedge (script = \varepsilon)$ and *rpobl* refers to a proof-obligation according to case 1.
3. (7.3) is a horizontal conclusion rule iff $(depth(C_1) = 0) \wedge (script = \varepsilon)$ and *rpobl* refers to a proof-obligation according to (7.4)
4. (7.3) is a compositional conclusion rule iff $(depth(C_1) > 0) \wedge (script = \varepsilon)$ and *rpobl* refers to a proof-obligation according to (7.5)
5. (7.4) is a derivational tautology proof-obligation
6. (7.5) is a compositional tautology proof-obligation
7. All other cases are illegal.

■

Based on the definition 7.6 a proof-graph can be constructed which allows to manage dependences between individual sub-proofs and supports propagation of proof results according to modifications of the design or specifications.

Definition 7.7 (Proof-Graph)
A proof-graph is a directed acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{D})$, where $\mathcal{N}$ are the nodes of the graph, and $\mathcal{D} \subseteq \mathcal{N} \times \mathcal{N}$ are the edges of $\mathcal{G}$.
Each node $n \in \mathcal{N}$ is a tuple $n{:=}(ent, pobl)$, where

- *ent* denotes an entity
- *pobl* is a proof-obligation according to definition 7.6

There exists an edge $d \in \mathcal{D}$, $d{:=}(n_1, n_2)$ iff:

$$
\begin{aligned}
&(n_2.ent = n_1.ent) \wedge (n_2.pobl.rpobl \neq \varepsilon) \\
&\wedge \Big(\big((n_2.pobl.rpobl.specsels \neq \emptyset) \wedge \big(n_1.pobl.specsel = (C, spec) : \exists i, j \in \mathbb{N}_0 : \\
&\quad (C_i, spec_j) \in n_2.pobl.rpobl.specsels : (C, spec) = (C_i, spec_j) \big) \big) \Big)
\end{aligned}
$$

Let $in : \mathcal{N} \rightarrow \mathcal{D}$ denote a mapping that obtains all incoming edges $d \in \mathcal{D}$ for node $n \in \mathcal{N}{:}in(n){:=}\{d \in \mathcal{D} | \exists n' \in \mathcal{N} : d = (n', n)\}$. Furthermore, let $src : \mathcal{D} \rightarrow \mathcal{N}$ obtain the source node of an edge $d \in \mathcal{D}$. Hence, for $d = (n_1, n_2)$ let $src(d){:=}n_1$.

■

Though component proof conclusion rules are nearly trivial, they are added for homogeneity reasons. We illustrate definition 7.7 with figure 7.5 for a fictitious compositional proof (cf. figure 7.3). Suppose that there exists a structural architecture A_{struc} of entity E and a configuration C', s.t. $ent(C') = E$, $arch(C') = A_{struc}$, $Ckind(C') = behavior$ and $depth(C') = 1$. Accordingly,

let C'' be a configuration with $ent(C'') = E$, $arch(C'') = A_{struc}$, $Ckind(C'') = specification$ and $depth(C'') = 1$. Furthermore, we assume that $\exists ca \in Assocs(C')$, s.t. $C_{11} = Conf(ca)$ and likewise for C_{12} and C_{13}. Accordingly, we assume that C'_{11}, C'_{12} and C'_{13} are specification configurations which are bound to the component instances of A_{struc} by configuration C''. Finally, let C''' be a configuration, with $depth(C''') = 0$ and $Ckind(C''') = specification$ that binds an architecture A_{spec} ($arch(C''') = A_{spec}$) to entity E.

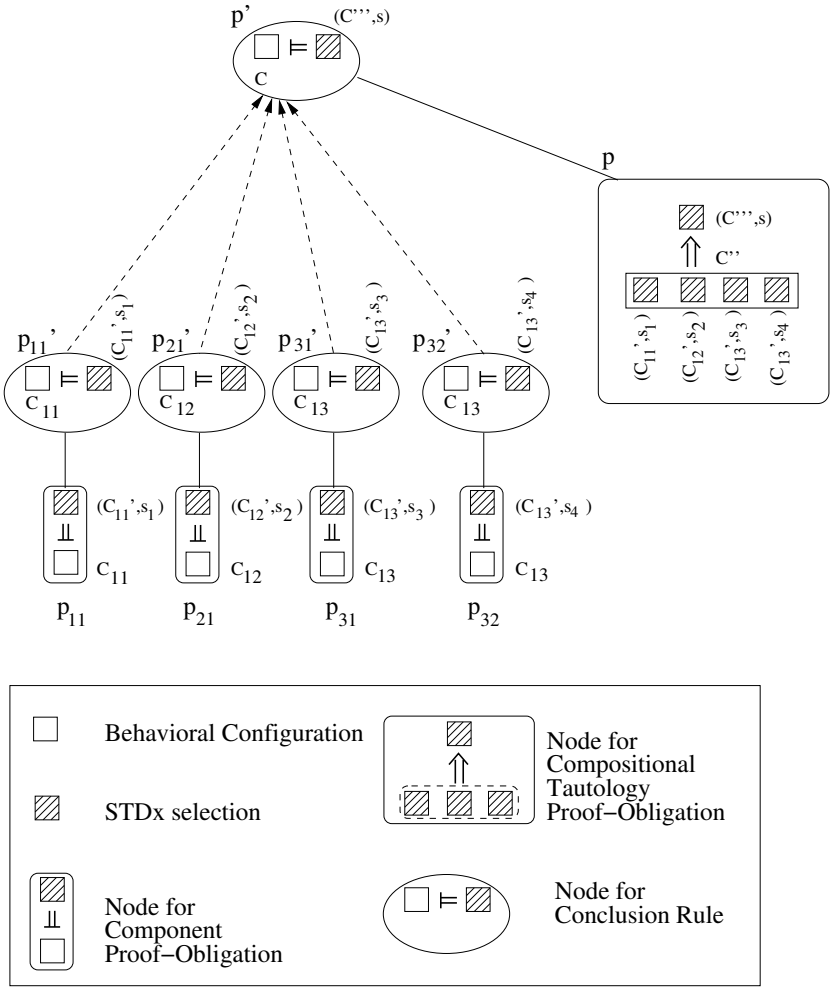


Figure 7.5: A Schematic Proof Graph

Dashed lines in the above figure denote edges of the proof-graph, while solid lines depict references of proof-obligations to other proof-obligations (called *rpobl* in definition 7.6).

Hence, p_{ij} with $i \in \{1,2,3\}$ and $j \in \{1,2\}$ in figure 7.5 are nodes referring to component proof-obligations with $\exists k : p_{ij}.pobl = (script, result, C_{jk}, specsels, specsel, rpobl)$, s.t. $script \neq \varepsilon$,$specsels = \emptyset$ and $rpobl = \varepsilon$. p'_{ij} with $i \in \{1,2,3\}$ and $j \in \{1,2\}$ in figure 7.5 are nodes referring to component-proof conclusion rules, s.t. $\exists k : p'_{ij}.pobl = (script, result, C_{jk}, specsels, specsel, rpobl)$, s.t. $script = \varepsilon$, $specsels = \varepsilon$, $p'_{ij}.pobl.specsel = p_{ij}.pobl.specsel$, and $p'_{ij}.pobl.rpobl = p_{ij}.pobl$. p if figure 7.5 is a node for a compositional tautology proof-obligation, which is referred to by node p'

that represents a compositional conclusion rule: $p'.pobl.rpobl = p.pobl$.

Definition 7.8 (Result Propagation)
Given a proof-graph $\mathcal{G} = (\mathcal{N}, \mathcal{D})$. Then for $n \in \mathcal{N}$ holds:

- if $(n.pobl.script \neq \varepsilon) \wedge (n.pobl.rpobl = \varepsilon)$, then $n.pobl.result$ is the result of the last execution of $n.pobl.script$
- if $(n.pobl.script = \varepsilon) \wedge (n.pobl.rpobl \neq \varepsilon)$, then

$$n.pobl.result := \begin{cases} true & \text{iff } (n.pobl.rpobl.result = true) \wedge \\ & \Big(\forall s \in n.pobl.rpobl.specsels \exists d \in in(n) : \\ & n' = src(d) \wedge (n'.pobl.specsel = s) \wedge \\ & (n'.pobl.result = true)\Big) \\ not\_proved & \text{otherwise} \end{cases}$$

■

Results are propagated along the edges of a proof-graph if a proof-script has been executed and obtained a proof result for the proof-obligation to which the script belongs. Propagation stops in a node , if its stored result is not affected by result propagation. Changes in the design or the specification hierarchy require more effort in order to keep the managed proof results consistent with the actual proof state.

Proof-management is based upon a SSL data-base. Changes in the behavior of a model, in its structure or regarding its STDx specifications are sensed by the proof-manager only if the modifications are checked into the SSL data-base, whereat every part of the newly checked-in information is assigned a modification time-stamp. Since all verification tools integrated with the STVE operate file-system based, proof-management provides a directory structure called verification data-base (VDB). This way, the VDB mirrors the contents of the SSL data-base and serves as working space for the verification tool set, but also proof-obligations, proof results and the proof-graph are stored in this file system. On every check-in into the SSL data-base proof-management compares the contents of the SSL data-base with copies of the contents in the VDB. Only if this comparison reveals a difference, the contents are also updated in the file system.

The results stored in the proof-graph have to be recomputed for every change in the VDB. Therefore, first all nodes n with $(n.pobl.rpobl = \varepsilon) \wedge (n.pobl.script \neq \varepsilon)$ are considered and it is determined whether the changes in the data-base affect the stored result. If for some node n any of the specifications or the model representation referred to by $n.pobl.script$ have been modified since the last execution of the script, then $n.pobl.result$ has to be set to not_proved. To enable this invalidation, for every script a list of references to parts of the design and specification hierarchy is maintained by proof-management. Invalidation is only performed if there are real changes affecting some of the referred parts, i.e. if some referred contents of the VDB have been updated. If the referred data is only attached a new time-stamp then the result remains valid. In a second step, all nodes n with $(n.pobl.rpobl \neq \varepsilon) \wedge (n.pobl.script = \varepsilon)$ have to be considered and it has to be determined whether $n.pobl.rpobl.result$ has been changed by the first step. Finally, this changes are propagated along the edges of the proof-graph.

7.5 Compositional Techniques - Related Work

In this section, we overview several compositional verification approaches based on model checking. We only describe the main characteristics of the different approaches. An excellent overview can be found in e.g. [KV98].

Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties describing the behavior of parts of the system. Since in general the complexity of a systems grows exponentially with the complexity of its parallel components, an obvious strategy is to check local properties using only the part of the system that is directly concerned with the implementation of these local properties. Hence, the central idea of compositional verification is: if it can be deduced that a combination of local properties implies an overall specification and if these local properties can be proven to be fulfilled by parts of the system, then it can be concluded that the system satisfies this specification as well.

There are a number of difficulties involved in developing a verifier that supports this style of reasoning. First, the verifier must be able to check whether every system containing a given component satisfies a given local property. It is often the case, that the local property is true only under certain conditions. Hence, the verifier must provide an environment for the component to be verified. This can be achieved by either deriving an *abstracted environment* from the *known concrete environmental component*s or by support of making *assumptions about the environment* of the component when doing verification. These assumptions, representing requirements on other components, must also be checked in order to complete verification. Second, the verifier must provide a method for concluding validity of a given specification from a particular combination of verified local properties. In principle, two possible solutions can be found in the literature. Either the conclusion is justified by the constructive rules of the compositional verification framework or establishing validity of the conclusion is itself a verification task, e.g. the implication between system specification and local properties needs to be proved using a tautology checker.

In general, a major distinction between different compositional verification approaches lies in the required knowledge about a component's environment when doing verification of a local property. Approaches originating in program verification, such as *compositional minimization*, rely on the knowledge of a concrete environment of the function to be verified. In contrast, approaches originating in hardware verification, such as *assume-guarantee reasoning* aim at verifying properties of a component regardless of a concrete environment. Here, a component is proven to guarantee a particular property φ in an arbitrary environment, provided this environment satisfies the assumptions made for verifying φ.

Compositional Minimization and Compositional Model Checking

Instead of verifying a system model by model checking the parallel composition of the constituent components, compositional model checking [CLM89] is based on the automatic reduction of components to the behavior that is observable at the interface to other components. Figure 7.6 illustrates this idea for a system comprised of two components P and Q. When checking a local property of component P, Q is considered being part of the environment of P. Using compositional minimization [CLM89, AdAG+01, AHM+98, Hol00], a *reduced* version Q' of Q is derived that characterizes just the behavior of Q that is visible to P via the communication between P an Q (cf. figure 7.6 (B)). The reduced component Q' is called an *interface process*. Hiding the communications of Q that are not visible to P and merging those states of Q that become indistinguishable may permit

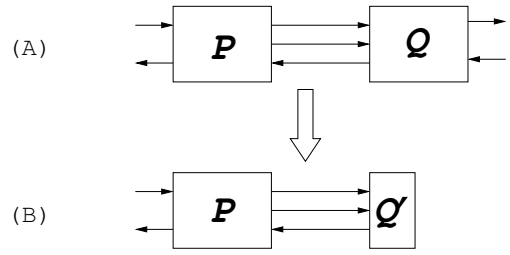


Figure 7.6: Compositional Minimization

the reduction of Q to a smaller component. The parallel composition of such simplified components will result, in general, in a much smaller system representation than the original one.

The problem in order to check correctness for the reduced system is ensuring that the simplified parallel composition satisfies the same logical properties as the original system. A rule of inference called *interface rule* provides the basis for compositional model checking. This rule deals with the parallel composition of two processes P and Q. Each of these processes is associated with a set of atomic propositions Σ_P and Σ_Q, respectively, used in distinguishing states and transitions. Interface processes P' and Q' of P and Q are then constructed according to the restriction of $P' \equiv P \downarrow \Sigma_Q$ and $Q' \equiv Q \downarrow \Sigma_P$, respectively.

$$\frac{\begin{array}{c} P \downarrow \Sigma_Q \equiv P' \\ \varphi \in \mathcal{L}(\Sigma_Q) \\ P'||Q \models \phi \end{array}}{P||Q \models \phi} \qquad \frac{\begin{array}{c} Q \downarrow \Sigma_P \equiv Q' \\ \psi \in \mathcal{L}(\Sigma_P) \\ P||Q' \models \psi \end{array}}{P||Q \models \psi} \quad ,$$

where $\mathcal{L}(\Sigma)$ is some logic ranging over the atomic propositions. In [CLM89] an algorithm is presented for construction of interface processes for parallel components for *asynchronous* as well as for *synchronous processes*[7] w.r.t. the universal fragment of the temporal logic CTL for asynchronous processes and CTL* for the synchronous case. Obviously, the soundness of the interface rule depends on the choice of the equivalence relation $\equiv$ as well as on the supported logic $\mathcal{L}(\Sigma)$. Thus, four properties are given in [CLM89] in order to establish soundness of the interface rule w.r.t. $\equiv$ and $\mathcal{L}(\Sigma)$.

1. Suppose $\Sigma_P = \Sigma_Q$, then $P \equiv Q$ implies $\forall \varphi \in \mathcal{L}(\Sigma_P)[P \models \phi \Leftrightarrow Q \models \phi]$.
2. If $P \equiv Q$ and R is another process, then $P||R \equiv Q||R$ and $R||P \equiv R||Q$
3. $(P||Q) \downarrow \Sigma_P \equiv P||(Q \downarrow \Sigma_P)$ and $(P||Q) \downarrow \Sigma_Q \equiv (P \downarrow \Sigma_Q)||Q$
4. If $\varphi \in \mathcal{L}(\Sigma)$ and $\Sigma \subseteq \Sigma_P$, then $P \models \varphi$ iff $P \downarrow \Sigma \models \varphi$

If $\equiv$and $\mathcal{L}(\Sigma)$ conform to these pre-order properties, then $P||Q \models \varphi$ follows from the interface rule. According to the choice of $\equiv$ and $\mathcal{L}(\Sigma)$ e.g. the following rule is sound:

[7] The authors use the term *asynchronous* to indicate, that in such processes there is no notion of "next system state", whereas synchronous processes are strictly synchronized with their environment s.t. "next system state" is a concept one can reason about.

$$\frac{\begin{array}{cc} P \downarrow \Sigma_Q \equiv P' & Q \downarrow \Sigma_P \equiv Q' \\ \varphi \in \mathcal{L}(\Sigma_Q) & \psi \in \mathcal{L}(\Sigma_Q) \\ P'||Q \models \varphi & P||Q' \models \psi \end{array}}{P||Q \models \varphi \wedge \psi}$$

Analogously to this rule also rules for other boolean combinations of φ and ψ can be obtained. A disadvantage of this approach to compositional verification is that only boolean combinations of φ and ψ can be verified, temporal logical relations between the components would require more complicated conclusion rules. In [CLM89] the authors surmise that it may be impossible to develop fully general system of inference rules that will handle arbitrary temporal properties. They argue that it may be necessary in order to use the interface rule to prove an implication of the form $(\varphi \wedge \psi) \rightarrow \delta$, where δ is another temporal logic formula that expresses a global property. In general, decomposition of a global property δ into local specifications φ and ψ can not be automated and thus remains an interactive task.

The interface processes P' and Q' are obtained from the representation of P and$Q$. Hence, verification of a local property φ of component P requires a concrete environment model of component P.

Assume-Guarantee Paradigm

In contrast to compositional minimization which requires a concrete representation of the environment of a component for verification of a local property, assume-guarantee reasoning permits the independent verification of local properties for components of a system. Using assumptions about the environment of a component P, it can be verified whether P guarantees a property φ in all environments conforming to the assumptions. For some concrete environment of P it remains to be proved that the assumptions made for verification of φ are not violated. The first approaches for reactive systems following this methodology were presented in [CM81] for invariant properties and in [Pnu85] for linear temporal logics properties.

A usual notation for assume-guarantee reasoning is a triple $\langle\varphi\rangle\, P\, \langle\psi\rangle$ where φ denotes the assumptions made for verification of the guarantee part ψ for component P. Validity of an assume-guarantee triple is defined inductively:

Definition 7.9 (Validity of an Assume-Guarantee Triple)
An assume-guarantee triple $\langle\varphi\rangle P \langle\psi\rangle$ is valid for all computations $\pi = s_0 s_1 ....$ of P, iff:

$$\forall i \leq k : s_i \text{ fullfills } \varphi \;\; \Rightarrow \;\; \forall j \leq k+1 : s_j \text{ fullfills } \psi$$

■

Informally, an assume-guarantee triple is valid if the satisfaction of ψ for a computations of length $k+1$ follows from the satisfaction of φ up to step k of the same computation. This definition can be exploited in compositional reasoning. Given assume-guarantee triples for n parallel components of a system, the satisfaction of the conjunction of their guaranteed properties follows for the parallel composition of all components from the satisfaction of the conjunction of their assumptions. This is formally expressed by the following rule:

Rule 7.1 (Compositional Assume-Guarantee Rule)

$$\frac{\langle\varphi_1\rangle\, P_1\, \langle\psi_1\rangle \wedge ... \wedge \langle\varphi_n\rangle\, P_n\, \langle\psi_n\rangle}{\langle\wedge_i\varphi_i\rangle\ P_1||...||P_n\ \langle\wedge_i\psi_i\rangle}$$

■

Note, that since the satisfaction of the guarantees is only implied by the satisfaction of the assumptions, the above rule is also valid if the conjunction of the component assumptions is not satisfiable. Hence, using the rule above, it remains to be shown that the system can perform any computation conforming to the assumptions.

Decomposition of global property specifications can, in general, not be automated, but remains an interactive task. Hence, expert knowledge is required in order to find a valid decomposition of a system specification. Pnueli argued in [Pnu85], that this interactive decomposition improves comprehension of the system under consideration and is thus a desirable activity. Nonetheless, even for large system manual decomposition is often difficult and error-prone. On the other hand, knowledge of how the system should behave plus feedback from an automatic verifier makes this feasible in practice.

Modular Model Checking

A special approach to compositional verification based on assume-guarantee reasoning is known as Modular Model Checking in the literature [GL94, KV98, KV00]. Once a valid decomposition of a system specification has been found, verification activities for local specifications for components of a system can be performed using a model checker. Variants of this approach have been presented for various different specification logics and techniques.

For linear temporal logic, an assume guarantee specification can be seen as a pair $\langle\varphi, \psi\rangle$, where both φ and ψ are linear temporal logic formulas. The meaning of such a pair is that all computations of the component are guaranteed to satisfy ψ, assuming that all the computations of the environment satisfy φ. This is formally[8] denoted by the assume-guarantee assertion $[\varphi]\, M\, [\psi]$. As observed in [Pnu85], in this case the assume-guarantee assertion $[\varphi]\, M\, [\psi]$ can be combined to a single linear temporal logic formula and checking whether M satisfies $\varphi \rightarrow \psi$. Thus, model checking a component with respect to assume-guarantee specifications is essentially the same as model checking the component with respect to linear temporal logic formulas.

The situation is different for branching time temporal logic. Here the guarantee is a branching time temporal logic formula, which describes the computation tree of the component. There are two approaches to the assumptions in assume-guarantee pairs.

One approach was considered by Grumberg and Long in [GL94], where branching time temporal assumptions are taken to the computation tree of the system within which the component is interacting. Grumberg and Long argued, that in the context of modular verification it is advantageous to use only universal branching time temporal logic, i.e. branching time temporal logic without existential path quantifiers. In a universal branching time temporal logic one can state properties of all computations of a model, but one cannot state that certain computations exist. Consequently , universal branching time temporal logic formulas have the helpful and desired property that once they are satisfied in a component, they are satisfied also in a system that contains this component.

[8] in the following we will use $[\phi]$ in order to emphasize that ϕ has a linear time interpretation, while $\langle\phi\rangle$ is used in order to emphasize that ϕ has a branching time interpretation.

More formally, a component M satisfies an assume-guarantee pair, formally denoted by assume-guarantee assertion $\langle\varphi\rangle\, M\, \langle\psi\rangle$ iff whenever M is part of a system satisfying φ, the system satisfies ψ too. The relation between systems containing M is defined using a simulation-pre-order '$\preceq$'. The simulation pre-order required by Grumberg and Long has to satisfy (w.r.t. synchronous parallel composition):

1. $\forall\phi \in \text{ACTL} : P \models \phi \Rightarrow P' \models \phi, \quad \text{iff} \quad P' \preceq P.$
2. $\forall P, P' \,:\, P||P' \preceq P$
3. $\forall P, P', P'' \,:\, P \preceq P' \Rightarrow P||P'' \preceq P'||P''$

Grumberg and Long have shown in [GL94] that one can associate with every ACTL formula φ a maximal model M_φ - also known as a tableau of φ - such that a model M' satisfies φ precisely when $M' \preceq M_\varphi$. Hence, branching time temporal assumptions can be dealt with by building the tableau $M\varphi$ for the assumption specification φ, and then checking whether the parallel composition of the component $M||M_\varphi$ guarantees ψ.

Given a pre-order according to the specification above, the rule for compositional verification can be formalized as follows:

$$\frac{\begin{array}{c} M \preceq M_\varphi \\ M'||M_\varphi \models \psi \end{array}}{M'||M \models \psi}$$

Another approach, presented by Josko in [Jos87, Jos93] prior to the approach of Grumberg and Long claims that the assumption in the assume-guarantee pair concerns the interaction of the component with its environment along each computation, and is therefore more naturally expressed in linear time temporal logic. A temporal logical framework - called MCTL - is presented, where an assume-guarantee pair consists of a *linear temporal assumption* φ and a *branching time temporal guarantee* ψ. The meaning of such a pair is that ψ holds in the computation tree that consists of all computations satisfying φ, formally denoted by $[\varphi]\, M\, \langle\psi\rangle$. Although different logics are used for assumptions and guarantees, the composition rule looks quite similar. Both approaches mainly differ in the tableau construction methods. Using linear temporal logic assumptions, the tableau construction required in Josko's approach has to follow a linear time interpretation. Kupferman and Vardi have shown in [KV98] that construction of the maximal model of an ACTL* formula φ involves double exponential blowup ($2^{2^{O(l)}}$, where l is the length of φ [KV98]), while the construction of the maximal model of ACTL as well as LTL formulas involves only exponential blowup ($2^{O(l)}$ [KV00]). There, Kupferman and Vardi also have shown that

- the model checking problem for assumption φ and guarantee ψ in ACTL, i.e. proving whether $\langle\varphi\rangle\, M\, \langle\psi\rangle$ can be done in time $km2^{O(l)}$ and in space $O\left(m(\log k + l + m)^2\right)$, where l is the length of φ, k is the size of M, and m is the length of ψ.
- the model checking problem for φ and ψ in ACTL*, i.e. can be done in time $k2^{O(m)+2^{O(l)}}$ and in space $O\left(m(m + \log k + 2^{O(l)})^2\right)$, where l is the length of φ, k is the size of M, and m is the length of ψ.
- the model checking problem for φ in ACTL* and ψ in ACTL can be done in time $km2^{2^{O(l)}}$ and in space $O\left(m(\log k + 2^{O(l)})^2\right)$, where l is the length of φ, k is the size of M, and m is the length of ψ.

Since LTL is embedded in ACTL*, these bounds hold also for φ in LTL and ψ in ACTL, as it is, in general, also the case for MCTL.

On the other hand, Kupferman and Vardi emphasize in [KVW00], that these complexities are the worst-case bounds. In practice, the constructions need not yield an exponential blowup. They surmise, that if the size of the assumption is not too large, the algorithms are impractical only for worst-case complexities.

However, Kupferman and Vardi point out a fundamental difference between the impact that the guarantee and the assumption have on the complexity of model checking. When verifying assume-guarantee assertions of the form

$$\langle \varphi_1 \wedge ... \wedge \varphi_l \rangle \, M \, \langle \psi_1 \wedge ... \wedge \varphi_m \rangle$$

it is easily possible to decompose the problem to verifying assertions of the form

$$\langle \varphi_1 \wedge ... \wedge \varphi_l \rangle \, M \, \langle \psi_i \rangle$$

in isolation, while it is not possible in general to decompose the assumption in a similar fashion.

Modular Model Checking using Synchronous Observers

Our approach to compositional verification is very similar to the one presented by Halbwachs, Lagnier and Raymond in [HLR93]. There, verification using synchronous observers has been presented first - to the best of our knowledge - for component verification as well as for compositional verification.

Synchronous observers are combined with a component M using a particular asymmetric parallel composition such that the observer only observes the component without ever disabling any possible computation of the component. An observed computation of the component under verification is accepted by the observer if its designated output never indicates a complaint.

Since a synchronous observer observes a computation step by step, the concept of time follows a linear time interpretation. Because the acceptance criterion of the observers as presented in [HLR93] is defined as a stepwise acceptance, this approach is restricted to safety specifications[9].

Like for the observation of a component, observers can be used also to judge about sequences of inputs provided by the environment of the component, and can thus be applied to specify assumptions.

Based on assume-guarantee reasoning, synchronous observers can be applied for compositional verification, which is considered in [HLR93] and e.g. in [Hol00] explicitly.

Obviously, the complexity of verification using synchronous observers is proportional to the product of the sizes of the parallel ingredients. In order to present a more concrete complexity bound, the complexity of a specific *observer construction procedure* has to be taken into account.

Summary

All approaches to modular model checking listed above including our own approach - regardless of the concrete specification formalism of the particular approach - follow the assume-guarantee paradigm for which a composition rule can be generalized by the following rule:

[9] By using a more advanced acceptance criterion - e.g. Büchi-acceptance - this approach could be extended to liveness specifications as well, at the price of a higher verification complexity.

Rule 7.2 (Generalized Composition Rule)

$$\left.\begin{array}{c} \{\varphi_1\}\, M_1\, \{\psi_1\} \\ \{\varphi_2\}\, M_2\, \{\psi_2\} \\ C(\varphi_1, \varphi_2, \psi_1.\psi_2, \phi, \psi) \end{array}\right\} \{\phi\}\, M_1||M_2\, \{\psi\}\,,$$

where $\{\phi\}\, M\, \{\psi\}$ *is an assume-guarantee assertion with respect to a particular interpretation suiting the specification formalisms to which* ϕ *and* ψ*, respectively, belongs, and* C *is a* composition condition *ruling the allowed compositional conclusion with respect to local assumptions and guarantees and global assumption and guarantee.*

■

In many approaches to modular model checking the composition condition C is a fixed conclusion rule - usually the conjunction of the local specifications. In principle there is no difference in our approach. In our approach composition condition C has to be established by tautology checking, i.e. C itself has to be verified by a separate verification task. Provided that it can be proved that a particular combination of local specifications and global specifications is a tautology w.r.t. the synchronous parallel composition of the involved components, complicated relations between local specifications and global specifications can automatically be concluded based on the conclusion rule.

Rule 7.3 (Composition Rule of this work)

$$\left.\begin{array}{c} \{\varphi_1\}\, M_1\, \{\psi_1\} \\ \{\varphi_2\}\, M_2\, \{\psi_2\} \\ \models (((\varphi_1 \rightarrow \psi_1) \wedge (\varphi_2 \rightarrow \psi_2) \wedge \phi) \rightarrow \psi) \end{array}\right\} \{\phi\}\, M_1||M_2\, \{\psi\}\,,$$

where $\{\phi\}\, M\, \{\psi\}$ *is an assume-guarantee assertion with respect to a particular interpretation suiting the specification formalisms to which* ϕ *and* ψ*, respectively, belongs. The composition condition ruling the allowed compositional conclusion with respect to local assumptions and guarantees and global assumption and guarantee must explicitly be checked using tautology checking.*

■

The approaches based on linear time temporal logic, branching time temporal logic, as well as the MCTL approach that combines both interpretations, are not equipped for reasoning about a quantitative perception of real-time that is independent from the temporal logical 'next step' operator. Through using synchronous observers, the specification formalism presented in this work is equipped to quantitatively refer to time by counting occurrences of particular observations.

The effort required for tableau construction of TPTL [AH89] - a discrete time quantitative logic extending LTL - is worse than the effort required for LTL since explicite specification clocks have to be represented in the tableau[10]. The same counts for TCTL [EMSS90]- which is a discrete time quantitative extension of CTL - compared to the complexity of CTL model checking.

Semantically, observers are represented in the same formalism that is used for the model representation. Since model checking is applied to a parallel composition of observers, model checking of observer acceptance benefits from reduction techniques - like Cone of Influence Reduction - that is applied by state-of-the-art model checkers such as, for example the VIS model checker. For applying

[10] Alur and Henzinger have shown in [AH89] that tableau construction for a TPTL formula with N logical and temporal operators and K as the product of constants in timing constraints can be done in time $2^{O(NK)}$, the model checking problem is EXSPACE-complete, as it is for ACTL*

invariance checking to the parallel composition of a set of observers, the worst case model checking complexity is that of computing a reachability analysis for this parallel composition.

8 Application of Verification Techniques - Experiences and Results

In this section we illustrate the presented verification techniques with some examples of their application to the Radio-based Signaling System. In section 8.1, application of robustness analyses and formal debugging techniques is documented. All verification tasks have been applied to the case study using the asynchronous as well as the synchronous execution semantics in order to allow a comparison of the complexity of the respective interpretation. Section 8.2 documents the application of observer-pattern based verification for five important safety requirements regarding the case-study.

Application of STDx verification is demonstrated in section 8.3. There, a proof of a real-time property of train requesting a status report from crossing is documented, as well as a real-time related compositional proof for the reaction of crossing. Finally, it is shown by a compositional proof that - in absence of hardware errors - a train passes crossing only after crossing has reported itself safe.

8.1 Application of Robustness Analyses and Formal Debugging

Analysis and debugging checks are applied to a model in an iterative process of finding and fixing bugs, adding features and reapplying the checks to the modified model. Application of analysis and debugging techniques support the developer in finding errors and generating simulations in order to explore the model and to discover flaws. In this section, we document the application of analysis and debugging techniques to the Radio-based Signaling System case study.

Since - to our knowledge - there exists no official version management for the case study, we have entered the iteration cycle at a fictitious point in time: somewhere in the development between the version of the model as documented in [KT00] and the version presented by [Klo03].

Each analysis has been applied to several temporary versions of the case study during the iteration of finding and fixing errors. In order to give an impression of the complexity of each of this iterations, we have collected the results of analysis application to the latest versions of the model which are free of robustness errors. All checks were performed on two dual processor SunOS 5.8 Blade 1000 work stations, with each 2GB memory and 900MHz SPARC processors. Since both machines could not be claimed exclusively for running the checks, the measured verification times are not absolutely exact and are generously rounded. Moreover, the times measured for invariance checking and bounded model checking are not exactly comparable. The times for model checker runs capture the pure model checking times, without previous model preparation times (in particular Finite State Machine generation), which is neglect-able compared to the time required by the model checker. In contrast, the times listed for bounded model checking also include model preparation steps. This difference originates in different integration and usage of the verification engines. Since the times measured for

bounded model checking and invariance checking for the different checks in general differ in orders of magnitude, we feel that these inaccuracies are tolerable.

8.1.1 A Synchronous Variant of the Radio-based Signaling System

Klose [Klo03] refers to a synchronous variant of the case-study using the words:

> The model presented here uses the asynchronous simulation semantics of STATEMATE, a variation using the synchronous semantics exists as well, but is not described in detail here, since the differences are only minute.

Approximately half of the proofs presented in [Klo03] are performed using this undocumented synchronous variant of the model. In principle, the synchronous variant equals the asynchronous model and deviates from it only w.r.t. interpretation of time. Executing a STATEMATE model - designed for asynchronous execution - using the synchronous execution semantics, allows the model to react to inputs at every step instead of stable *states*. Hence, *in absence of explicite references to time* - such as timeout expressions, synchronous execution *over-approximates* the asynchronous execution.

In presence of timeout expressions and scheduled actions it must be ensured, that time-triggered transitions and actions are taken and executed at the right moment in time. Thus, a *consistent* interpretation of time has to be ensured, such that specific combinations of states are reachable in the synchronous interpretation.

Here, the meaning of *consistency* is: all states and transitions reachable in the asynchronous model must be reachable also in the synchronous model. Notice, that internal computations may be performed more often compared to the asynchronous model. We are interested in a synchronous variant of the model only for obtaining complexity and run-time results in comparison with the results for the asynchronous model. Consequently, the execution w.r.t. synchronous execution semantics deviates in the interpretation of concrete values for speed, acceleration, and deceleration from the asynchronous model - the interpretation of positions is preserved**.**

Evidence for a consistent treatment of time can be obtained by checking on the one hand the reachability of particular phases of the modeled protocols. On the other hand, impossibility of violation of critical requirements regarding the protocol has to be verified. For example, crossing must be able to answer a status request sent by the train. Hence, train has to guess a correct crossing closing time in order to send its status request at the right moment. Furthermore, train has to wait an appropriate amount of time before assuming the crossing to be faulty.

Seven formal debugging checks have been identified to capture key properties of timing for the Radio-based Signaling System. It turned out that these seven checks succeed for both time models with nearly the same values of constants referred to in the timeout expressions of the model. Results for these checks are given at the end of this section.

One of these checks (Check 5 on page 241) revealed that constant `CCT` must have at least the value '12' for the synchronous execution semantics. Otherwise a status request from `TRAIN` is received by `CROSSING` at an instant of time, where crossing is still not able to report its status, because `CROSSING` can not enter the relevant state fast enough (state `BARRIER_CLOSED` - the static reaction of this state answers a status request from `TRAIN`).

Unfortunately, different values for input `D_SPEED` (desired speed - input from the train driver) have to be supplied to the model for the synchronous and the asynchronous execution semantics in order

to observe a "normal run". For the synchronous semantics, we ascertained that in absence of input ACTIVATE_CROSSING_CTRL:CP_REG_INP a normal run is impossible for all values of D_SPEED>19 (cf. Check 'normal run' on page 243).

This check revealed a *back-door effect*: It is possible to treat the crossing point TRAIN:CP as already regarded by setting input ACTIVATE_CROSSING_CTRL:CP_REG_INP. Considering a crossing point (TRAIN:CP) to be already regarded disables the braking-curve - implemented by COMPUTE_NOMINAL_SPEED - which means that the train overruns an unsecured crossing for arbitrary values of D_SPEED. Only if ACTIVATE_CROSSING_CTRL:CP_REG_INP is assumed to be always false, TRAIN adheres to the protocol of activating CROSSING and waiting for a safe-report before passing CROSSING. Such a subtlety is not necessarily a bug in the model, because nothing bad will happen if the input is driven correctly. On the other hand, if such a subtlety is not documented appropriately, the model fails to be a clean specification of the system under development.

Moreover, a specification problem must be emphasized: Since TRAIN and CROSSING do not share variables for the position of TRAIN and CROSSING, respectively, there is no global indication for TRAIN passing CROSSING. Since CROSSING detects a passed TRAIN only using a sensor, which is driven by a free input of SYSTEM, we can only rely on the perception of TRAIN (SYSTEM:PASSED_XING). The perception of TRAIN is not always trustworthy, SYSTEM:PASSED_XING only coincides with reality, if TRAIN treats CROSSING not as already regarded. Otherwise, TRAIN does not notice (and hence, does not indicate) over-running CROSSING.

A "*normal run*" is observed only if the TRAIN passes CROSSING w.r.t. its own perception (SYSTEM:PASSED_XING) without being emergency-stopped (TRAIN:STPPED). TRAIN continues its run along the track as long as no STPPED event is issued by its sub-activity SPEED_CONTROL_CTRL before CROSSING_SAFE_REC[1] is delivered by COMMUNICATION. SPEED_CONTROL_CTRL can issue a STPPED event only if the braking-curve is not disabled.

With appropriate assumptions about the environment (Assumption: initially ACTIVATE_CROSSING_CTRL:CP_REG_INP only after ACTIVATE_CROSSING_CTRL:IDLE immediate) it turned out that an normal run for the synchronous model was only feasible if D_SPEED<20, while the asynchronous model also shows a normal run for e.g. D_SPEED=100.

8.1.2 Stabilization

A core feature of a model regarding the asynchronous semantics is its stabilization behavior. There a model must always eventually become stable, i.e. be able to accept new inputs from the environment. Otherwise the model can engage in an infinite sequence of internal computations, which means divergence in terms of the asynchronous semantics.

Hence, before one can rely on results of other checks, divergence freedom of the model has to be assured. Moreover, an upper bound can be determined for the amount of steps the model can maximally perform until becoming stable again. Application of two very powerful abstraction techniques to asynchronous models - Relaxed Cone of Influence (RCOI [Bie03]) and Counter Abstraction (cf. section 7.2) - is only justified after having proven divergence freedom of the model. For the latter technique - abstraction of dynamic stabilization using a counter - an upper bound for the length of all super-steps must be determined. In order to obtain a first hint for this upper bound, we applied bounded model checking to the stabilization check. Since checking for an upper bound for the length of all super-steps does not permit application of the Relaxed Cone Of Influence (RCOI[2]),

[1] originated by CROSSING as CROSSING_SAFE_REC

[2] RCOI does not preserve the internal synchronization of an asynchronous model. Thus, RCOI may only be used

the check has to cope with nearly the full complexity of the non-optimized model. We successively applied the stabilization check with guessed upper-bounds :

- Bound 6: violation (13 Steps) in less than 1second
- Bound 7 : Violation of length 58 after about 1 hour
- Bound 8 : Violation of length 59 after about 2 and a half hour
- Bound 9 : No violation up to 68 Steps in about 13 hours without termination.

Thus, '9' seemed to be a good first guess for the stabilization bound check. When applying a stabilization check with upper bound '9' to the original asynchronous model using invariance checking instead of the bounded model checking engine, it turned out that the original model is too complex to obtain a result.

When searching for sources of complexity, we first considered the timeout expressions occurring in the model at transition triggers. A transition triggered by a timeout can only be taken if its source state is active for a series of super-steps. Thus, large constants in time triggered transitions have an important impact on model complexity, since the transition is enabled only after sequences of super-steps have passed, each consisting of series of steps. Hence, the model checker has to apply the transition relation several times in order to reach a state in which the transition is enabled.

Obviously, the transition BARRIER_CLOSED → TIME_OUT of statechart CROSSING_CTRL (cf. figure 3.10) can only be taken if BARRIER_CLOSED is active for MBCT (*Maximum Barrier Closed Time*=40) super-steps. Consequently, the model checker has to apply the transition relation up to 40*9 times in order to reach state TIME_OUT - if 9 is assumed to be the smallest upper bound for the length of super-steps.

Our first approach to reduce model complexity was to reduce this timer constant to a meaningful minimum, i.e. a smaller value which does not influence the reachability of all states and transitions and which preserves the overall functionality of the model disregarding this concrete deadline: Only when being in state BARRIER_CLOSED of CROSSING_CTRL , activity CROSSING can answer an incoming *status request* originated from activity TRAIN. TRAIN can emit this event only at transition WF_CROSSING_CLOSED →REQUEST_CROSSING_STATUS - triggered again by a timeout event depending on constant CCT (*Crossing Closing Time*=8). Thus, a meaningful minimal value for MBCT must preserve the ability of CROSSING_CTRL to react on *status request*. This is captured by property 'last(CROSSING_CTRL:BARRIER_CLOSED) and CROSSING_CTRL:BARRIER_CLOSED and SYSTEM:STATUS_RQ_REC' (Check 5 on page 241).

In an iterative process, we searched for a meaningful value for MBCT, by repeatedly applying all relevant "drive-to-..." checks and coverage computations as documented below.

Although the modification MBCT seems not to be of great impact in terms of state-bits, reducing MBCT to 8 instead of 40 drastically decreases verification complexity for some checks. By applying the listed "drive-to-..." checks and coverage computations to the original and the modified model, we assured ourselves that the derived asynchronous model behaves the same as the original one - except for the concrete deadline represented by MBCT. In the following we refer to this *time-abstract* variant of the model as *asynch1.*

for a model for which the stabilization bound check already succeeded. In order to establish an upper bound for the length of all super-steps only the normal, functional Cone Of Influence optimization can be applied which preserves the synchronization behavior of the model.

After reducing MBCT to 8, the asynchronous model still remained too complex for verification of stabilization with bound '9'.

As another source of complexity we identified the computation of speed and position in activity TRAIN. Since speed and position are calculated at time-triggered transitions only - i.e. only once per super-step - the model stabilizes regardless of the concrete computations. Hence, if an over-approximation - guessing arbitrary values for speed and position whenever the respective transitions are taken - always stabilizes within a fixed bound, the original model will do so within the same bound. Unfortunately, like RCOI also propositional abstraction [Bie03]- as offered by the STVE - does not preserve stabilization of the model. *Freeing* as well as *strong*-abstraction abstract from concrete values of data-items in a step-oriented way. Abstract data-items are allowed to change every step, even if they are computed explicitly only once a super-step.

The local variables ODATA, T_COMMANDS and NOMINAL_SPEED of TRAIN are assigned new values only at transitions which are triggered by timeout events. In order to abstract from the concrete computations, we replaced the computations by fresh *slow* inputs, without influencing stabilization of the model. In contrast to the propositional abstraction technique available in the STVE, this hand-abstraction concerns only the action part of timeout-triggered transitions - the assigned values are abstracted, transition triggers remain unchanged: new values are provided only in stable states. Only the results of concrete computations are over-approximated using appropriate inputs. Hence, replacing concrete computations with inputs only in the action parts of transitions strictly over-approximates the model behavior. This would have an impact on the synchronization behavior of the model only if the synchronization were dependent on concrete results of these computations.

We refer to the resulting abstract asynchronous variant of model *asynch1* with hand-abstraction of ODATA, T_COMMANDS and NOMINAL_SPEED as *asynch3*. Figures 8.1 and 8.2 show the abstracted versions of statecharts ODOMETER_CTRL and SPEED_CONTROL_CTRL used for the stabilization bound check.

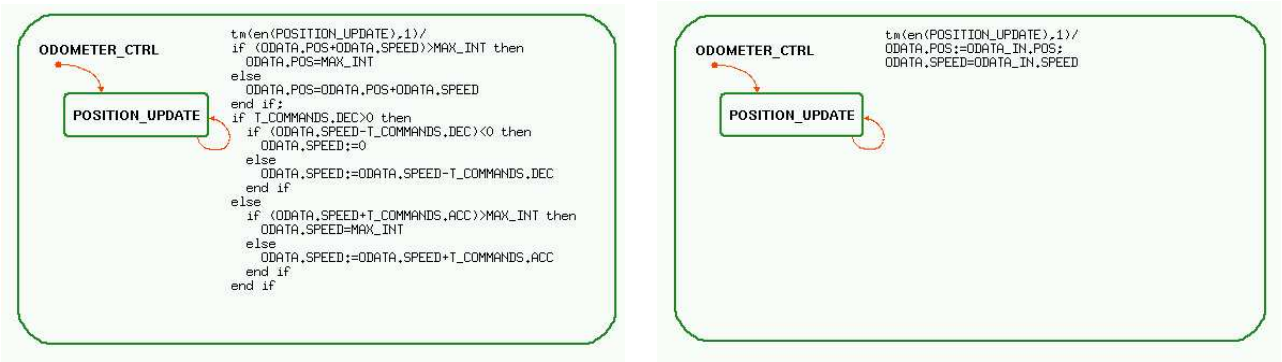


Figure 8.1: Statechart ODOMETER_CTRL - left : original , right: ODATA driven by input

Using this over-approximation, we have been able to verify that all super-steps are upper-bounded by 9 - each super-step of the model stabilizes within 9 steps. The complexity of verification is relatively independent from the choice of the upper bound: choosing 15 or 20 instead of 9 does not influence the verification time significantly.

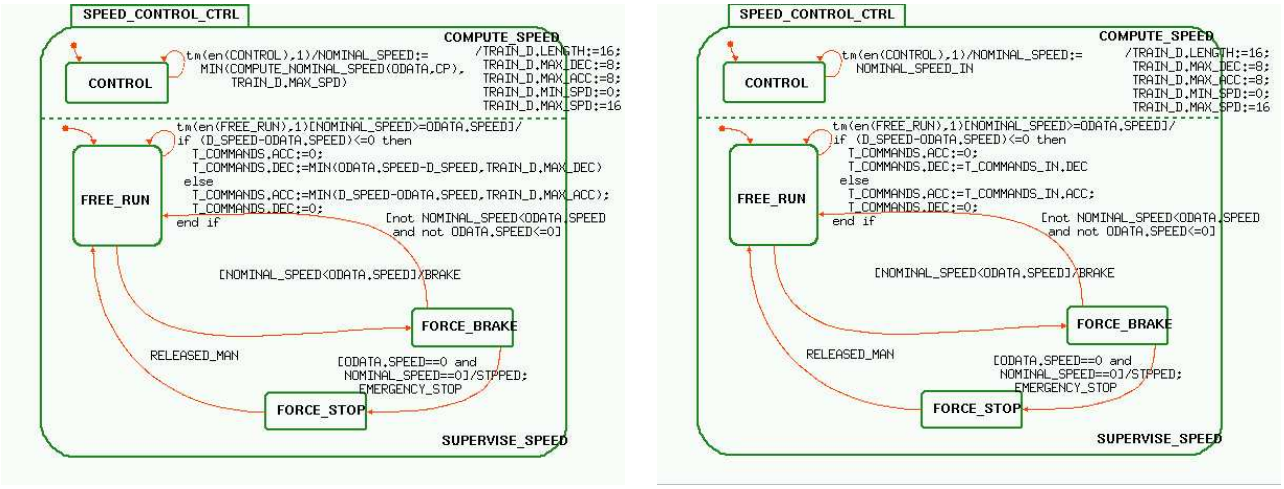


Figure 8.2: Statechart `SPEED_CONTROL_CTRL` - left : original, right : `NOMINAL_SPEED` and `T_COMMANDS` driven by inputs

bound	result	Time(mc)	asynch3 - 46i/273s
9	true	2h	(136 image comps)
10	true	2h	
15	true	2h	
20	true	2h	

Table 8.1: Bounded Stabilization Check Results

Once having proved stabilization, the usage of RCOI optimization and Counter Abstraction in the following analysis and debugging checks is justified. Until stabilization has been proved, all results of analysis and debugging checks obtained using RCOI optimization and Counter Abstraction are *valid only under the assumption* that there exists a bound, for which the model always eventually stabilizes.

8.1.3 Robustness Checks

Table 8.2 gives an overview of the variants of the model to which analyses and debugging checks have been applied. *Synch0* refers to the synchronous model, *asynch0* refers to the original asynchronous model, *asynch1* denotes the asynchronous variant with `MBCT` reduced to 8 instead of 40 and finally *asynch3* denotes the hand-abstracted model used for the stabilization check. For each variant of the model as well as for some of their sub-models, the number of input bits - denoted by i - and state-bits - denoted by s - is listed. Additional inputs for the resolution of non-deterministic choices are listed in parenthesis after the number of inputs. For example, the *asynch0* variant of the entire system model has 64 inputs representing the inputs of the STATEMATE model plus 33 inputs for resolving non-determinism. 508 state-bits are required to represent the states and variables.

The numbers of input- and state-bits is not the only relevant measure for complexity. Although *asynch3* is larger than *asynch0* in terms of inputs and state-bits, checking stabilization for *asynch3* is

feasible, while this is impossible for *asynch0*. Also, the original model *asynch0* is only two state-bits larger than *asynch1;* the impact of MBCT on the model complexity is disproportional. Nonetheless, model sizes in terms of inputs and state-bits give a rough impression of how complex the system and its components are.

It must be emphasized, that for many checks Cone of Influence Reduction and other model optimizations drastically reduce the size of the model. Therefore, for each documented verification task the effective size of the model - handled by the model checker after all optimizations - is listed. Similar optimization are applied to the model also for application bounded model checking. Hence, the reported model sizes roughly correspond also to the size of models to which bounded model checking has been applied.

Models:	synch0 *	asynch0	asynch1 **	asynch3 ***
SYSTEM	64(+33)i/425s	64(+33)i/508s	64(+33i)/506s	113(+33)i/557s
TRAIN	42(+13)i/215s	42(+13)i/261s	42(+13)i/261s	-
ACTIVATE_CROSSING_CTRL	91(+5)i/59s	91(+5)i/83s	91(+5)i/83s	-
SPEED_CONTROL_CTRL	43(+4)i/103s	43(+4)i/120s	43(+4)i/120s	-
COMMUNICATION	7(+2)i/28s	7(+2)i/32s	7(+2)i/32s	-
CROSSING	28(+18)i/188s	28(+18)i/227s	28(+18)i/225s	-

Table 8.2: Model Sizes

* CCT=12 instead of 8 as in asynch0
** MBCT=8 instead of 40 as in asynch0
*** MBCT=8 and manual abstraction of ODATA, T_COMMANDS and NOMINAL_SPEED

Range-Violations: (RV) Range violations are severe bugs in a model. In the original model - as presented in [KT00] - three range violations were detected. Although, all range violations have already been fixed in the model version presented in [Klo03], we list these bugs here nonetheless, since they document how simple the sources of severe bugs in a model can be:

- In the model version documented in [KT00] a data-item T has been increased at a self-loop in statechart TIMER_CTRL without checking for an upper bound. Obviously, this bug resulted from an unfinished attempt to model additional functionality. Since there existed no reference to T anywhere in the model, deleting the entire self-loop fixed the problem.
- According to [KT00] the static reaction of state FREE_RUN in statechart SPEED_CONTROL computes:
  ```
  /if (D_SPEED-ODATA.SPEED)<0 then
  T_COMMANDS.ACC:=0
  T_COMMANDS.DEC:=MIN(D_SPEED-ODATA.SPEED),TRAIN_D.MAX_DEC)
  else ...
  ```
  Hence T_COMMANDS.DEC is assigned a negative value, but is declared with a natural domain. This bug was simply an inadvertent permutation of function arguments, which could be easily fixed. Recall, that range violations only emerges dynamically: if the observed speed ODATA.SPEED is greater than the intended speed D_SPEED demanded by the driver. Hence, the problem does not arise always during simulation but only for

particular values of D_SPEED .

- The third range violation could not be detected by analysis - since analysis only considers user defined data-items - but was revealed during error-path concretization for a *drive to state* check:
  In the implementation of procedure COMPUTE_NOMINAL_SPEED (in the model version described in [KT00], but not documented there) a local variable of natural domain is assigned a negative value. Since this local variable is only an auxiliary variable, which is not known to the user, the analysis failed to detect this error.

For the fixed version of the model, range violation analysis detected six more potential situations of range violations. The un-reachability of these potential violations has been determined already in the generation of a BDD representation of the model.

Model	#possible	#reachable	Time inv	Model Size
asynch0	6	0	1sec	0i/0s
asynch1	6	0	1sec	0i/0s
synch0	6	0	1sec	0i/0s

Table 8.3: Range Violation Checks

Non-Determinism: (ND) For the original model, non-determinism analysis detected two potential non-determinism conflicts, one in statechart ACTIVATE_CROSSING_CTRL and the other one in statechart BARRIER_CONTROL_CTRL.

- Triggers of transitions PASS_CROSSING→IDLE and PASS_CROSSING → FAULTY_CROSSING in statechart ACTIVATE_CROSSING_CTRL (cf. figure 3.6) have been non-exclusive. Invariance checking has been applied in order to check dynamic reachability of this potential non-determinism. The model checker proved the situation to be unreachable.
  In order to provide an unambiguous description of the system, the STATEMATE model should reflect exclusiveness of transitions explicitly. Consequently, the STATEMATE model has been modified such that mutual exclusion of the respective transitions is obvious at first glance. This has been achieved by adding '(not TIMEOUT)' to the trigger of PASS_CROSSING→IDLE.
- The detection of a potential non-determinism in scope BARRIER_CONTROL_CTRL is an artefact due to the translation of STATEMATE models to SMI: Timeout events are modeled using counters in the SMI code. These counters are initialized to the respective timeout expression and are decremented every (Super-) step. An internal event is emitted when the count-down reaches zero. Since neither counters nor the internal events are shared between transitions triggered by timeout expressions, the analysis tool is not able to determine static exclusivity of such time-triggered transitions..
  It is sufficient to check reachability of this potential transition non-determinism first in the scope of the affected transitions, i.e. statechart BARRIER_CONTROL_CTRL. Only if the analysis detects a reachable non-determinism it can be useful to apply the analysis to a larger scope, for which the non-determinism may be unreachable, due to mutual exclusiveness of computations. Anyhow, for documentation purposes, we have applied

the checks also to the enclosing scope CROSSING and the entire system model. Table 8.4 shows the results for application of non-determinism analysis for the different model variants:

Model	#potential	#reachable	Time inv	Model Size
asynch0	1	0	1sec	7i/32s
synch0	1	0	1sec	7i/24s

Table 8.4: Non-Determinism Checks for BARRIER_CONTROL_CTRL

Model	#potential	#reachable	Time inv	Model Size
asynch0	1	0	5m	12i/102s
asynch1	1	0	35sec	12i/100s
synch0	1	0	10sec	12i/83s

Table 8.5: Non-Determinism Checks for CROSSING

Model	#potential	#reachable	Time inv	Model Size
asynch0	1	0	TO!12h	31i/274s
asynch1	1	0	TO!12h	31i/272s
synch0	1	0	TO!12h	31i/225s

Table 8.6: Non-Determinism Checks for entire SYSTEM

Write/Write-Hazards: (WW) Even though multiple write accesses to the same data-item within one step are resolved by the STATEMATE simulator, they should be treated as modeling flaws in a model based development process. The behavior of a reference model should be independent from internal resolution strategies of the simulator. In order to obtain an unambiguous reference model from the early development phases, write/write hazards have to be avoided. Write/Write-hazards analysis detects two potential hazards for record T_COMMANDS: for component ACC used for communication of acceleration commands in TRAIN among activities SPEED_CONTROL, BRAKE and ODOMETER and for component DEC capturing deceleration commands:

TRAIN : Write/Write_Hazards: Since all accesses to record T_COMMANDS are located in the scope of activity TRAIN it suffices to check activity TRAIN for multiple writer conflicts. Unfortunately, TRAIN is too complex for application of invariance checking, but TRAIN is the minimal scope for applying a multiple writer analysis, since T_COMMANDS is driven as well by SPEED_CONTROL_CTRL as by activity BRAKE. Unless a write/write hazard is expected to be reachable, application of bounded model checking instead of model checking is not recommended. Only if bounded model checking detects a multiple writer conflict within a user-defined bound, the application is successful. Otherwise, if no hazardous situation is reached bounded model checking provides no useful result. Hence, only a complete verification technique can be applied in order to determine whether a hazard is reachable or to prove that there is no multiple writer hazard in the scope of TRAIN.
The asynchronous model turned out to be too complex for application of invariance checking

in order to check the reachability of both potential hazards. Only CTL model checking using an over-approximation of reachability computation [MJH+98] obtained the desired results. The results are listed in columns 'Time mc' of the tables below.

Model	#potential	#reachable	Time inv	Time mc	Model Size
asynch0	2	0	TO!12h	40sec	24i/150s
synch0	2	0	2h	20sec	24i/124s

Table 8.7: Write/Write Hazard Checks for TRAIN

SYSTEM : Write/Write Hazards For the entire model, static analysis also detects two possible multiple writer conflicts, both affecting the record `T_COMMANDS`.

Model	#potential	#reachable	Time inv	Time mc	Model Size
asynch0	2	0	TO!12h	3m	31i/270s
asynch1	2	0	TO!12h	3m	31i/268s
synch0	2	0	TO!12h	1m	31i/225s

Table 8.8: Write/Write Hazard Checks (entire SYSTEM)

Sequential Write/Write Hazards: (SWW) It is not in general clear how to assess occurrence of sequential multiple writer conflicts in a model. In the case of the Radio-based Signaling System, static analysis detects that both components of record `T_COMMANDS` might be affected by sequential multiple writer conflicts. Both potential conflicts can be proven to occur in runs of the model. Since `T_COMMANDS` is used for internal communication between the sub-activities `BRAKE, ODOMETER` and `SPEED_CONTROL_CTRL` we treat the detected conflicts as intended behavior. Recall, that internal activities communicate in a step-by-step manner according to the asynchronous semantics of STATEMATE. `T_COMMANDS` is only used for internal computations of the model and not referred to in communications with the environment.

Model	#potential	#reachable	Time bmc	Time inv	Model Size
asynch0	2	2	1m	4m	31/274s
asynch1	2	2	1m	4m	31i/272s

Table 8.9: Sequential Write/Write Hazards (entire SYSTEM)

Read/Write Hazards: (RW) The assessment of Read/Write hazards depends on the intended usage of the affected objects. Most of the affected objects are events. Communication between parallel activities emitting and consuming events is an essential aspect of the modeling style of the case-study. Data-items affected by Read/Write hazards are only these data-items which are used for the computation of speed and positions. The `SYSTEM` model turned out to be too complex for application of invariance checking. Using the bounded model checking engine 31 of 32 potential Read/Write hazards turned out to be dynamically reachable within an unroll

bound of 100 for the transition relation.

Model	#potential	#reachable	Time bmc	Time inv	Model Size
asynch0	32	31 (k=100)	1h30m	TO!12h	31i/270s
asynch1	32	31 (k=100)	1h	TO!12h	31i/268s
synch0	32	31 (k=100)	3m30sec	TO!12h	31i/225s

Table 8.10: Read/Write Hazards Checks (entire SYSTEM)

Since the case-study was never intended to serve as reference model for an implementation but only as conceptual model illustrating a protocol, we decided to assess the detected Read/Write hazards as being harmless. The simulator guarantees that all read accesses to objects refer to their values from the beginning of the actual step while write accesses affect an object only after all read-accesses have been executed. In a real development process, Read/Write accesses would probably have to be treated with greater care.

Sequential Read/Write Hazards: (SRW) Also Sequential Read/Write hazard analysis revealed 22 reachable hazards out of 22 potential hazards. The same as for Read/Write hazard analysis, the SYSTEM model is too complex for application of invariance checking. Bounded model checking has been able to reach all hazards in an acceptable time (with maximal unroll depth of 100).

Model	#potential	#reachable	Time bmc	Time inv	Model Size
asynch0	22	22	45m	TO!12h	31i/317s
asynch1	22	22	30m	TO!12h	31i/315s

Table 8.11: Sequential Read/Write Hazards Checks (entire SYSTEM)

8.1.4 Summary of the Application of Analyses

Even though analyses require a complete verification technique, which can obtain witnesses for detection of conflicts as well as definitely determine un-reachability of the examined potential conflict, it is often advantageous to apply bounded model checking. Invariance checking using a BDD based model checker suffers more often from complexity of the model. Therefore, in particular when reachability of examined conflicts is expected, bounded model checking obtains witnesses in nearly all considered cases much faster than invariance checking. As a rule of thumb, bounded model checking should always be the first choice in the application of analyses. Only if bounded model checking does not detect a witness for the examined conflict in reasonable time, invariance checking or even

CTL model checking - with over-approximation of reachability computation - as in the case of the presented non-determinism analysis - need to be applied in order to obtain a definite result.

In those cases for which a complete verification technique is unavoidable - as it is the case for the stabilization check - only abstractions can help to tackle verification complexity. Except for the stabilization check, all presented checks could have been performed by the developers of the case-study using a nearly pure push-button technique, requiring only little expert knowledge.

We have presented a model-specific hand-abstraction for the considered case study, which led to success for the presented stabilization check. An issue for future work should be the automation and integration of the presented hand-abstraction with the STVE. Replacing local variables by slow inputs preserves stabilization properties, but can significantly reduce complexity.

Obviously, application of analyses to the synchronous variant of the model leads to significantly less verification complexity than application to the model using the asynchronous execution semantics. If it can be ensured, that no timing problems disable parts of the model, application of analyses for the synchronous execution semantics can obtain helpful hints also for the asynchronous case.

In general, application of analyses has been successful: for the considered case-study bounded stabilization, absence of non-determinism and range-violations as well as absence of write/write hazards have been formally proved for the asynchronous as well as for the synchronous variant of the model. Thus, it has been verified that the case study is free of critical design flaws.

8.1.5 Application of Formal Debugging

In this section, seven formal debugging checks are presented, which iteratedly have been applied to the case study. The obtained witnesses have been translated to simulation control programs and executed with the STATEMATE simulator. All checks have been performed on dual processor SunOS 5.8 Blade 1000 work stations, with each 2GB memory and 900MHz SPARC processors. If not explicitly stated otherwise, the maximal unroll-depth for bounded model checking has been chosen 100. Justified by the successful stabilization check, for the formal debugging checks also approximation of dynamic stabilization with a counter has been applied. For each check the rows *asynch0-cnt9* and *asynch1-cnt9* in the tables depict the results obtained from application of the check to the models *asynch0* and *asynch1* , respectively, with counter approximation using bound 9. For the resulting witnesses a SMI-based simulation of the model has been applied after verification in order to re-establish dynamic stabilization for the approximated witnesses. In the tables below, the results of these simulations are taken into account by entries of the form
<counter-trace-length>→<stabilizing trace length>

Recall from section 6.1 that the expression language supported for drive-to-property checks provides the special constructs '`last()`' and '`primed()`', of which we make use in the definition of checks 2,3,4, and 5. '`last(<variable>)`' introduces a new variable in the model always keeping the value of `<variable>` of the last step, while '`primed(<variable>)`' introduces a new output referring to the primed value of `<variable>`. This way, the value of variable in the previous step and their computed values for the next step can be referred to in simple invariant specifications for drive-to-property checks.

Check 1: Drive to Transition: '`ACTIVATE_CROSSING_CTRL:PASS_CROSSING` → `ACTIVATE_CROSSING_CTRL:IDLE`'
The check is aimed at obtaining a witness for some run according to the protocol for which

train passes the crossing.
In order to take the specified transition, the system has to engage in a computation, s.t.

- TRAIN approaches the CROSSING
- TRAIN initiates communication with CROSSING (via COMMUNICATION)
- TRAIN requests status information from CROSSING or stops due to un-established COMMUNICATION.
- If TRAIN sends a status request, either CROSSING responds to the status request or the TRAIN is stopped due to a timeout. If TRAIN is not stopped it passes a secured CROSSING. Otherwise CROSSING has to be released manually.

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	28	1m	8h	31i/270s
asynch1	28	1m	8h	31i/268s
asynch0-cnt9	65→32//65→33	30sec	15m	31i/273s
asynch1-cnt9	65→32 // 65→33	30sec	15m	31i/271s
synch0	16	5sec	3m	31i/225s
with freezing:				
asynch0	50	1m	10h	26i/259s
asynch1	50	1m	10h	26i/257s
asynch0-cnt9	-/105→50	k=100 too small(2m)	15m	26i/262s
asynch1-cnt9	-/105→50	k=100 too small(2m)	15m	26i/260s
synch0	30	10sec	3h30m	26i/220s

Table 8.12: **Check 1**: Drive to Transition: 'ACTIVATE_CROSSING_CTRL:PASS_CROSSING → ACTIVATE_CROSSING_CTRL:IDLE' without freezing and with error-indication inputs frozen to false

Note, that the witness need not necessarily show a 'normal run' of the system. Even if e.g. COMMUNICATION rejects establishing communication or if CROSSING fails to report a safe status due to light- or barrier errors, the transition is reachable.
Table 8.12 lists the results and run-times for check_1. The check can be combined with assumptions and freezing: the paths of the upper half 8.12 involve a 'faulty crossing' and 'manual release' (cf. figure 3.6), where the latter is represented simply by a free input of the model. By freezing this input to false, transition 'FAULTY_CROSSING →PASS_CROSSING' can be disabled in order to obtain more interesting witness. Also, disabling all error-sensors influences the resulting witness-traces. The lower part of table 8.12 shows the effects of freezing SYSTEM:SENSOR_ON, SYSTEM:SENSOR_ERR, SYSTEM:RED_ERR, SYSTEM:YELLOW_ERR and SYSTEM:RELEASED_MAN to false.

Check 2: Drive to Property: 'CROSSING_CTRL:OPENING_GATES and CROSSING:BARRIER_OPENING and primed(CROSSING_CTRL:IDLE)'
checks 2,3 and 4 check for reachability of different configurations of basic states of CROSSING (statechart CROSSING_CTRL, cf. figure 3.10) in the context of the entire system model. Check_2

checks the ability of CROSSING to become idle again and hence being ready for a new activation without getting stuck. Although check_2 is defined only by referring to states and one event in the scope of CROSSING, the resulting witness traces are of interest w.r.t. the entire system. In order to reach the specified configuration, TRAIN has to initiate COMMUNICATION, COMMUNICATION has to transmit all communications, and CROSSING has to be activated by TRAIN via COMMUNICATION. The resulting witness-traces according to table 8.13 show all prerequisites for finally taking transition CROSSING_CTRL:OPENING_GATES→CROSSING_CTRL:IDLE.
Check 2 has been applied in combination with freezing the input SYSTEM:D_SPEED to 100 and 18 for the asynchronous and the synchronous variant of the model, respectively. Moreover, crossing point TRAIN:CCPOS has been frozen to 150. Application of freezing drastically reduces the degree of freedom in activity TRAIN to activate CROSSING and to compute an appropriate nominal speed when approaching CROSSING.
Interestingly, while invariance checking obviously benefits from this restriction, bounded model checking is aggravated by narrowing the possible solutions.

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	49	5m	TO!12h	31i/270s
asynch1	49	5m	TO!12h	31i/268s
asynch0-cnt9	-	k=100 too small(1m)	TO!12h	31i/273s
asynch1-cnt9	-	k=100 too small(1m)	TO!12h	31i/271s
synch0	25	5sec	3h30m	31i/225s
with freezing:				
asynch0	53	15m	5m	15i/254s
asynch1	53	15m	5m	15i/252s
synch0	25	5sec	1m	15i/225s

Table 8.13: **Check 2:** Drive to Property: 'CROSSING_CTRL:OPENING_GATES and CROSSING:-BARRIER_OPENING and primed(CROSSING_CTRL:IDLE)' without freezing and with SYSTEM:D_SPEED frozen to 100 (for asynchronous model) and 18 (for synchronous model) , respectively, and TRAIN:CPPOS frozen to 150

Check_3, check_4, check_5 and check_6 together are aimed at obtaining witness for the possibilities of TRAIN to pass CROSSING according to the protocol. Check_4 is aimed at obtaining a witness for receiving a SYSTEM:CROSSING_FREE_SND from TRAIN, due to a timeout in TRAIN, because TRAIN has approached CROSSING too slow. Check_5 checks for reachability of the situation that CROSSING receives a SYSTEM:STATUS_RQ_REC from TRAIN right in the moment, when CROSSING is ready to answer the status request. Finally, check_6 checks for possibility of a timeout in CROSSING (indicated by TIMEOUT_OPCENTER), which ends CROSSING's willingness to answer a status request from TRAIN. Since CROSSING's ability to answer a status request is disabled, if the maximum barrier closed time (MBCT) elapses before receiving CROSSING:PASSED or SYSTEM:CROSSING_FREE_REC, successful application of checks 3, 4, 5 and 6 justifies the modification of MBCT for model *asynch1*.

Check 3: Drive to Property: 'CROSSING_CTRL:BARRIER_CLOSED and
primed(CROSSING_CTRL:OPENING_GATES) and CROSSING:PASSED and

not(SYSTEM:CROSSING_FREE_REC)'
Check_3 and check_4 focus on different configurations for taking compound transition CROSSING_CTRL:BARRIER_CLOSED→CROSSING_CTRL:OPENING_GATES in statechart CROSSING_CTRL (figure 3.10) of activity CROSSING. The examined transition is triggered by a disjunction of two events. Event CROSSING:PASSED is driven by statechart SENSOR_CTRL and depends on free input SYSTEM:SENSOR_ON. Event SYSTEM:CROSSING_FREE_REC is driven by COMMUNICATION reacting on SYSTEM:CROSSING_FREE_SND, which is issued by TRAIN indicating a timeout when approaching a secured CROSSING.
Check 3 focus on receipt of CROSSING:PASSED in absence of SYSTEM:CROSSING_FREE_SND

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	45	2m	TO!12h	31i/274s
asynch1	45	1m	TO!12h	31i/272s
asynch0-cnt9	101→52/-	1m	TO!12h	31i/277s
asynch1-cnt9	101→52/-	1m	TO!12h	31i/275s
synch0	23	10sec	1h15m	31i/225s

Table 8.14: **Check 3:** Drive to Property: 'CROSSING_CTRL:BARRIER_CLOSED and primed(CROSSING_CTRL:OPENING_GATES) and CROSSING:PASSED and not(SYSTEM:CROSSING_FREE_REC)'

Check 4: Drive to Property: 'CROSSING_CTRL:BARRIER_CLOSED and primed(CROSSING_CTRL:OPENING_GATES) and not(CROSSING:PASSED) and SYSTEM:CROSSING_FREE_REC'
In contrast to check_3, check_4 is aimed at producing a witness for TRAIN being stopped by a timeout when approaching an already secured crossing, because TRAIN approaches CROSSING not fast enough.

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	54	25m	TO!12h	31i/270s
asynch1	54	15m	TO!12h	31i/268s
asynch0-cnt9	-	k=100 too small	TO!12h	31i/273s
asynch1-cnt9	-/(112→56)	k=100 too small	TO!12h(36h)	31i/271s
synch0	30	20sec	TO!12h(24h)	31i/225s

Table 8.15: **Check 4:** Drive to Property: 'CROSSING_CTRL:BARRIER_CLOSED and primed(CROSSING_CTRL:OPENING_GATES) and not(CROSSING:PASSED) and SYSTEM:-CROSSING_FREE_REC'

Check 5: Drive to Property: 'last(CROSSING_CTRL:BARRIER_CLOSED) and CROSSING_CTRL:BARRIER_CLOSED and SYSTEM:STATUS_RQ_REC'
CROSSING only responds to a status request when in state CROSSING_CTRL:BARRIER_CLOSED,

because response to a status request is modeled using a static reaction of this state. This static reaction is executed only if CROSSING_CTRL:BARRIER_CLOSED (1) was active in the last step and (2) is active in the actual step and (3) a status request is received. By checking for reachability of this situation, evidence for correctness of essential timing is established: When TRAIN reaches a designated activation point, communication with CROSSING is initiated and CROSSING is sent an activation command. Once CROSSING is activated, CROSSING_CTRL commands the lights to be turned on and the barriers to be lowered. State CROSSING_CTRL:BARRIER_CLOSED is entered only after LIGHT_CONTROL and BARRIER_CTRL have reported successful completion of these commands. Meanwhile, TRAIN simply waits for a particular time - defined by constant CCT - before sending a status request.
Besides other aspects of this protocol, check 5 gives evidence for the correct value of CCT. Since TRAIN send a status request only once, CCT must be large enough to ensure that CROSSING_CTRL can enter CROSSING_CTRL:BARRIER_CLOSED before receiving the request. Using this check we were able to determine the value '12' of CCT for the synchronous model. With the original value of '8' , the synchronous model was unable to meet the desired protocol.
State CROSSING_CTRL:BARRIER_CLOSED is left after "maximum barrier closed time" (MBCT). Using check 5 we assured ourselves that reducing MBCT to '8' (for model *asynch1*) instead of the original value '40' (model *asynch0*) does not disable the protocol in general.

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	48	5m	TO!12h	31i/271s
asynch1	48	3m	TO!12h	31i/269s
asynch0-cnt9	-/102→50	k=100 too small(30sec)	3h	31i/274s
asynch1-cnt-9	-/102→50	k=100 too small(30sec)	50m	31i/272s
synch0	26	10sec	2h	31i/226s

Table 8.16: **Check 5:** Drive to Property: `'last(CROSSING_CTRL:BARRIER_CLOSED) and CROSSING_CTRL:BARRIER_CLOSED and SYSTEM:STATUS_RQ_REC'`

Check 6: Drive to State: `'CROSSING_CTRL:TIME_OUT'`

This check is mainly aimed at justifying the reduction of MBCT from '40' of the original model *asynch0* to '8' in models *asynch1* and *asynch3*. As discussed in the context of the stabilization check, this modification aims at tackling verification complexity, but has to preserve the reachability of all states and preserves the behavior of the model w.r.t. the specified protocol in general. Since state CROSSING_CTRL:TIME_OUT is reachable in *asynch0*, the state has to be reachable also in *asynch1* and *asynch3* as well as in the synchronous variant *synch0* of the model. Table 8.17 summarizes results and run-times for check_6:

Model	Trace-Length	Time bmc	Time inv	Model Size
asynch0	-	TO!12h	TO!12h	31i/270s
asynch1	67	25m	TO!12h	31i/268s
asynch0-cnt-9	-	k=100 too small	TO!12h	31i/273s
asynch1-cnt9	-	k=100 too small	TO!12h	31i/271s
synch0	64	8m	TO!12h	31i/225s
with freezing:				
asynch0	173	TO!12h	5m	4i/212s
asynch1	77	18m	2m	4i/209s
asynch3	67	2m	4m	31i/219s
synch0	66	30sec	1m	4i/194s

Table 8.17: **Check 6:** Drive to State: `'CROSSING_CTRL:TIME_OUT'` without freezing and with freezing according to table 8.18 plus frz1 or frz2, respectively, of table 8.19.

Check 'normal run': Drive to Property: `'TRAIN:ODATA.POS>250'`

This check is aimed at obtaining witnesses for `TRAIN` reaching the 'end of the track'. Track positions are modeled using natural numbers in the range from 0 to 255. For all runs, freezing `TRAIN:CPPOS=150` (cf. table 8.18) locates `CROSSING` at position 150. Hence, `'TRAIN:ODATA.POS>250'` specifies a situation where `TRAIN` has passed `CROSSING` and approaches the end off the track. Since `TRAIN` pays no attention to the crossing position as desired, if `CROSSING` is treated as 'already regarded', an assumption (cf. ass1 in table 8.19) is required, assuming that input `ACTIVATE_CROSSING_CTRL:CP_REG_INP` may be set only after `ACTIVATE_CROSSING_CTRL:IDLE` has been entered[3]. The check is expected to obtain witnesses for `TRAIN` passing a secured `CROSSING` without being stopped before and without manual release of `CROSSING`, due to malfunctions of lights or barriers (cf. table 8.18). Since `TRAIN` must automatically compute an appropriate speed - depending on the desired speed - in order not to overrun `CROSSING` but also not to be stopped before it, driver interaction by changing the desired speed has been disabled using freezing or assumptions.

As already mentioned, the interpretation of speed differs between the asynchronous models and the synchronous model. For the synchronous model `SYSTEM:D_SPEED=18` turned out to be a meaningful choice in order to reach `TRAIN:ODATA.POS>250`. For the asynchronous model, choosing 100 has been successful.

Very interesting witnesses could be obtained by not freezing `SYSTEM:D_SPEED` but assuming to be an *arbitrary but fixed* value. Assuming an arbitrary but fixed value for `SYSTEM:D_SPEED`, forces the verification engine to determine the adequate speed in order to reach the end of the track without being stopped but within as few steps as possible. It is thus let to the verification engine to solve also an optimization problem. This could be achieved by using assumption *ass2* (cf. table 8.19), which allows `SYSTEM:D_SPEED` to be chosen arbitrarily in step 0, but not to change for all subsequent steps. The computed value for `SYSTEM:D_SPEED`

[3] `TRAIN:CP.ALREADY_REGARDED` is assigned the value of input `ACTIVATE_CROSSING_CTRL:CP_REG_INP` when entering `ACTIVATE_CROSSING_CTRL:IDLE`. Only if `TRAIN:CP.ALREADY_REGARDED` is not true, the function `COMPUTE_NOMINAL_SPEED` computes values according to the desired braking-curve in order to stop the `TRAIN` before passing an unsecured `CROSSING`. Assumption *ass1* enforces `TRAIN:CP.ALREADY_REGARDED` to be false until reentering `ACTIVATE_CROSSING_CTRL:IDLE`, which can only happen after the `CROSSING` has been passed.

is listed for all results referring to assumption *ass2* in table 8.20 in the column Trace-length. Moreover, the table lists the results and run-times for all different combinations with assumptions and freezing.

Freezing for all variants of check 'normal run':

Input	Value
TRAIN:CPPOS	150
TIMER_CTRL:V_STILL_SAFE_P	true
SYSTEM:V_BRAKE_POINT_P	false
SYSTEM:SENSOR_ON	false
SYSTEM:RED:ERROR	false
SYSTEM:YELLOW_ERR	false
SYSTEM_RELEASED_MAN	false
SYSTEM:CROSSING_VACATED	false

Table 8.18: Freezing for check 'normal run'

The freezing for all variants of check 'normal run', disable indication of light errors by the free inputs `RED_ERROR` and `YELLOW_ERR`. Furthermore, detection of a `TRAIN` spontaneously passing `CROSSING` is disabled (`SENSOR_ON`). `CROSSING_VACATED` and `RELEASED_MAN` are free inputs aimed at resetting `TRAIN` and `CROSSING` after emergency stop and timeouts, respectively. `V_STILL_SAFE_P` and `V_BRAKE_POINT_P` are abstractions from internal computations of `TRAIN` by inputs, used in the computation of the braking curve. These freezing are referred to by '*FRZ*' in table 8.20.

Additional assumptions and freezing for the different variants of check 'normal run' :

ass1	**init_Q_only_after_P_immediate** P= ACTIVATE_CROSSING_CTRL:IDLE Q= ACTIVATE_CROSSING_CTRL:CP_REG_INP	*effect:* do not treat Crossing Position as "already regarded"
ass2	**inv_P__after_N_steps** P = SYSTEM:D_SPEED==last(SYSTEM:D_SPEED) N = 1	*effect:* chose D_SPEED arbitrarily but fixed
frz1	SYSTEM:D_SPEED frozen to **100**	for asynchronous models
frz2	SYSTEM:D_SPEED frozen to **18**	for synchronous mode

Table 8.19: Assumptions for check 'normal run'

Runs :

For application of bounded model checking to the counter approximation models *asynch0-cnt9* and *asynch1-cnt9*, the maximal unroll depth has been chosen 200.

Model	Trace-Length	assumptions	Time bmc	Time inv	Model Size
asynch0	92/92 97/n.a.	FRZ,ass1, frz1	1h10m (k=96:3m30sec)	2m30sec	8i/213s
asynch0	-/94:D_SPEED=15	FRZ,ass1, ass2	TO!12h	1h20m	16i/229s
asynch1	94/92 97/n.a.	FRZ,ass1 frz1	1h12m (k=96:2m)	2m	8i/211s
asynch1	-/94:D_SPEED=15 97:D_SPEED=15/n.a.	FRZ,ass1, ass2	TO!12h (k=96:5m)	1h20m	16i/226s
asynch0-cnt9	191→94/191→95	FRZ,ass1,frz1	30sec	1m45sec	8i/216s
asynch0-cnt9	191→93:D_SPEED=224/ 191→93:D_SPEED=15	FRZ,ass1, ass2	13m	6m	16i/232s
asynch1-cnt-9	191→94/191→95	FRZ,ass1,frz	30sec	1m45sec	8i/214s
asynch1-cnt-9	191→94:D_SPEED=224/ 191→95:D_SPEED=15/n.a.	FRZ, ass1,ass2	13m	6m	16i/226s
synch0	44	FRZ,ass1, frz2	30sec	45sec	8i/193s
synch0	44 D_SPEED=18/D_SPEED=19	FRZ,ass1, ass2	1m	3m	16i/201s

Table 8.20: Results of **check 'normal run'**

Check 'normal run' reveals that invariance checking can sometimes be advantageous over bounded model checking. When the determination of `SYSTEM:D_SPEED` has been left to the verification engine, bounded model checking has been significantly slower than invariance checking and obtained in 2 of 5 cases no result in reasonable time for the first trial. The reason is that only few solutions exist regarding the involved variables and timing of events for which `TRAIN` reaches the end of the track - as can be seen from the results for the counter approximated models. In particular dynamic stabilization of super-steps obviously aggravates finding a solution. As can be seen from the results, the shortest witness reflecting a 'normal run' is of length 92 steps. Hence, the bounded model checker has to unroll the transition relation of the system-model 92 times in order to obtain a solution. The situation looks different if bounded model checking is applied with an already 96 times unrolled model. In this case a solution has been found within a time comparable to invariance checking (rows 1,3,4 of table 8.20).

8.1.6 Summary of the Application of Formal Debugging

Application of formal debugging proved to be a powerful technique in order to obtain witnesses and simulations driving the system into specific situations of interest. In most cases, application of bounded model checking has been successful, whereas invariance checking often suffered from complexity. Precisely because formal debugging checks are in general expected to reach the specified situation and to obtain a witness, bounded model checking is the recommended technique.

It has been presented, how formal debugging checks can be applied to examine scenarios leading to particular situations in a protocol. The results bear evidence for reachability of the specified system states. Application of formal debugging obtained in all cases *'high quality custom-specific'*

simulations, driving the system to user-defined configurations of the system. Creating such simulations by interactive simulation would require a considerable effort, because in interactive simulation the user has to provide the system with the appropriate inputs exactly at the right instances of time, even if events are involved, which are visible only for one step.

8.2 Application of Verification using Observer Pattern

In this section, we present briefly the application of verification using predefined observer pattern, as offered by the STVE. The requirements, for which verification has been performed, have been taken from [DK01]. There, 5 sample requirement specifications have been listed :

P1 : The train only passes a secured railway crossing

P2 : The barrier is only opened after the train has passed the crossing.

P3 : The yellow light is only activated after the traffic lights have been switched on.

P4 : The red light is only activated after the yellow light.

P5 : The minimum green time is respected.

In [DK01], the requirements are formalized using symbolic timing diagrams, for which temporal logic formulae have been generated. Verification has been applied to the sub-activities, which are responsible for the realization of the respective requirement. In contrast, we applied all verification tasks also to the system-model. Therefore, two variants of propositional abstraction have been employed: First, '*automatic propositional abstraction*' as presented in [Bie03], which is a comfortable method that does not require any expert knowledge. In the second variant of propositional abstraction the user is prompted for selection of local variables to be treated abstractly. In all proofs it has been abstracted (by syntactic existential quantification) from all user-defined local variables except for the variables referred to by the pattern.

P2 is a requirement regarding activity `CROSSING`, which has been checked once for the activity in isolation and also for the entire system-model.

The verification tasks for requirements P3, P4 and P5 have been applied to `CROSSING` instead of only to `CROSSING`'s sub-activity `LIGHTS_CONTROL_CTRL`. Finally P1 has been checked for activity `TRAIN` as well as for `SYSTEM`.

Regarding the synchronous variant of the case-study, the execution-times for verification are comparable to the results of [DK01]. Since the focus of this work is put on the asynchronous execution semantices, we furthermore present verification results for the more complex super-step variant of the case-study.

As there exists no system-wide indication of "train passing the crossing", requirement P1 can only be formalized referring to the perception of `TRAIN`[4]. If we would refer to `CROSSING:PASSED` we would have to take `SYSTEM:SENSOR_ON` into consideration (`SYSTEM:SENSOR_ON` is a free input which must appropriately set by the environment!). Correct functionality for `SYSTEM:SENSOR_ON` can only be established at the costs of a highly non-trivial assumption. In this case we would have mainly verified the assumption but not considered reactions of the model. On the other hand, if `SYSTEM:PASSED_XING` is referred to as indication, the specification captures only the perception of `TRAIN`.

[4] We refer the reader to section 3.3 for the explanation of the model.

Formalisation of the Requirements

Requirements P1-P4 are already formulated in the form 'event A may happen *only after* event B'. Hence, an appropriate formalization can make use of a predefined 'Q_onlyafter_P' pattern.

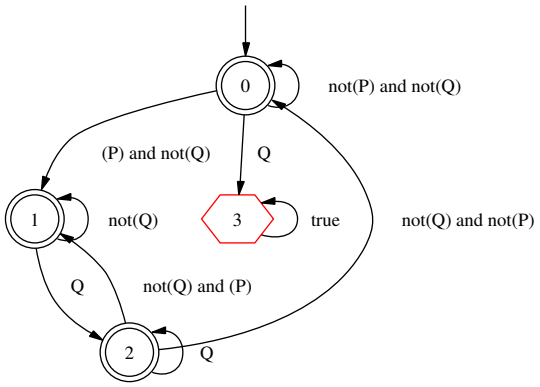


Figure 8.3: Pattern iter_Q_onlyafter_P__immediate

P5 does not require a relationship of events, but is an invariant, which is required to hold always for all computations of the model. Hence, an appropriate pattern is 'inv_P_immediate':

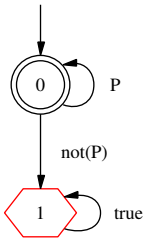


Figure 8.4: Pattern inv_P__immediate

In the tables below, the columns are marked as follows:

I denotes application of '*automatic propositional abstraction*' [Bie03]

II denotes user defined propositional abstraction (cf. section 7.2).

III denotes verification without propositional abstraction; verification has been applied only with COI computation

All proofs were performed on two dual processor SunOS 5.8 Blade 1000 work stations, with each 2GB memory and 900MHz SPARC processors. All applications of the automatic propositional abstraction succeeded in first iteration.

P1: The train only passes a secured railway crossing

```
p1_pattern = iter_Q_onlyafter_P__immediate
prop P : (SYSTEM:RELEASED_MAN) or (SYSTEM:CROSSING_SAFE_REC);
prop Q : (SYSTEM:PASSED_XING);
```

SYSTEM:RELEASED_MANis an input of SYSTEM, which enables TRAIN to pass a CROSSING that did not answer a status request from TRAIN. In this case TRAIN treats the CROSSING as being faulty, stops before CROSSING and can only pass after manual release. SYSTEM:CROSSING_-SAFE_REC is the expected normal reaction of CROSSING to a status request of TRAIN. If after activation of CROSSING securing is completed, CROSSING emits event CROSSING_SAFE_SND to COMMUNICATION, which transmits the information by emitting CROSSING_SAFE_REC to TRAIN. SYSTEM:PASSED_XING is an output of TRAIN. Using this event, TRAIN indicates that a CROSSING position according to the track data has been passed.
On page 229 we have described the back-door effect regarding the input ACTIVATE_CROS-SING_CTRL:CP_REG_INP, which lets TRAIN consider CROSSING as already regarded. The authors of [DK01] did not state an assumption regarding ACTIVATE_CROSSING_CTRL:CP_REG_INP. In contrast, for verification P1 we have applied a freezing ACTIVATE_CROSSING_CTRL:CP_REG_INP to false. Nonetheless, P1 only specifies the local perception of TRAIN. In order to specify the situation of TRAIN only passing a secured CROSSING more appropriately, the specification should state that SYSTEM:PASSED_XING is only possible after CROSSING:IN_SAFE, as it is proved by the example for compositional verification using STDx in section 8.3.3. Verification of such a pattern - under the assumption that SYSTEM:RELEASED_MAN and ACTIVATE_CROSSING_CTRL:-CP_REG_INP are false forever - suffers from complexity, even for application of automatic abstraction refinement.

asynch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		2sec	35i/17s		2sec	10i/27s		TO!12h	31i/264s
Train		1sec	32i/13s		1sec	8i/19s		TO!12h	24i/144s

Table 8.21: Verification of P1 for Asynchronous Model

synch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	34i/12s		220sec	30i/95s		TO!12h	31i/218s
Train		1sec	31i/14s		40sec	28i/88s		TO!12h	24i/117s

Table 8.22: Verification of P1 for Synchronous Model

P2: The barrier is only opened after the train has passed the crossing

```
p2_pattern = iter_Q_onlyafter_P__immediate
prop P : (SYSTEM:CROSSING_FREE_REC) or (CROSSING:PASSED) or (SYSTEM:CROSSING_VACATED);
prop Q : (CROSSING_CTRL:OPENING_GATES);
```

CROSSING:PASSED is a local event of activity CROSSING, which is emitted by the sensor control activity according to a falling edge of input condition SENSOR_ON, where SENSOR_ON represents a way-side sensor. SYSTEM:CROSSING_VACATED is a free input of the system, which can be used to reset CROSSING. Finally SYSTEM:CROSSING_FREE_REC is the transmission event from

COMMUNICATION which is emitted in reaction to SYSTEM:CROSSING_FREE_SND from TRAIN. The formal parameter Q of the pattern is mapped to CROSSING_CTRL:OPENING_GATES, which is a basic state of the crossing controller. At all transitions entering this state event OPEN_BARRIER is emitted to the barrier control, which controls the physical barriers .

asynch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	14i/23s		5sec	11i/35s		TO!12h	31i/263s
Crossing		1sec	11i/19s		n.a.	n.a.		450sec	12i/99s

Table 8.23: Verification of P2 for Asynchronous Model

synch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	12i/27s		2sec	11i/26s		TO!12h	31i/217s
Crossing		1sec	10i/19s		n.a.	n.a.		10sec	12i/78s

Table 8.24: Verification of P2 for Synchronous Model

P3: The yellow light is only activated after the traffic lights have been switched on

```
p3_pattern = iter_Q_onlyafter_P__immediate
prop P : (CROSSING:TURN_LIGHTS_ON );
prop Q : (SYSTEM:SWITCH_ON );
```

CROSSING:TURN_LIGHTS_ON is a local event of CROSSING, which is used in the communication between activity CROSSING_CTRL and LIGHTS_CONTROL_CTRL, while SYSTEM:SWITCH_ON is an output, which has been added only for external observability.

asynch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	20i/33s		10sec	34i/45s		TO!12h	31i/264s
Crossing		2sec	33i/20s		n.a.	n.a.		240sec	12i/99s

Table 8.25: Verification of P3 for Asynchronous Model

synch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	28i/21s		2sec	14i/21s		TO!12h	31i/218s
Crossing		2sec	28i/21s		n.a.	n.a.		12sec	12i/79s

Table 8.26: Verification of P3 for Synchronous Model

P4: The red light is only activated after the yellow light

```
p4_pattern = iter_Q_onlyafter_P__immediate
prop P : (SYSTEM:SWITCH_ON );
prop Q : (SYSTEM:SWITCH_OVER );
```

Both, `SYSTEM:SWITCH_ON` and `SYSTEM:SWITCH_OVER` are output events, which have been added for external observation.

asynch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	24i/11s		2sec	14i/27s		TO!12h	31i/264s
Crossing		1sec	24i/11s		n.a.	n.a.		306sec	12i/99s

Table 8.27: Verification of P4 for Asynchronous Model

synch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	20i/7s		1sec	11i/10s		TO!12h	31i/218s
Crossing		1sec	20i/7s		n.a.	n.a.		8sec	12i/79s

Table 8.28: Verification of P4 for Synchronous Model

P5: The minimum green time is respected

```
p5_pattern = inv_P__immediate
prop P : not(SYSTEM:SWITCH_ON and LIGHTS_CONTROL_CTRL:GT < LIGHTS_CONTROL_CTRL_DD_MGT);
```

Again, `SYSTEM:SWITCH_ON` is an output, which has been added for external observation. This event is emitted at a transition in activity `LIGHTS_CONTROL_CTRL`. `LIGHTS_CONTROL_CTRL:GT` is a counter, which reflects the duration of the current green time, which `LIGHTS_CONTROL_CTRL_DD_MGT` is a constant.

asynch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	16i/18s		2sec	12i/35s		TO!12h	31i/264s
Crossing		1sec	16i/18s		n.a.	n.a.		400sec	12i/99s

Table 8.29: Verification of P5 for Asynchronous Model

synch0	I	Time	ModelSize	II	Time	ModelSize	III	Time	ModelSize
System		1sec	12i/20s		1sec	11i/24s		TO!12h	31i/218s
Crossing		1sec	12i/20s		n.a.	n.a.		15sec	12i/79s

Table 8.30: Verification of P5 for Synchronous Model

8.2.1 Summary of Application of Observer Pattern Verification

The example applications of pattern based verifcation demonstrate that important requirements can be captured using observer pattern and that verification for these specifications can be applied efficiently. In particular, using the automatic propositional abstraction refinement [Bie03] capability of the STVE obtained optimal results for all of the requirements. An advantage of this technique is that the user does not need to have any knowledge about the technique and that the used needs not to choose variables for abstraction. The only expert knowledge in pattern based verification is required for selection of the appropriate pattern and mapping of concrete expressions to the formal parameters of the selected pattern.

Of course, not all specifications of interest can be formalized using observer pattern, even though additional specific observer pattern can be added (and have been added during the time of writing) to the pattern library. But there are still limitations for this approach, like the amount of formal parameters in the pattern definition or the flexibility with which pattern can specify real-time requirements. Experiences have shown that nearly all safety-critical requirements for a large amount of models from project-partners and customers can be captured by the offered pattern library, while only about maximal 50% of the functional requirements could be specified using pattern [Hol05].

8.3 Application of Verification using Symbolic Timing Diagrams

In this section, we illustrate the application of verification using Symbolic Timing Diagrams with three examples. The first example is a component proof for activity `ACTIVATE_CROSSING_CTRL` of activity `TRAIN`. This example demonstrates the combined usage of step and super-step constraints within one specification.

The second application example demonstrates compositional verification of a real-time property of activity `CROSSING`. It is shown that `CROSSING` always will report itself safe, if the status request is received within a certain time-interval after activating `CROSSING` and provided that hardware errors are absent and that neither the train nor the environment aborts the securing process.

Finally, the third application example demonstrates a typical use-case of compositional verification for pure safety properties. Using a series of un-timed specifications regarding the order of events, it is proved by compositional reasoning from specifications of the sub-activities of `SYSTEM`, that `TRAIN` can never pass an unsecured `CROSSING`.

In order to ease the understanding of the presented diagrams, we have replicated some screenshots of the specified activities. For a more detailed explanation regarding the depicted activities, the reader is referred to section 3.3 for explanations regarding the illustrations.

Remark 8.1

In the diagrams presented in this sections, the names of STATEMATE*-variables appear with the suffixes '_F' and '_IN_S', respectively. This is due to the interface representations using SSL, where* fast *interface objects (inputs as well as outputs) are marked with suffix '_F', while* slow *inputs from the environment are marked with the suffix '_IN_S', in order to emphasize the origin of the respective observable.*

■

8.3.1 Component Proof for ACTIVATE_CROSSING_CTRL

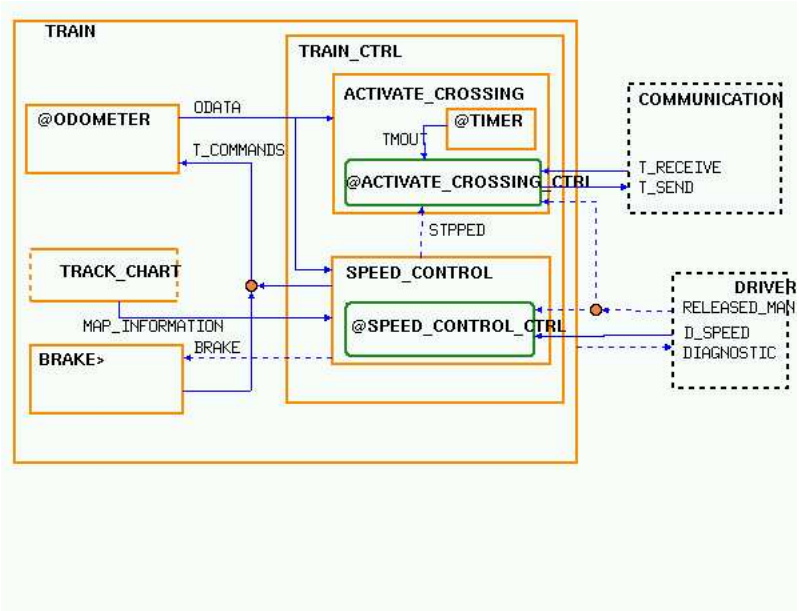


Figure 8.5: Activity TRAIN

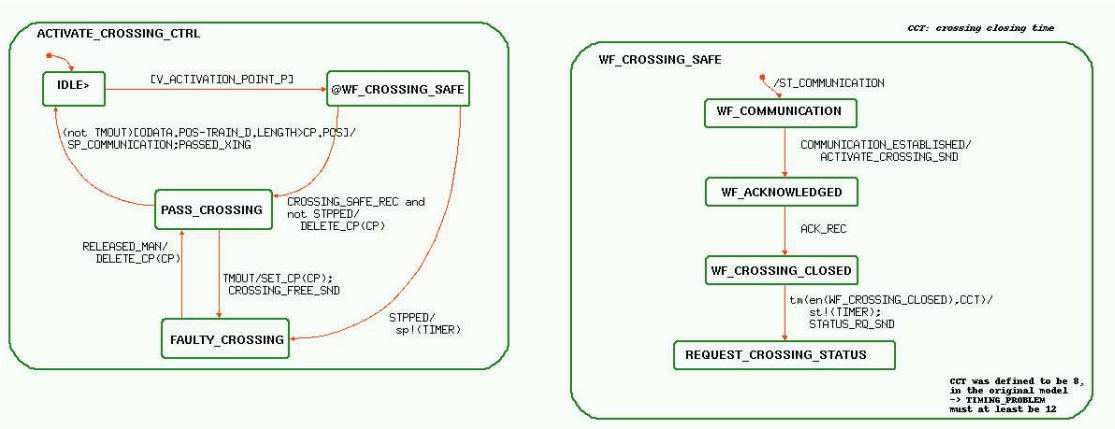


Figure 8.6: Statecharts ACTIVATE_CROSSING_CTRL and WF_CROSSING_SAFE

Activity ACTIVATE_CROSSING_CTRL is an inner activity of TRAIN which is responsible for establishing of the communication link and activation of CROSSING. Once an activation point has been signalized, statechart WF_CROSSING_SAFE is entered and ST_COMMUNICATION is sent to COMMUNICATION. After COMMUNICATION indicates an established radio link, CROSSING is activated and CCT (=8) time units after receipt of the corresponding acknowledge, a status request is sent to CROSSING. The entire securing process is aborted if the sibling activity SPEED_CONTROL_CTRL emits a STPPED event indi-

cating an emergency break. Hence, provided that no emergency break stops the train and provided that COMMUNICATION and CROSSING react appropriately, a status request has to be send exactly 8 super-steps after receipt of ACK_REC. Correct behavior of ACTIVATE_CROSSING_CTRL according to this scenario is specified by commitment snd_st_rq_after_act (cf. figure 8.7).

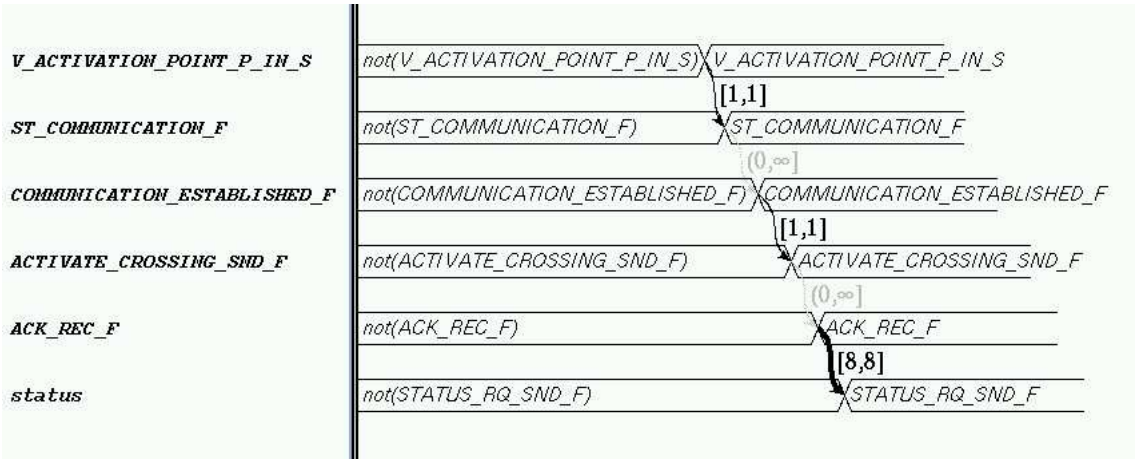


Figure 8.7: ACTIVATE_CROSSING_CTRL : Commitment snd_st_rq_after_act

Commitment snd_st_rq_after_act specifies an ordered and timed sequence of events with which ACTIVATE_CROSSING_CTRL reacts to setting of an activation point by the environment. The diagram illustrates the necessity of combining step and super-step constraints in the specification of subsystems.

It is committed that ACTIVATE_CROSSING_CTRL reacts on V_ACTIVATION_POINT_P_IN_S by emitting event ST_COMMUNICATION_F to activity COMMUNICATION in the subsequent step. In order to provide the diagram unwinding algorithm with an ordering information and this way to avoid creation of all possible combinational paths in the TSA representation, *possible* constraints are used in the diagram, e.g. to capture the expectation about that COMMUNICATION_ESTABLISHED_F will be observed only after emission of ST_COMMUNICATION_F.

Once the communication link is established snd_st_rq_after_act guarantees that in the subsequent step ACTIVATE_CROSSING_SND_F is emitted - which is translated into ACTIVATE_CROSSING_REC_F by COMMUNICATION and communicated to CROSSING.

Adherence to the specified sequence of events depends on five assumptions, which are documented on the following pages.

Assumption ass_comm (figure 8.8) captures the assumption that COMMUNICATION will react on this event with event COMMUNICATION_ESTABLISHED_F within the subsequent super-step. The assumption reflects the fact that the reaction of COMMUNICATION takes place in the action part of a timeout-triggered transition (with constant ELT=1, cf. figure 8.65).

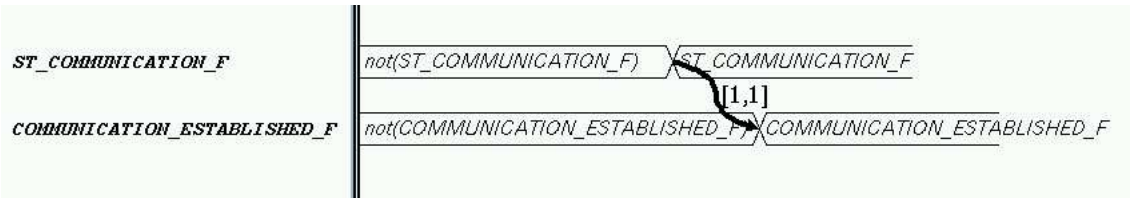


Figure 8.8: ACTIVATE_CROSSING_CTRL : Assumption ass_comm

Since COMMUNICATION reacts on ST_COMMUNICATION_F by entering state WAIT_FOR_CONNECTION (which is left by timeout ELT=1 after entering the state with reaction COMMUNICATION_ESTABLISHED_F), we can safely assume that COMMUNICATION reacts on the first ST_COMMUNICATION_F with emitting COMMUNICATION_ESTABLISHED_F in the next super-step.

In order to guarantee sending a STATUS_RQ_SND_F it is assumed that CROSSING reacts on ACTIVATE_CROSSING_REC_F by emitting ACK_SND_F in the subsequent step (which again is translated into ACK_REC_F by COMMUNICATION and communicated to ACTIVATE_CROSSING_CTRL). This assumption about COMMUNICATION and CROSSING is captured by assumption ass_cross_and_comm (cf. figure 8.9). Since forward and backward translation and communication is performed using fast communication once the communication link has been established, this entire portion of the protocol takes place within one super-step. COMMUNICATION will take one step to emit ACTIVATE_CROSSING_REC_F, CROSSING will take one step to react on ACTIVATE_CROSSING_REC_F by emitting ACK_SND_F and again COMMUNICATION will need one step to emit ACK_REC_F as reaction. Hence, we assume that the appropriate reaction on ACTIVATE_CROSSING_SND_F will be received 3 steps after emission of the event.

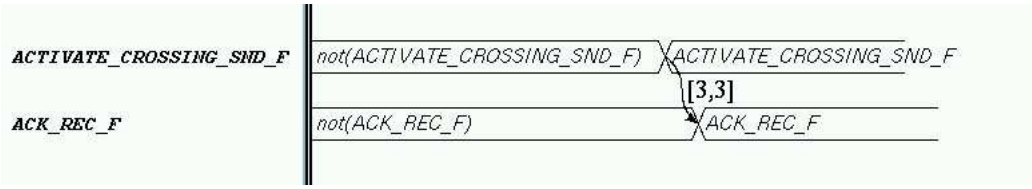


Figure 8.9: ACTIVATE_CROSSING_CTRL : Assumption ass_cross_and_comm

ass_cross_and_comm specifies the assumption that COMMUNICATION as well as CROSSING and again COMMUNICATION will react as expected after the communication link is established. Obviously, this assumption is critical for the presented component proof and only a compositional proof could establish validity of this assumption. In the context of this work, we rely on the validity of this assumption and confide in critical inspection of COMMUNICATION and CROSSING.

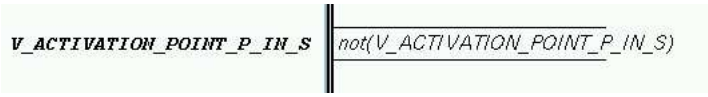


Figure 8.10: ACTIVATE_CROSSING_CTRL : Assumption ass_ext

Assumption ass_ext is required only for enabling the activation condition of snd_st_rq_after_act, because receipt of V_ACTIVATION_POINT_P_IN_S in step 0, would violate the activation condition of commitment snd_st_rq_after_act.

Figure 8.11: ACTIVATE_CROSSING_CTRL : Assumption crossing_safe

Since nothing concrete is known about the environment form the perspective of activity ACTIVATE_CROSSING_CTRL, CROSSING_SAFE_SND_F can be emitted by the environment spontaneously without any correspondence to the expected protocol. From inspection of the model, it can be put as fact that CROSSING_SAFE_REC_F can only be observed if WF_CROSSING_SAFE has emitted STATUS_RQ_SND_F first (provided that COMMUNICATION does not produce arbitrary events - which would be an unforeseen error-mode of the model) .

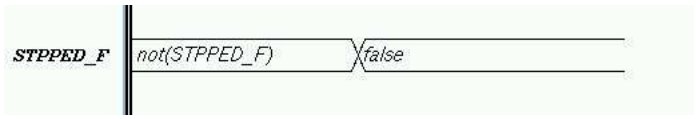


Figure 8.12: ACTIVATE_CROSSING_CTRL : Assumption stpped_fals

In order to focus on the scenario that ACTIVATE_CROSSING_CTRL is not interrupted by an emergency break before sending a status request, it has to be assumed that no STPPED_F aborts waiting for the status report from CROSSING. Notice, that STPPED_F is a free and unrestricted input from the perspective of ACTIVATE_CROSSING_CTRL.

Component-Proof of snd_st_rq_after_act of ACTIVATE_CROSSING_CTRL

The following table shows the size of the model, and the sizes of the assumption and the commitment observer modules in terms of input and state bits as well as the time needed for performing invariance checking on the parallel composition of the model with the observer modules. In this and the following tables of this section, BDD-Nodes denotes the size of the transition relation for forward image computation. Image-Comps denotes the number of image computations performed by the VIS model checker for specification verification using invariance checking.

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
71i/38s	15i/34s	13i/19s	50sec	13516	62

Specified by snd_st_rq_after_act and proved to be true is the fact that ACTIVATE_CROSSING_CTRL (and hence activity TRAIN) always sends a STATUS_RQ_REC_F 8 time units (super-steps) after receipt of an acknowledge for activation from CROSSING, provided that no emergency break interrupts the

activation of CROSSING by ACTIVATE_CROSSING_CTRL. According to [KT00], "the train sends the status request after waiting an amount of time which corresponds to the time needed for crossing to carry out the securing procedure." In contrast to this textual specification, CCT only captures a mean closing time.

It will turn out in the subsequent example that CROSSING guarantees a safe-report only if STATUS_RQ_REC_F is received at least 13 time units (super-steps) after receipt of ACTIVATE_CROSSING_REC_F. Even though CROSSING may react also on earlier status requests, the maximal duration of the securing procedure amounts to 13 super-steps after which a safe-report can be guaranteed (under the assumptions explained below) . Hence, the proof of snd_st_rq_after_act in combination with the following compositional proof for CROSSING reveals that timer constant CCT has been chosen too small in order to guarantee that CROSSING can always appropriately react on STATUS_RQ_SND_F.

8.3.2 Compositional Verification of CROSSING

The compositional proof presented in the sequel guarantees that CROSSING will - under particular circumstances - always report itself safe, if TRAIN sends its status request within a particular time-interval after activation of CROSSING. In order to establish this guarantee, it is assumed that :

- no hardware mal-function detains CROSSING from normal reaction, i.e.
  - no light is defect,
  - the barrier closes within a specified time-interval after receiving a lower-command, and
  - the sensor does not indicate an error.
- neither the sensor indicates a passed train spontaneously, nor TRAIN aborts the securing process spontaneously by sending a crossing-free signal until the maximal relevant point in time.

Provided that these assumptions hold, CROSSING will report itself safe, if a status request is received at earliest 13 time units after receipt of an activation request and latest 45 time units after activation, where 13 time units is the maximum sum of all delays in activity LIGHTS_CONTROL_CTRL plus the maximum delay between a close-barrier command from CROSSING_CTRL and the input CLOSED_IN_S indicating the physical barriers closed. The maximum of 45 time-units originates from the minimal sum of these delays plus the time triggered transition exiting state BARRIER_CLOSED in CROSSING_CTRL indicating a timeout from the operation center.

Specification of Activity CROSSING

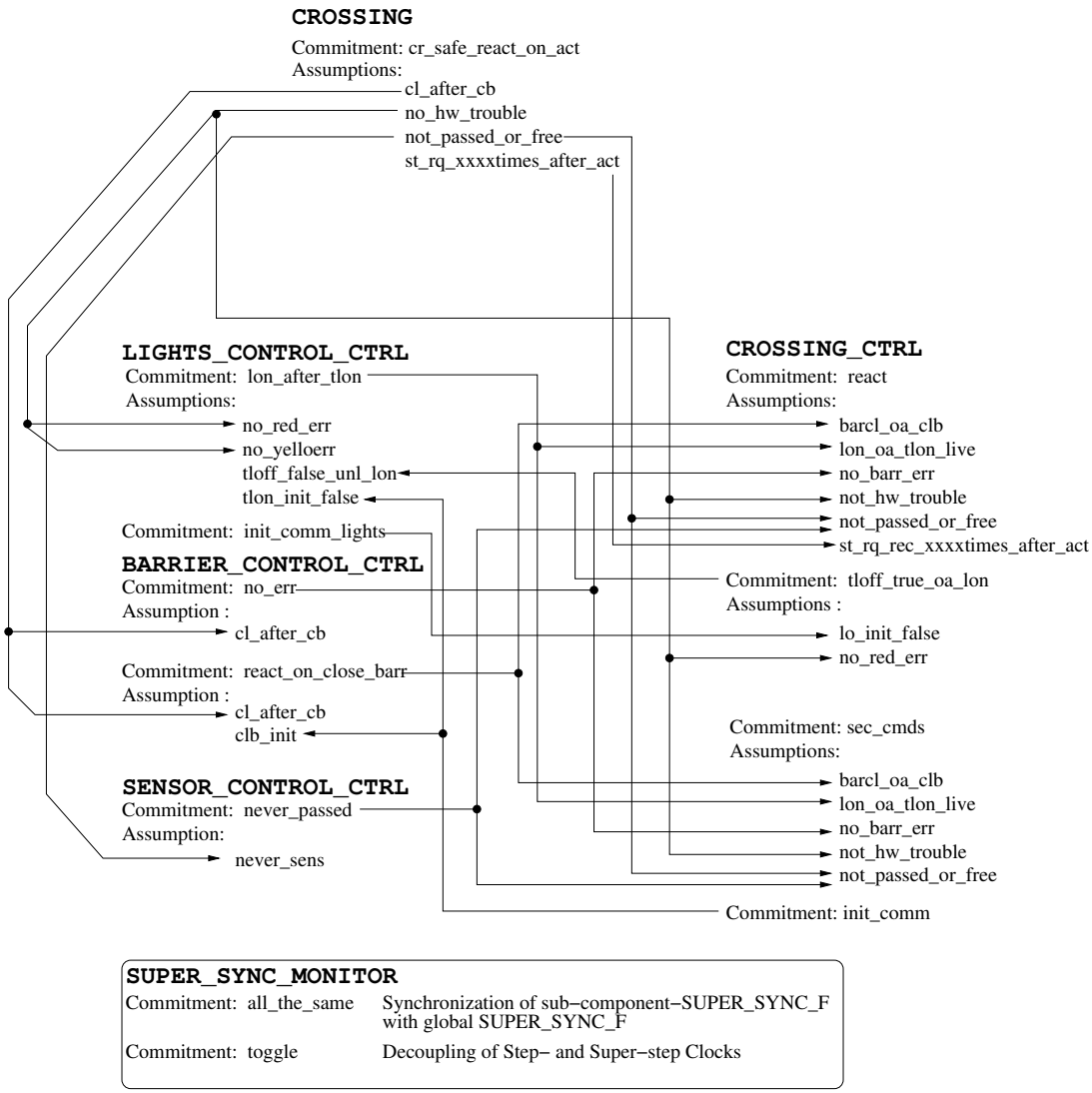


Figure 8.13: CROSSING: Dependences in Compositional Proof

Figure 8.13 gives an overview of the dependences among the specifications in the compositional proof, where the lines denote dependences of local assumptions on local commitments and top-level assumptions, respectively: For example, assumption barcl_oa_clb of commitment react is implied by commitment react_on_close_barr of activity BARRIER_CONTROL_CTRL. Accordingly, the dependences between local assumptions and component commitments and an top-level assumptions form the network of inter-dependences as shown in the figure. In order to prove that the implication of cr_safe_react_on_act by the conjunction of all sub-component specifications is a tautology, all the dependences in this network have to be satisfied, i.e. all local assumptions of sub-component specifications have to be fulfilled by either some sub-component commitments or by top-level assumptions.

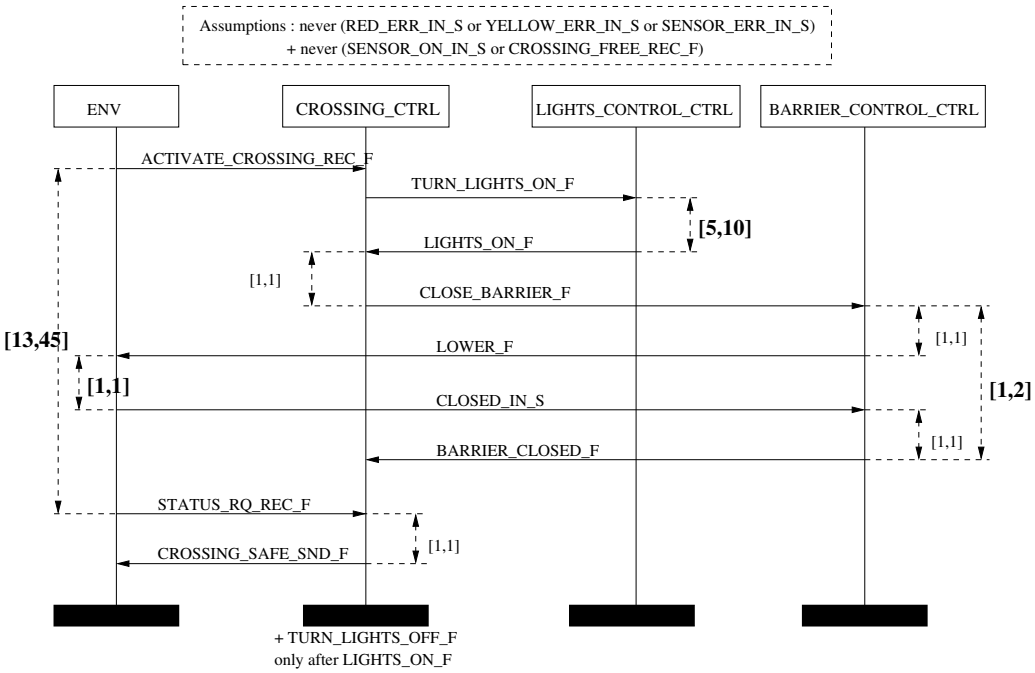


Figure 8.14: Principle Ordering and Timing of the Events contributing to the Verified Protocol

Figure 8.14 illustrates the ordering and timing of the events contributing to the compositional proof using a pseudo-Live Sequence Chart [DH99, Klo03, KW01]. The displayed chart does not strictly adhere to the formal semantics - as defined in [Klo03] - but serves only for documentation purposes. Anyhow, let sending and receiving of events (*messages*) along solid *instance* lines (the vertical lines denoting the environment of `CROSSING` and the sub-components `CROSSING_CTRL`, `LIGHTS_CONTROL_CTRL` and `BARRIER_CONTROL_CTRL` respectively) denote mandatory occurrences, where all such events along an instance line are totally ordered from top to bottom. Since messages in the chart correspond to emitting events, sending and receiving have to be interpreted as simultaneous occurrences. `SENSOR_CONTROL_CTRL` is omitted in figure 8.14, because `SENSOR_CONTROL_CTRL` contributes to the protocol only by guaranteeing that never a `PASSED_F` event is emitted, which would end the desired behavior of `CROSSING_CTRL`. Also the assumptions regarding inputs indicating hardware errors are not incorporated graphically but only textual. This also counts for the assumption that never a `CROSSING_FREE_REC_F` will be observed. Let timing intervals written in thin letters denote step intervals, whereas intervals typed in bold letters denote super-step intervals.

Compositional Proof of Commitment `cr_safe_react_on_act` of `CROSSING`

Size of parallel composition of all observers: 85i/216s (according to the construction explained in section 7.4)
Involved Diagrams: 37 (9 Sub-component commitments plus 2 commitments for monitor)

Variant	# Implications	time Implications	time Proof	BDD-Nodes	Image-Comps
subst.	21	7600sec	800sec	19598/17014	54/99
naive	-	-	TO!24h	-	-
proposed implications	330 (26 True)	43888sec	-	24106	99

The compositional verification task has been executed in two variants, a naive one and an optimized variant. In the naive variant all sub-component specifications are referred to as they are used for component verification without any modification. The naive variant suffers from the complexity of the involved specifications. In the optimized variant all sub-component assumptions have been substituted by either assumptions of `CROSSING` or by commitments of other sub-components according to the dependences shown in figure 8.13 below. The optimized proof task employing substitutions consists of two parts: First all substitutions provided for optimization are checked. In particular, it is checked whether fulfillment of the local assumptions to be substituted by the local commitments provided as substitution is a tautology. Hence checking the provided 21 substitution rules consists of 21 tautology checks. Afterwards the substituted proof-obligation for establishing the top-level commitment by tautology checking is proved.

As described in section 7.3, compositional proof-obligation construction optionally provides the user with a list of possible substitutions[5]. The proof-obligation generator constructed 330 proof-obligations for possible substitutions, of which 26 turned out to be useful substitutions. The last row of the table documents the effort to check all proposed possible implications.

In the following, we document the individual diagrams of the specifications involved in the tautology proof-obligation as well as the complexity results of the sub-component proofs establishing the compositional conclusion.

[5] By brute-force construction of all possible implications for local assumptions with local commitments of the other components and top-level assumptions as premises.

Assumptions for Activity `CROSSING` (Top-level Assumptions)

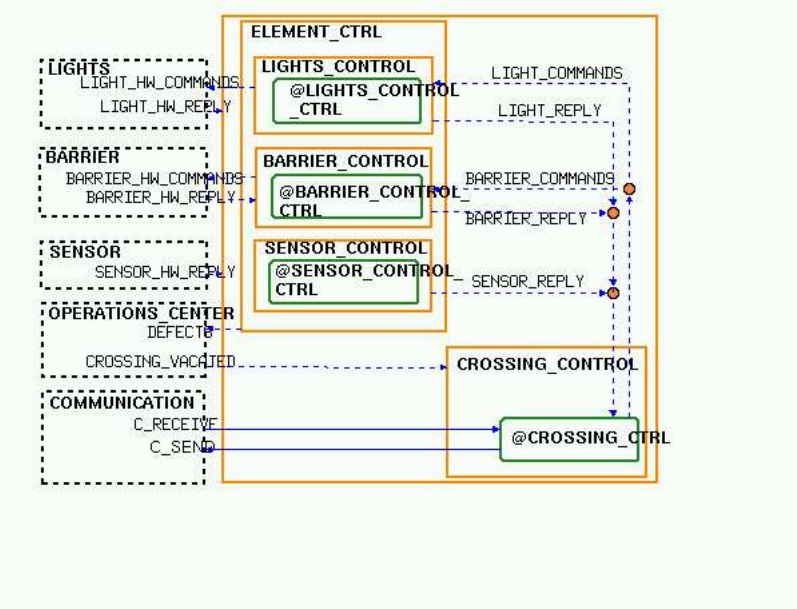


Figure 8.15: Activity CROSSING

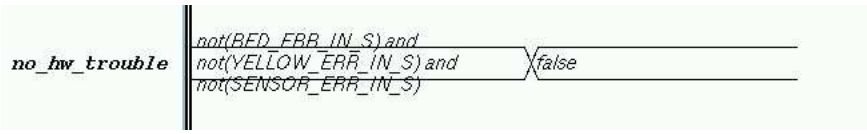


Figure 8.16: `CROSSING` : Assumption `no_hw_trouble`

The error-indicators (free inputs of `SYSTEM`) are assumed to be `false` initially and to remain `false` forever. Assumption `no_hw_trouble` is only satisfied if for all regarded runs of `CROSSING` the initial trigger predicate holds forever. Otherwise, the environment would have to finally satisfy `false`, which is impossible except for the case that the environment is restricted in a way that *no run is possible* at all.

Figure 8.17: `CROSSING` : Assumption `not_passed_or_free`

Assumption `not_passed_or_free` disables the indication of a spontaneously passed train, as well as the indication that `TRAIN` has decided to abort the securing process due to an internal timeout.

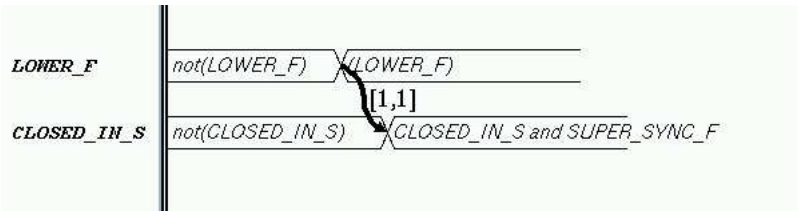


Figure 8.18: CROSSING : Assumption cl_after_cb

The reaction of the physical barriers on LOWER_F is only modeled by a free input of the SYSTEM. Hence, in order to adhere to the time-limits imposed by BARRIER_CONTROL_CTRL for the detection of physical barrier mal-functions, we have to assume that the physical barriers will definitely be closed in the super-step after the LOWER_F command. Otherwise, BARRIER_CONTROL_CTRL would set its BARRIER_ERR_F condition indicating an error of the physical barriers. Notice, that since CLOSED_IN_S is a slow input from outside the system, we have to use a super-step constraint to express the expectation, that regardless of the amount of steps between emitting LOWER_F and observing CLOSED_IN_S, we expect to observe CLOSED_IN_S - simultaneous with SUPER_SYNC_F - in the next stable-state.

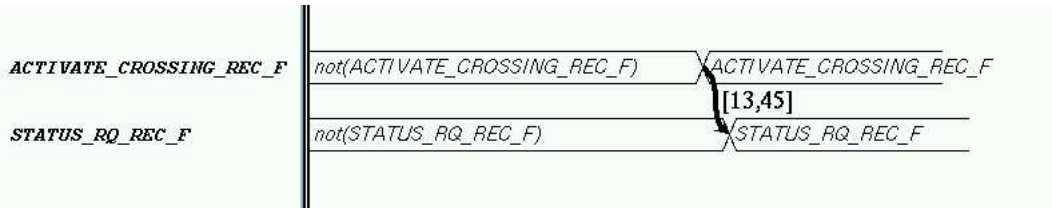


Figure 8.19: CROSSING : Assumption st_rq_xxxxtimes_after_act

Assumption st_rq_xxxxtimes_after_act (Figure 8.19) is the central assumption of the compositional proof. CROSSING will guarantee a CROSSING_SAFE_SND_F reaction one step after STATUS_RQ_REC_F only if - under the listed assumptions - a status request is received at earliest 13 time units and latest 45 time units after receipt of the ACTIVATE_CROSSING_REC_F signal. Under certain circumstances, CROSSING can also send a safe report even if the status request is received earlier, but for these cases a safe report can not be guaranteed (cf. specifications of CROSSING_CTRL and LIGHTS_CONTROL_CTRL).

Commitment `cr_safe_react_on_act` **of Activity** `CROSSING`

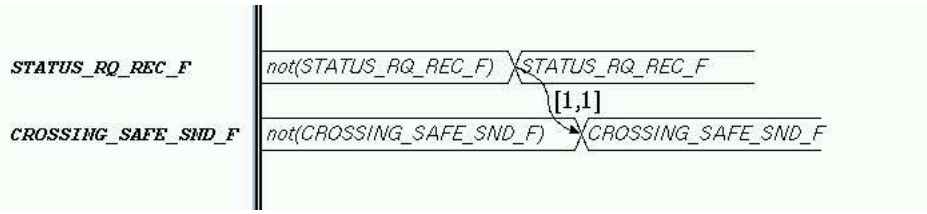


Figure 8.20: `CROSSING` : Commitment `cr_safe_react_on_act`

Under the assumptions listed above, `cr_safe_react_on_act` commits that a status request is answered with a safe-report in the next step.

Component-Proof of `CROSSING`

For the example considered here, it turned out that also component verification is applicable. Recall, that the complexity results for checking the implication of `cr_safe_react_on_act` by the subcomponent specifications to be a tautology and establishing the commitment by a component proof are not comparable, because the complexity of tautology checking only depends on the complexity of the involved specifications. Nonetheless, for reasons of completeness, the following table lists model size as well as the sizes of assumption and commitment observers in terms of input and state bits. The fourth column shows the time needed for performing the component proof for `cr_safe_react_on_act.` BDD-Nodes denotes the size of the transition relation for forward image computation. Image-Comps denotes the number of image computations performed by the VIS model checker for specification verification using invariance checking.

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i/100s	11i/29s	4i/9s	5131sec	16630	231

Specification of Activity CROSSING_CTRL

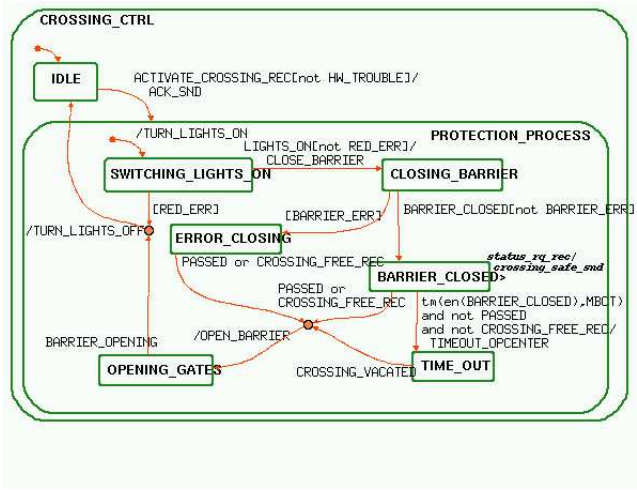


Figure 8.21: Statechart CROSSING_CTRL (with MBCT=40)

Assumptions for Activity CROSSING_CTRL :

Assumption st_rq_xxxxtimes_after_act of CROSSING_CTRL is simply a replication of assumption st_rq_xxxxtimes_after_act of CROSSING (figure 8.19). Since CROSSING_CTRL is the responsible sub-activity of CROSSING for answering a STATUS_RQ_REC_F, the protocols between CROSSING_CTRL and its sibling activities determine the time interval between activation and a status request that has to be assumed for CROSSING in order to guarantee answering a status request with a safe report.

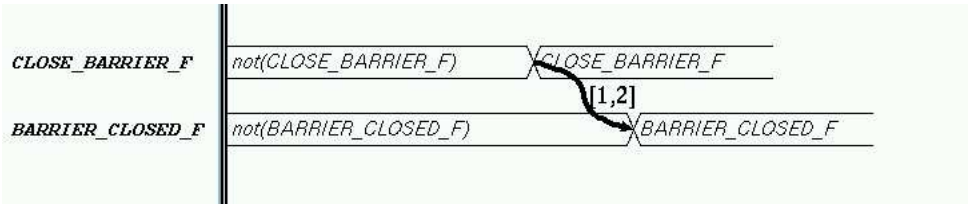


Figure 8.22: CROSSING_CTRL: Assumption barcl_oa_clb

Diagram barcl_oa_clb is an assumption about BARRIER_CONTROL_CTRL as well as about the environment of CROSSING: CROSSING_CTRL emits a CLOSE_BARRIER_F event to BARRIER_CONTROL_CTRL which in the next step reacts on CLOSE_BARRIER_F with a LOWER_F event. For CROSSING we have already assumed that LOWER_F is answered with CLOSED_IN_S by the environment in the next stable state. On CLOSED_IN_S again BARRIER_CONTROL_CTRL reacts with BARRIER_CLOSED_F in the next step. Hence, CROSSING_CTRL will observe a reaction to CLOSE_BARRIER_F within an interval of 1 to 2 time units (super-steps). Notice, that in the time interval [1,2] also step constraints are summarized.

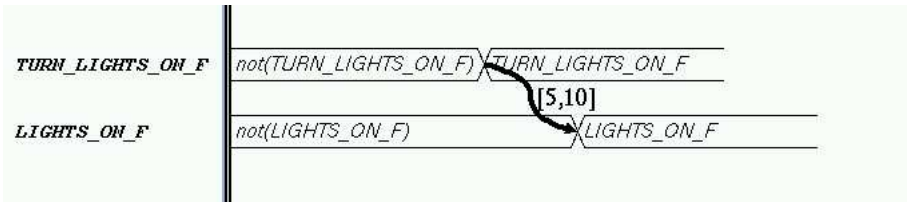


Figure 8.23: CROSSING_CTRL: Assumption lon_oa_tlon_live

Assumption lon_oa_tlon_live refers to the reaction of activity LIGHTS_CONTROL_CTRL to a TURN_LIGHTS_ON_F event. In activity LIGHTS_CONTROL_CTRL, most of the transitions are enabled only after some time triggered counter operations have been performed. The earliest point in time at which LIGHTS_CONTROL_CTRL will react to a TURN_LIGHTS_ON_F with LIGHTS_ON_F is 5 time units thereafter, if no TURN_LIGHTS_OFF_F disturbs the internal computations. The maximal time for emitting a LIGHTS_ON_F is 10 time units after receipt of TURN_LIGHTS_ON_F.

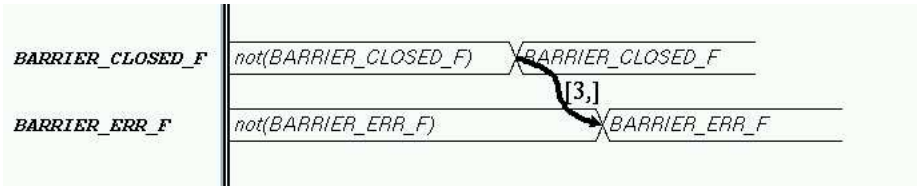


Figure 8.24: CROSSING_CTRL: Assumption no_barr_err

Assumption no_barr_err refers to the condition BARRIER_ERR_F controlled by BARRIER_CONTROL_CTRL. This condition will be set to true by BARRIER_CONTROL_CTRL if either CLOSED_IN_S is not observed within MCT=2 time units after a LOWER_F event or OPENED_IN_S is not observed within MOT=2 time units after a RAISE_F event. By assumption barcl_oa_clb the former case is disabled and only the latter case can lead to BARRIER_ERR_F. Hence, BARRIER_CONTROL_CTRL has to guarantee that BARRIER_ERR_F can only become true due to a failed opening procedure, which can be initiated by a RAISE_F event at earliest 3 time units after BARRIER_CLOSED_F (cf. specification of BARRIER_CONTROL_CTRL).

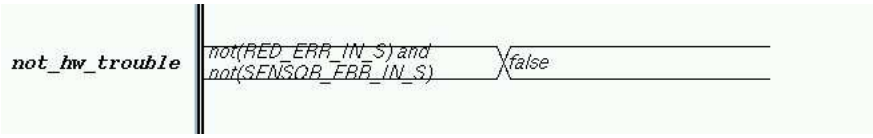


Figure 8.25: CROSSING_CTRL: Assumption not_hw_trouble

Diagram not_hw_trouble expresses the assumption that never a sensor error or red light error will occur.

Assumption not_passed_or_free (without illustration) is a repetition of the respective assumption of CROSSING (figure 8.17). The diagram states that never a spontaneous PASSED_F event from

SENSOR_CONTROL_CTRL will be observed. Furthermore, it is assumed that the (external) train does not indicate abandonment of nearing the crossing by CROSSING_FREE_REC_F.

Commitment `react` of Activity `CROSSING_CTRL`:

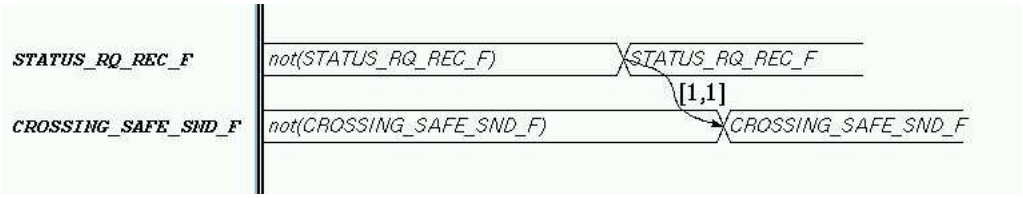


Figure 8.26: CROSSING_CTRL: Commitment react

Provided that all assumptions (cf. figure 8.13) are satisfied, commitment react of CROSSING_CTRL guarantees that a STATUS_RQ_REC_F will be answered with a CROSSING_SAFE_SND_F in the next step. The diagram corresponds to commitment cr_safe_react_on_act of CROSSING (cf. figure 8.20).

Commitment `seq_cmds` of Activity `CROSSING_CTRL`:

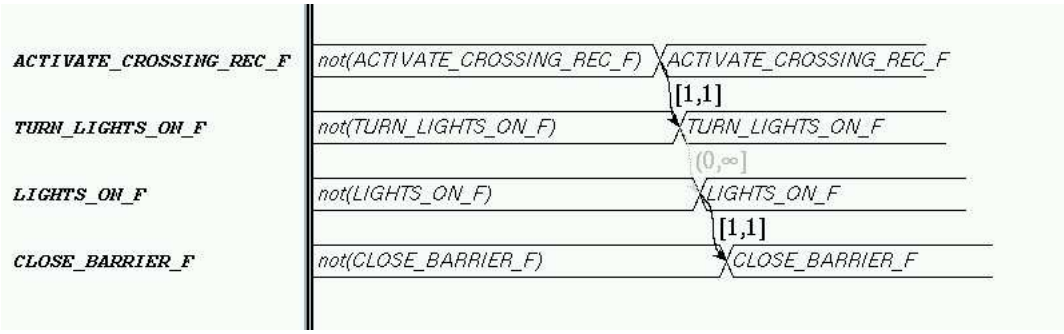


Figure 8.27: CROSSING_CTRL: Commitment seq_cmds

Provided that the assumptions (cf. figure 8.13) are satisfied seq_cmds guarantees reactions on ACTIVATE_CROSSING_REC_F from TRAIN and LIGHTS_ON_F from LIGHTS_CONTROL_CTRL, respectively. In the step after receipt of ACTIVATE_CROSSING_REC_F event TURN_LIGHTS_ON_F will be emitted to activity LIGHTS_CONTROL_CTRL. The reaction of LIGHTS_CONTROL_CTRL on TURN_LIGHTS_ON_F will be the event LIGHTS_ON_F within 5 to 10 time units later. CROSSING_CTRL will react on this event by emitting CLOSE_BARRIER_F to activity BARRIER_CONTROL_CTRL.

Commitment `tloff_true_oa_lon` **of Activity** `CROSSING_CTRL`:

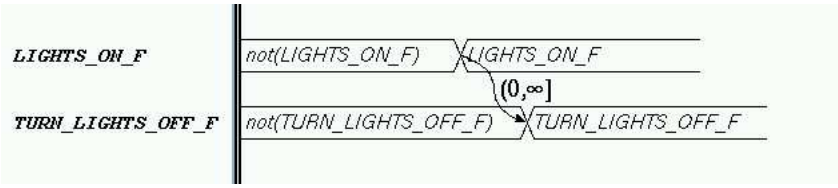


Figure 8.28: `CROSSING_CTRL`: Commitment `tloff_true_oa_lon`

Provided that `LIGHTS_CONTROL_CTRL` will not be interrupted once a `TURN_LIGHTS_ON_F` has been emitted, `tloff_true_oa_lon` guarantees that `TURN_LIGHTS_OFF_F` will be emitted only after `LIGHTS_ON_F` has been observed, provided that the following two assumptions are satisfied.

Figure 8.29: `CROSSING_CTRL`: Assumption `lo_init_false`

`lo_init_false` is a sufficient assumption for `tloff_true_oa_lon`. We prefer to use this simple assumption, even though `lo_init_false` is also covered by assumption `lon_oa_tlon_live` (Fig. 8.23). Using `lo_init_false` instead of `lon_oa_tlon_live` avoids circular dependences among specifications of `LIGHTS_CONTROL_CTRL` and `CROSSING_CTRL`.

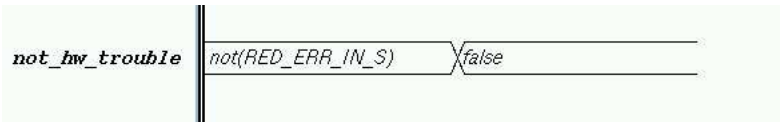


Figure 8.30: `CROSSING_CTRL`: Assumption `no_red_err`

Assumption `no_red_err` (Fig 8.30) is a necessary assumption in order to prove `tloff_true_-oa_lon`. (Fig. 8.25) .

Commitment `init_comm` **of Activity** `CROSSING_CTRL`

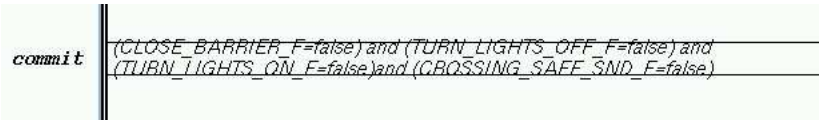


Figure 8.31: `CROSSING_CTRL`: Commitment `init_comm`

For CROSSING_CTRL it is guaranteed with a separate commitment that none of its controlled events is emitted in step 0. This commitment is used to break circular dependences in the tautology proof-obligation.

Component-Proofs of CROSSING_CTRL

The following tables list the results of the component proofs for CROSSING_CTRL.

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
11i/24s	17i/48s	4i/9s	321sec	9777	174

Commitment: react
Assumptions: barcl_oa_clb, lon_oa_tlon_live, no_barr_err, not_hw_trouble, not_passed_or_free, st_rq_rec_xxxxtimes_after_act

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
10i/23s	14i/40s	8i/13s	56sec	13138	110

Commitment: sec_cmds
Assumptions: barcl_oa_clb, lon_oa_tlon_live, no_barr_err, not_hw_trouble, not_passed_or_free

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
10i/22s	2i/8s	4i/7s	3sec	4238	98

Commitment: tloff_true_oa_lon
Assumptions: lo_init_false, no_red_err

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
11i/25s	-	1i/4s	1sec	2304	90

Commitment: init_comm
Assumptions: -

Specification of Activity LIGHTS_CONTROL_CTRL

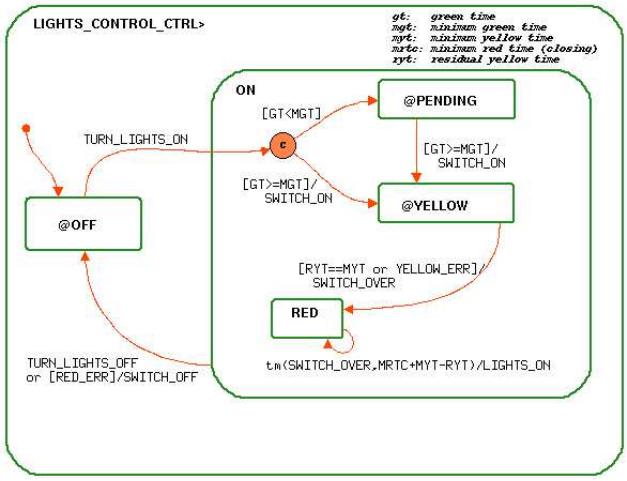


Figure 8.32: Statechart LIGHTS_CONTROL_CTRL (with MGT=4,MYT=2,MRTC=4)

Assumptions for Activity LIGHTS_CONTROL_CTRL

no_rederr (not depicted here, cf. figure 8.30) specifies the assumption that never a red-light error is observed. A red-light error would force an immediate leave of state ON by the higher prioritized transition to state OFF, which will abort the normal sequence of reactions on TURN_LIGHTS_ON_F.

Assumptions no_yelloerr (also without illustration, similar to no_rederr) and no_rederr could be grouped together in one assumption. Even though this would reduce the complexity of the assumption observer a little, we refrain from doing so, since from a logical point of view both assumptions have very different effects. While no_rederr is necessarily required for finally emitting LIGHTS_ON_F, no_yelloerr has only an impact on the continuance in states YELLOW and RED, respectively.

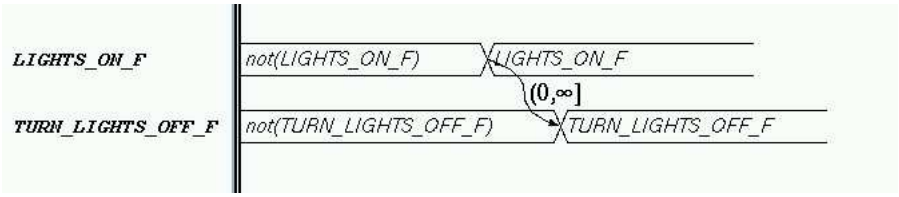


Figure 8.33: LIGHTS_CONTROL_CTRL: Assumption tloff_false_unl_lon

Assumption tloff_false_unl_lon is the dual of commitment tloff_true_oa_lon of activity CROSSING_CTRL. Since state ON of LIGHTS_CONTROL_CTRL will be left immediately if TURN_LIGHTS_OFF_F is received, the assumption is necessarily required in order to finally emit LIGHTS_ON_F.

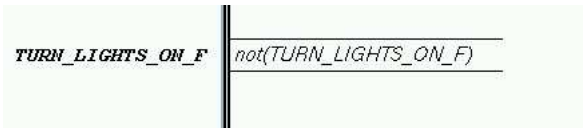


Figure 8.34: LIGHTS_CONTROL_CTRL: Assumption tlon_init_false

TURN_LIGHTS_ON_F has to be false initially in order to match the activation condition of commitment lon_after_tlon.

Commitment lon_after_tlon of Activity LIGHTS_CONTROL_CTRL

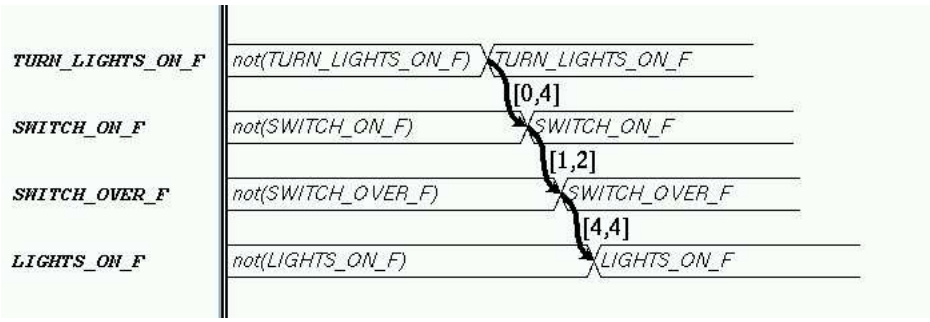


Figure 8.35: LIGHTS_CONTROL_CTRL: Commitment lon_after_tlon

Even though SWITCH_ON_F and SWITCH_OVER_F are not referred to by the specification of CROSSING_CTRL, the overall interval of 5 to 10 time-units between receipt of TURN_LIGHTS_ON_F and LIGHTS_ON_F can best be specified by referring to the sub-intervals of which it is comprised. SWITCH_ON_F and SWITCH_OVER_F are not referred to anywhere in SYSTEM, but have been added to make internals of LIGHTS_CONTROL_CTRL externally observable.

Commitment init_comm_lights of Activity LIGHTS_CONTROL_CTRL

Figure 8.36: LIGHTS_CONTROL_CTRL: Commitment init_comm_lights

Activity LIGHTS_CONTROL_CTRL has to guarantee that the lights are off in step 0. Again this commitment is used to break circular dependences in the tautology proof.

Component-Proofs of LIGHTS_CONTROL_CTRL

The following tables list the results for component verification of LIGHTS_CONTROL_CTRL.

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
4i/44s	7i/19s	9i/17s	13sec	13633	73

Commitment : lon_after_tlon
Assumptions : no_red_err, no_yelloerr, tloff_false_unl_lon, tlon_init_false

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
4i/43s	-	1i/4s	2sec	3670	38

Commitment : init_comm_lights
Assumptions : -

Specification of Activity SENSOR_CONTROL_CTRL

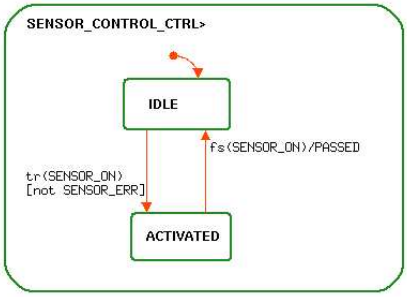


Figure 8.37: Statechart SENSOR_CONTROL_CTRL

Assumption for Activity SENSOR_CONTROL_CTRL

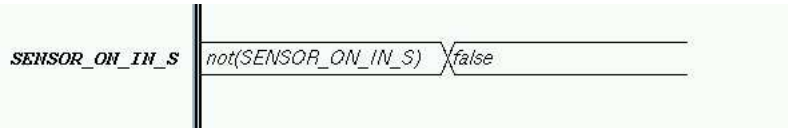


Figure 8.38: SENSOR_CONTROL_CTRL: Assumption never_sens

SENSOR_ON_IN_S is a free input of SYSTEM representing a wayside sensor at the track in front of CROSSING. SENSOR_ON_IN_S indicates a train passing CROSSING, and is meant to play its role in the opening procedure after securing. Since the portion of the protocol considered here is only the securing procedure, we can assume for simplicity SENSOR_ON_IN_S to be false forever .

Commitment never_passed

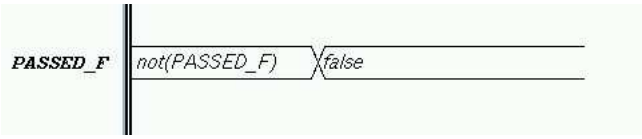


Figure 8.39: SENSOR_CONTROL_CTRL: Commitment never_passed

If SENSOR_CONTROL_CTRL never receives SENSOR_ON_IN_S, never PASSED_F will be emitted.

Component-Proof of SENSOR_CONTROL_CTRL

Even though, the component proof of never_passed is nearly trivial, we list the results for completeness in the table below:

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
2i/10s	1i/4s	1i/4s	0.2sec	419	16

Specification of Activity BARRIER_CONTROL_CTRL

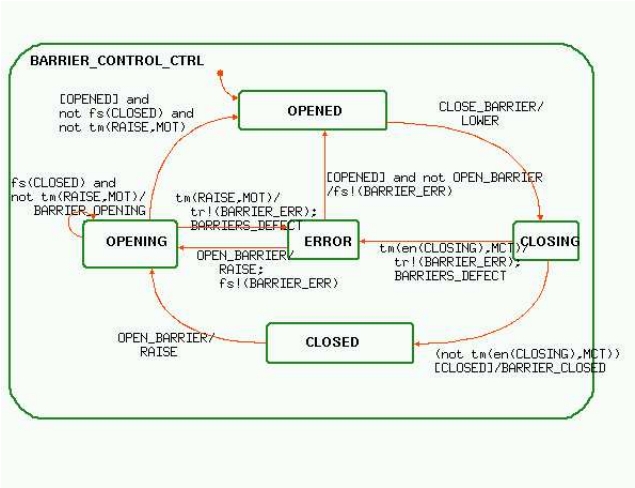


Figure 8.40: Statechart BARRIER_CONTROL_CTRL (with MCT=2,MOT=2)

Assumptions for Activity BARRIER_CONTROL_CTRL

Assumption cl_after_cb (not depicted, cf. figure 8.18) replicates cl_after_cb of CROSSING. By assumption cl_after_cb, a failure of the barrier closing procedure is excluded. Thus, BARRIER_ERR_F can become true only when barrier opening fails.

Figure 8.41: BARRIER_CONTROL_CTRL: Assumption clb_init

CLOSE_BARRIER_F has to be false initially, in order to match the activation condition of the commitment react_on_close_barr which refers to CLOSE_BARRIER_F.

Commitment no_err **of Activity** BARRIER_CONTROL_CTRL

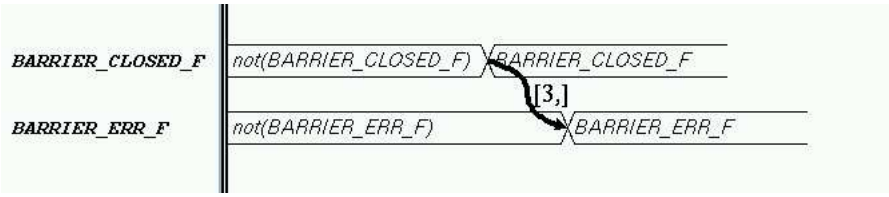


Figure 8.42: BARRIER_CONTROL_CTRL: Commitment no_err

Due to assumption cl_after_cb, barrier closing is successful in any case. Hence, the earliest point in time for BARRIER_ERR_F to possibly become true is 3 time units after emission of BARRIER_CLOSED_F, if opening the barriers fails.

Commitment react_on_close_barr **of Activity** BARRIER_CONTROL_CTRL

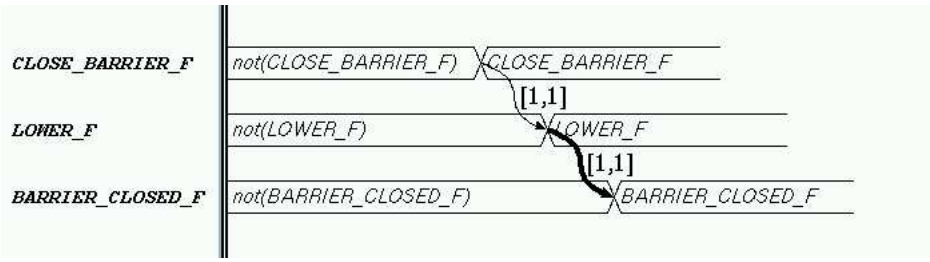


Figure 8.43: BARRIER_CONTROL_CTRL: Commitment react_on_close_barr

Commitment react_on_close_barr specifies the reactions on CLOSE_BARRIER_F. In the subsequent step LOWER_F is emitted. Due to assumption cl_after_cb, the environment reacts on LOWER_F with CLOSED_IN_S in the following super-step, on which again BARRIER_CONTROL_CTRL reacts with BARRIER_CLOSED_F within the same super-step.

Component-Proofs of BARRIER_CONTROL_CTRL

We summarize the results of component verification for BARRIER_CONTROL_CTRL in the following tables:

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
7i/31s	5i/9s	5i/8s	3sec	7154	23

Commitment: no_err
Assumptions : cl_after_cb

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
7i/30s	6i/13s	7i/12s	4sec	8881	22

Commitment: react_on_close_barr
Assumptions : cl_after_cb, clb_init

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
7i/30s	-	1i/4s	1sec	4300	13

Commitment: init_comm_barr
Assumptions : -

Specification of SUPER_SYNC_MONITOR

The commitments for SUPER_SYNC_MONITOR need not to be verified. They rather represent axiomatic invariants of the model representation.

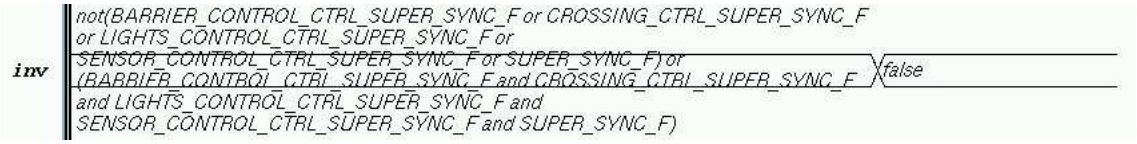


Figure 8.44: Axiom (Commitment) all_the_same of SUPER_SYNC_MONITOR

Since every specification refers to the local SUPER_SYNC_F event of the specified activity, commitment all_the_same relates the local SUPER_SYNC_F events of the sub-activities with the SUPER_SYNC_F event of the top-level activity to which compositional verification is applied. Recall from chapter 5 that stabilization is a property of the entire model. Hence, either all SUPER_SYNC_F events are emitted in the same step, or none of them is emitted. Notice, that the translation of STATEMATE models to SMI/SSL does neither provide a specification of monitors nor a behavioral description. It depends on the concrete specifications involved in the compositional verification task, which axioms regarding a monitor have to be taken into account. A formal characterization of monitors can be found in [DJHP97]. In practice, *it is left to the user to specify monitors in a way that is compliant with the compositional semantics* by providing commitments fitting the needs of the compositional verification task. In case of the compositional proof for cr_safe_react_on_act of activity CROSSING, it is sufficient to guarantee that (1) all SUPER_SYNC_F events of the sub-components

are either emitted simultaneously or none of them is emitted, and (2) that steps and super-steps are at least decoupled as specified by commitment `toggle`:

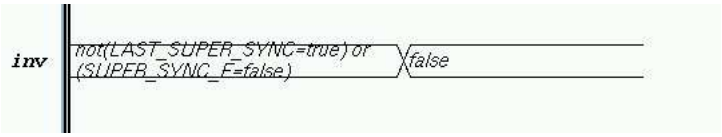


Figure 8.45: Axiom (Commitment) `toggle` of `SUPER_SYNC_MONITOR`

Since stabilization according to chapter 5 (listing 5.2) always takes at least 2 steps, it can be stated that either no `SUPER_SYNC_F` has been emitted in the last step or in the actual step no `SUPER_SYNC_F` is emitted. This decouples step and super-step clocks in a sufficient way for the needs of the presented compositional verification task. Axiom (Commitment) `toggle` makes use of a specification variable. In the specification instantiating diagram toggle a declaration

```
LAST_SUPER_SYNC = last(SUPER_SYNC_F)
```

introduces a new variable with name `LAST_SUPER_SYNC` for the verification task, that always keeps the value of `SUPER_SYNC_F` of the last step.

8.3.3 Compositional Verification of `SYSTEM`

The compositional proof for `SYSTEM` presented in the following proves that train can never pass an unsecured crossing. Therefore, we assume that:

- neither `SENSOR_ON_IN_S` indicates spontaneously a passing train, nor `CROSSING_VACATED_IN_S` interrupts `CROSSING` in its securing procedure.
- `CLOSED_IN_S` indicates closure of the physical barriers only after `LOWER_F` has been sent.
- `CP_REG_INP_IN_S` is always false. `CP_REG_INP_IN_S` determines that `CROSSING` has to be treated as already regarded by `TRAIN`, which would disable the protocol between `TRAIN` and `CROSSING`.
- `RELEASED_MAN_IN_S` is always false, and hence `TRAIN` can never pass a faulty `CROSSING` after manual release by the driver.
- `V_ACTIVATION_POINT_P_IN_S` is initially false. This is required for matching the enabling conditions of the top-level commitment as well as of specifications of `TRAIN`.

Provided that these assumption are satisfied, the compositional proof establishes the guarantee that `TRAIN` can never pass an unsecured `CROSSING`.

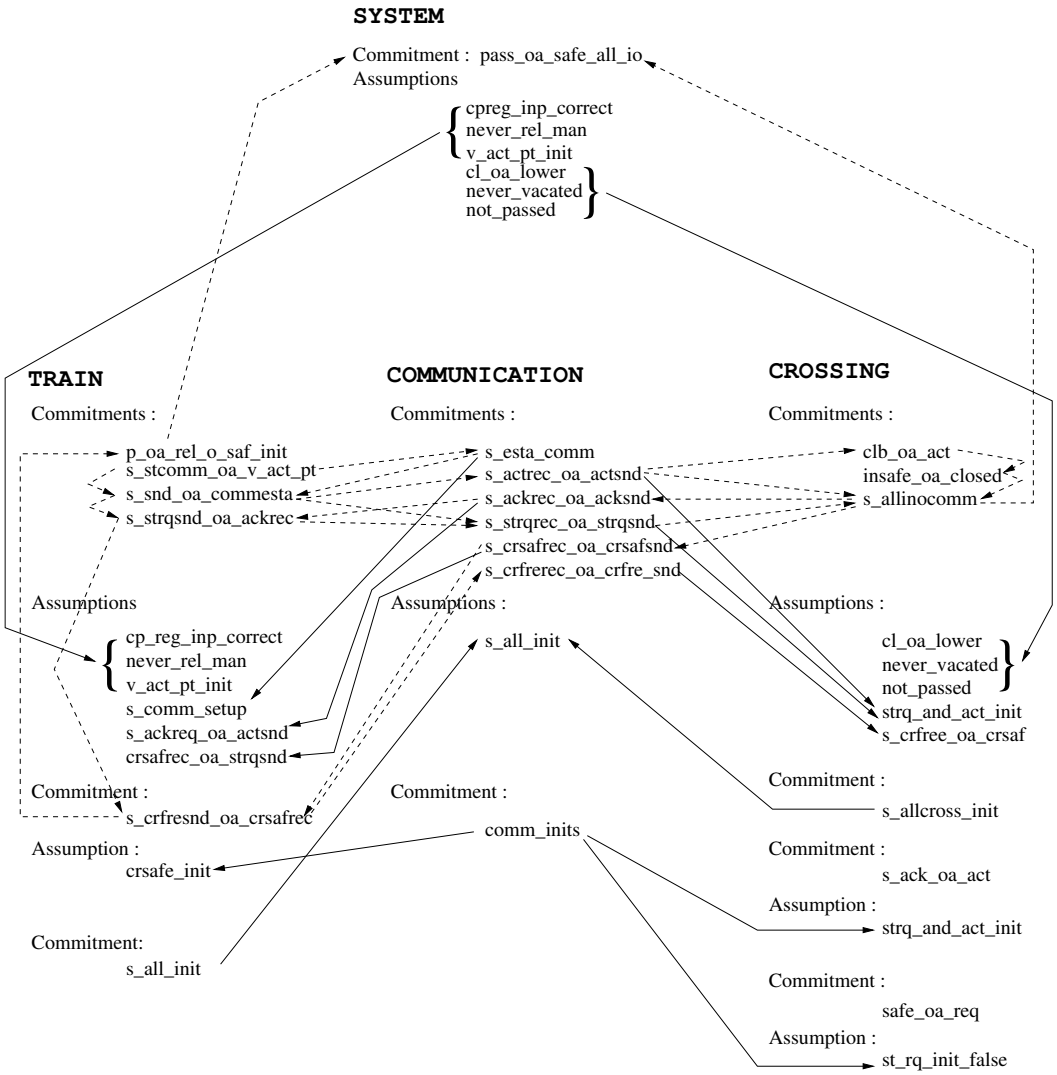


Figure 8.46: SYSTEM: Dependences in Compositional Proof

Figure 8.46 shows the dependences among the specifications in the compositional proof: solid lines depict implications; e.g. assumption `never_rel_man` of `SYSTEM` implies assumption `never_rel_man` of `TRAIN`, whereas assumption `s_all_init` of `COMMUNICATION` is implied by the conjunction of - nearly all - commitments of `TRAIN` as well as commitment `s_allinocomm` of `CROSSING`.

Dashed lines in figure 8.46 depict dependences among commitments imposing an ordering of the events contributing to the protocol. This imposed ordering of events contributing to the protocol is illustrated in figure 8.47 in a pseudo-live sequence chart notation.

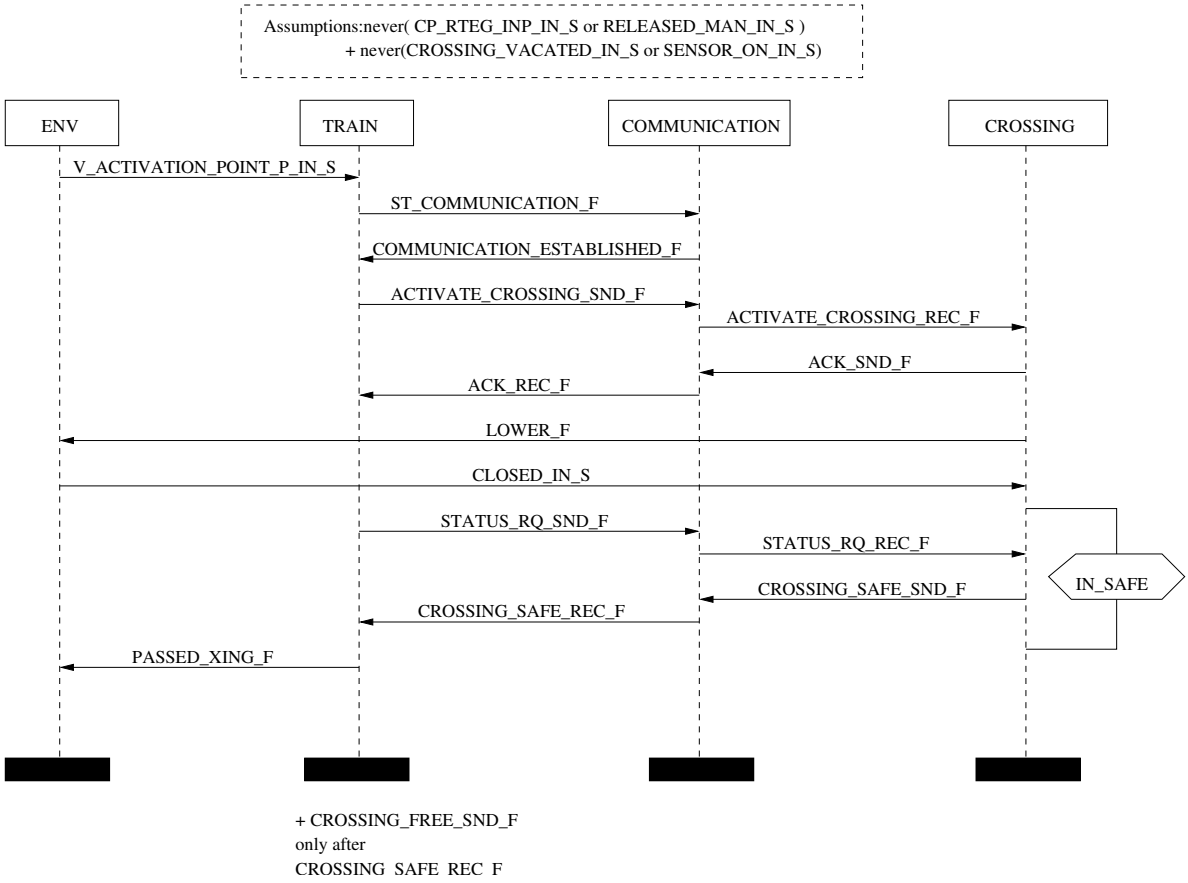


Figure 8.47: Principle Ordering of Events in the verified Protocol

The dashed instance lines in figure 8.47 reflect the fact that neither progress nor particular time-limits are captured by the component specifications contributing to the proof. Occurrences along dashed instance lines have to be interpreted as totally ordered but not necessarily occurring observations. For example, due to the ordering along instance lines, `ST_COMMUNICATION_F` may only be observed after `V_ACTIVATION_POINT_P_IN_S` but it is not required that `ST_COMMUNICATION_F` will be observed at all. This way, the protocol is specified in a pure '*event b may be observed only after event a*' manner, which can possibly get stuck at every point in time without further progress. The respective events are allowed only to occur in the specified order - if they occur at all.

The polygon at instance `CROSSING` represents condition `IN_SAFE` which is true after receipt of `CLOSED_IN_S` and remains true unless a `TIMEOUT_OPCENTER_F` is received. The same as in the example above, sequence chart 8.47 is aimed only at illustration purposes without pretension of being compliant to formal syntax and semantics of live sequence charts.

For the compositional proof presented in the following only an initial activation and securing is considered. Iterative specifications of the entire protocol among `TRAIN`, `COMMUNICATION` and `CROSSING` would require much more effort due to the degrees of freedom of the involved components. In particular, the interlocking of event communication and `TRAIN`'s ability to always abort the entire protocol by emitting an `EMERGENCY_STOP_F` as well as the chosen representation of meaningful sensors by free inputs aggravates specification of the entire protocol. E.g. specification of correct closing of the physical barriers involves the events `CLOSE_BARRIER_F`, `LOWER_F` , free inputs

CLOSED_IN_S and OPENED_IN_S which have to be assumed to react in a tight timing scheme as well as the events BARRIER_ERR_F and BARRIER_CLOSED_F. Furthermore, the behavior of free input SENSOR_ON_IN_S representing a passing TRAIN has to be restricted by assumptions accordingly. Also, missing observability of operation modes at system level makes specification difficult. E.g. COMMUNICATION is entirely encapsulated in the protocol interlocking, neither the state of the communication link - being established or disabled - nor any of the transmitted events is observable at system level. Besides inputs, only the outputs

- error indications BARRIERS_DEFECT_F, SENSOR_DEFECT_F, YELLOW_DEFECT_F, RED_DEFECT_F
- barrier commands LOWER_F and RAISE_F,
- light control indicators SWITCH_ON_F and SWITCH_OVER_F,
- a TIMEOUT_OPCENTER_F indication
- TRAIN's track-position T_F
- and EMERGENCY_STOP_F, IN_SAFE_F, PASSED_XING_F

are observables of the system.

Furthermore, the δ-delays of fast communication impose weaker specifications than desirable. Reaction to events is not instantaneous: e.g., even though a ST_COMMUNICATION_F may abort the communication link, a pending event as projected in the step before is nonetheless issued to the receiver. The same way, ACTIVATE_CROSSING_CTRL can in the same step be stopped by EMERGENCY_STOP_F (respectively STPPED_F) but nonetheless send a STATUS_RQ_SND_F. In case of EMERGENCY_STOP_F, the communication-link is turned off only after a manual release which is again represented by a free input.

Even though high degrees of freedom and lack of observability are obstacles in particular for compositional verification, it can be proved that for all possible initial scenarios the mandatory ordering of events in the system only allows TRAIN to pass CROSSING after CROSSING has reported itself safe.

Remark 8.2 *Nearly all diagrams involved in the compositional proof are of the forms*

- *initially Q only after P*
- *initially P*
- *forever P*

which would strongly suggest the use of three template diagrams with appropriate parameter mapping. Unfortunately, STDx-specifications are restricted to refer to only one commitment declaration, and hence only one parameter mapping can be specified for instances of the template diagrams. Instantiating each commitment template within an individual specification would lead to considerable complexity overhead due to redundant assumption observers. It should hence be an issue for future extensions of STDx, to allow reference to more than one commitment declaration in specifications. ■

Compositional Verification of Commitment `pass_oa_safe_all_io` of `SYSTEM`

Model-Size for parallel composition of all observers: 94i/174s
Number of involved diagrams: 41 (19 Sub-component commitments)

Variant	# Implications	time Implications	time Proof	BDD-Nodes	Image-Comps
subst.	15	164sec	3680sec	5864/18694	8/15
naive	-	-	TO!24h	-	-
proposed implications	375 (17 True)	5144sec	-	10893	15

Like the compositional proof for `CROSSING`, also the compositional verification task for `SYSTEM` has been executed in two variants, a naive one and an optimized variant. In the naive variant all sub-component specifications are referred to as they are used for component verification without any modification. In the optimized variant all those assumptions have been substituted, which are entirely covered by top-level assumptions or commitment of other components, according to dependences depicted in figure 8.46 by solid lines. The verification task for the naive variant suffered from the complexity of the parallel composition of all observers. Even though, the substitution of local assumptions by local commitments and top-level assumptions has only been applied to the invariant to be checked, the Cone of Influence Reduction applied by the VIS model checker for the simplified proof-obligation succeeded. The top-level commitment could be shown to be implied by the sub-component specifications in about one hour[6]. Checking validity of the substitutions could be performed in less than 3 minutes.

Also checking all 375 proof-obligations offered by the proof-obligation generator as possible substitutions proposal took less than 2 hours.

Assumptions for Activity `SYSTEM`

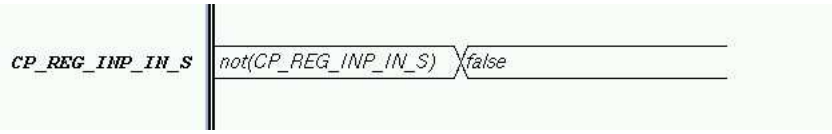


Figure 8.48: `SYSTEM`: Assumption `cpreg_inp_correct`

It has to be assumed that `CP_REG_INP_IN_S` is always false, since this input would cause `TRAIN` to treat `CROSSING` as already regarded which would disable the activation and securing protocol.

[6] cf. pattern proof P1 on SYSTEM of section 8.2: There the size of the un-abstracted model has been 31i/264s. P1 could only be established by applying propositional abstraction.

Figure 8.49: SYSTEM: Assumption never_rel_man

In case of an EMERGENCY_STOP_F, RELEASED_MAN_IN_S allows TRAIN to pass a faulty CROSSING, which has not reported itself safe (cf. figure 8.15). In order to focus on the activation and securing protocol, manual release of a faulty CROSSING has to be disabled.

Figure 8.50: SYSTEM: Assumption v_act_pt_init

V_ACTIVATION_POINT_IN_S causes TRAIN to start communication and to activate CROSSING. In order to match the activation conditions of the respective commitments V_ACTIVATION_POINT_P_IN_S is assumed to be false initially.

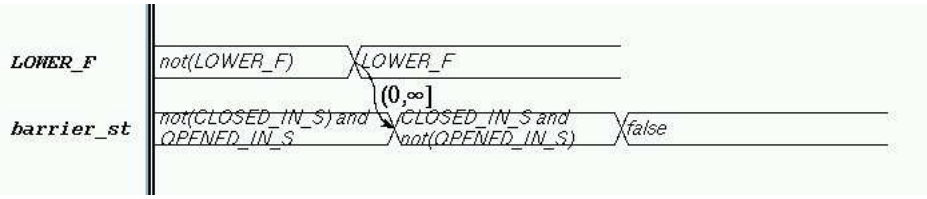


Figure 8.51: SYSTEM: Assumption cl_oa_lower

Only an initial securing is considered in the presented compositional proof, without subsequent re-opening of the barriers Hence, it can be assumed that only after a LOWER_F event a sensor may indicate closed physical barriers. For simplicity, it is assumed that once the barriers are closed, they remain closed forever. Alternatively, it could be assumed that the barriers remain closed unless RAISE_F is observed. This would enforce specification of the entire opening procedure after TRAIN has passed CROSSING and hence would complicate significantly.

Figure 8.52: SYSTEM: Assumption never_vacated

CROSSING_VACATED_IN_S triggers CROSSING to proceed the barrier opening procedure after TIME-

OUT_OPCENTER_F (in particular CROSSING_CTRL, cf figure 8.21). In order to focus on normal reaction of CROSSING on a STATUS_RQ_REC_F from TRAIN, it is assumed that CROSSING_VACATED_IN_S is never observed.

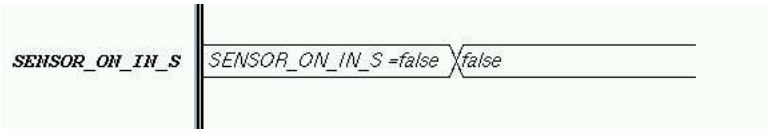


Figure 8.53: SYSTEM: Assumption not_passed

SENSOR_ON_IN_S represents a TRAIN passing a wayside sensor in front of CROSSING. Assuming SENSOR_ON_IN_S to be always false, disables spontaneous perception of a passing TRAIN and disabling the securing process.

Commitment pass_oa_safe_all_io of Activity SYSTEM

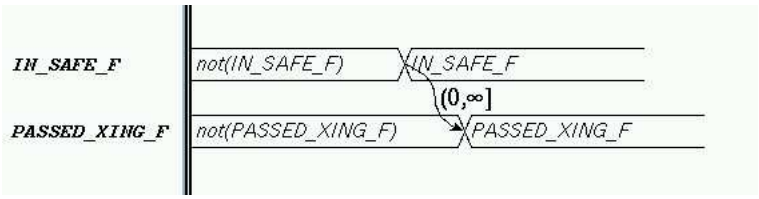


Figure 8.54: SYSTEM: Commitment pass_oa_safe_all_io

Commitment pass_oa_safe_all_io is the formalization of the overall safety property, that TRAIN can only pass CROSSING after CROSSING has been entirely secured. IN_SAFE_F is a condition set by statechart CROSSING_CTRL of CROSSING, whereas PASSED_XING_F is an event emitted by statechart ACTIVATE_CROSSING_CTRL of TRAIN.

Specification of Activity TRAIN

Figures 8.5 and 8.6 show activity TRAIN and statecharts ACTIVATE_CROSSING_CTRL and its sub-chart WF_CROSSING_SAFE, respectively. Activity TRAIN is too complex for the application of component verification without abstraction. Since we are only interested in TRAIN's contribution to the communication protocol with CROSSING via COMMUNICATION, we abstract from the activities SPEED_CONTROL and ODOMETER by applying propositional abstraction to the data-items SPEED_CONTROL_CTRL:NOMINAL_SPEED and TRAIN:ODATA. Recall from section 3.3 that SPEED_CONTROL controls a braking curve of TRAIN and emits a STPPED_F event dependent on SPEED_CONTROL_CTRL:NOMINAL_SPEED and TRAIN:ODATA if TRAIN has reached a critical distance from a not-yet secured CROSSING. By propositional abstraction (strong) of SPEED_CONTROL_CTRL:NOMINAL_SPEED and TRAIN:ODATA we abstract from the concrete computations that may potentially cause STPPED_F. The price to be paid is that STPPEP_F can now non-deterministically be emitted in the abstract model.

Assumptions for Activity TRAIN

Assumptions cp_reg_inp_correct, never_rel_man and v_act_pt_init are replications of the respective assumptions at system-level (cf. figure 8.47; the individual diagrams are depicted in figures 8.48,8.49, and 8.50, respectively).

Figure 8.55: TRAIN: Assumption crsafrec_oa_strqsnd

Diagram crsafrec_oa_strqsnd formalizes the assumption that CROSSING_SAFE_REC_F can be received only after a STATUS_RQ_SND_F has been emitted. This is an assumption regarding COMMUNICATION as well as regarding CROSSING.

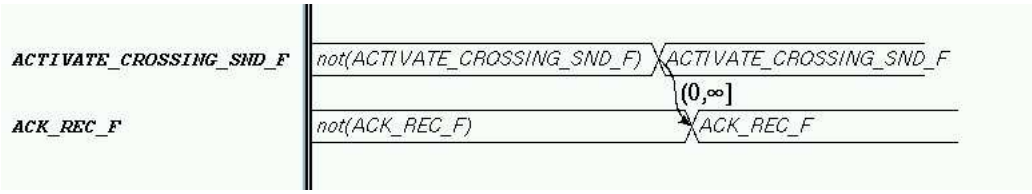


Figure 8.56: TRAIN: Assumption s_ackrec_oa_actsnd

Also s_ackrec_oa_actsnd is an assumption regarding COMMUNICATION as well as regarding CROSSING. ACK_REC_F from CROSSING is expected to be observed only after ACTIVATE_CROSSING_SND_F has been sent to CROSSING via the communication link.

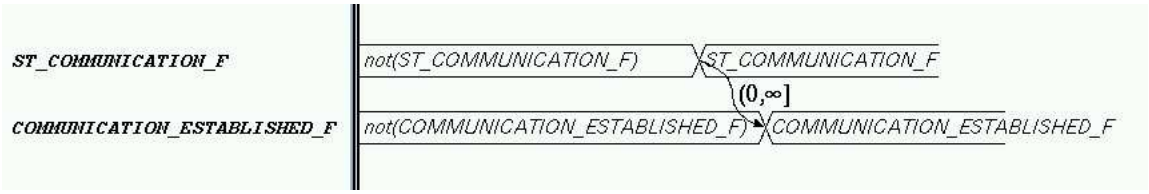


Figure 8.57: TRAIN: Assumption s_comm_setup

Before any communication with CROSSING can be performed, COMMUNICATION has to be started. It is assumed by s_comm_setup that COMMUNICATION emits COMMUNICATION_ESTABLISHED_F only after ST_COMMUNICATION_F has been sent to COMMUNICATION.

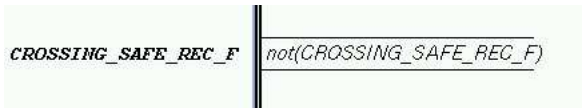


Figure 8.58: TRAIN: Assumption crsafe_init

It is assumed in a separate assumption, that CROSSING_SAFE_REC_F is initially not set.

Commitments for Activity TRAIN

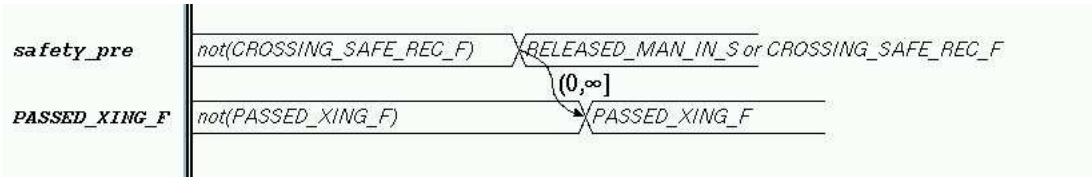


Figure 8.59: TRAIN: Commitment p_oa_rel_o_saf_init

p_oa_rel_o_saf_init is one of the basic component commitments of the compositional proof. TRAIN has to guarantee that CROSSING can only be passed if either (a faulty) CROSSING has been released manually (this case is excluded by assumption never_rel_man) or after CROSSING has reported itself safe.

Figure 8.60: TRAIN: Commitment s_crfresnd_oa_crsafrec

It is committed that CROSSING_FREE_SND_F is not issued spontaneously - aborting CROSSING's securing procedure - but only after a CROSSING_SAFE_REC_F has been received from CROSSING. CROSSING_FREE_SND_F could also be sent after a manual release, which is disabled due to assumption never_rel_man.

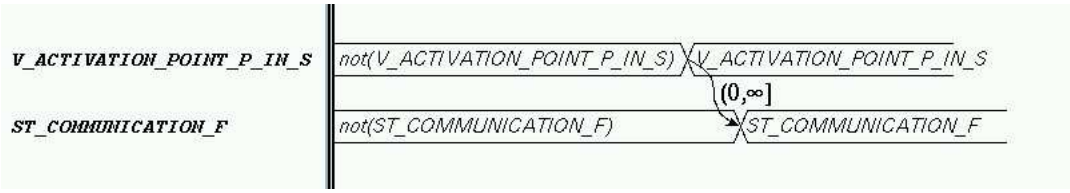


Figure 8.61: TRAIN: Commitment s_stcomm_oa_v_act_pt

The first action of TRAIN after V_ACTIVATION_POINT_P_IN_S has become true has to be the initialization of a communication link. The diagram commits that ST_COMMUNICATION_F is emitted only after V_ACTIVATION_POINT_P_IN_S.

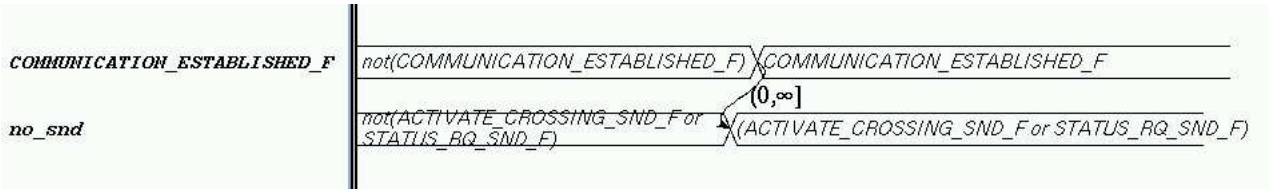


Figure 8.62: TRAIN: Commitment s_snd_oa_commesta

s_snd_oa_commesta guarantees that TRAIN will not send an activation or status-request to CROSSING before COMMUNICATION indicates an established communication link.

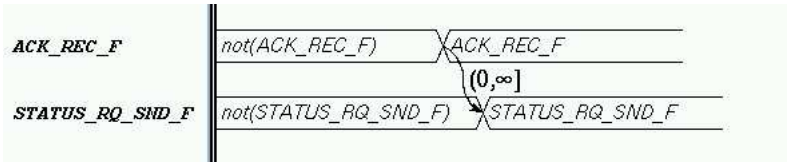


Figure 8.63: TRAIN: Commitment s_strqs_nd_oa_ackrec

TRAIN will send a status request only after receipt of an acknowledge for activation from CROSSING.

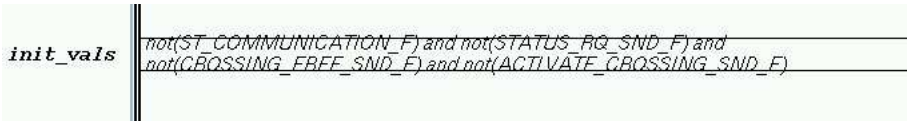


Figure 8.64: TRAIN: Commitment s_all_init

s_all_init commits, that all relevant events are initially not emitted.

The interlocking with the commitments of COMMUNICATION and CROSSING, imposes an ordering of the event receipts and emissions by TRAIN. The only possible order of received and emitted events conforming to the assumption and commitments of TRAIN's specification is compliant with the messages received and sent by instance line TRAIN in figure 8.47.

Component-Proofs of TRAIN (using propositional abstraction of SPEED_CONTROL_CTRL:NOMINAL_SPEED and TRAIN:ODATA)

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i/39s	15i/33s	16i/27s	118sec	8588	57

Commitments: p_oa_rel_saf_o_saf_init, s_stcomm_oa_v_act_pt, s_snd_oa_commesta, s_strqsnd_oa_ackrec

Assumptions: `cp_reg_inp_correct, never_rel_man, v_act_pt_init, s_comm_setup, s_ackreq_oa_actsnd, crsafrex_oa_strqsnd`

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
12i/35s	1i/4s	4i/7s	4sec	6099	56

Commitment: `s_crfesnd_oa_crsafrec`
Assumption: `crsafe_init`

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
12i/38s	-	1i/4s	3sec	5900	30

Commitment: `s_all_init`
Assumption: -

Specification of Activity `COMMUNICATION`

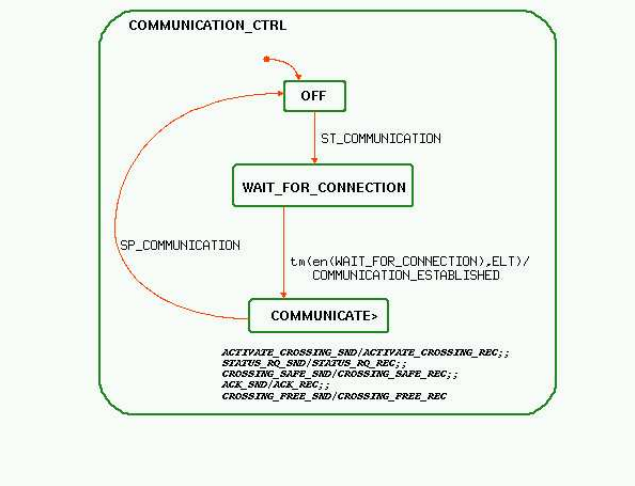


Figure 8.65: Statechart `COMMUNICATION_CTRL(ELT=1)`

Assumption for Activity `COMMUNICATION`

Figure 8.66: `COMMUNICATION`: Assumption `s_all_init`

The only assumption for activity `COMMUNICATION` regards the initial values of the input events from `TRAIN` and `CROSSING`. All events are expected to be initially absent.

Commitments for Activity `COMMUNICATION`

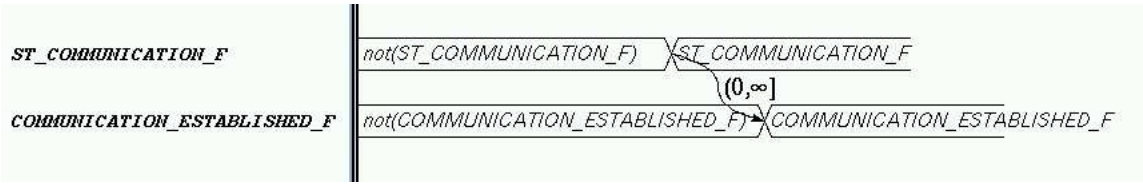


Figure 8.67: `COMMUNICATION`: Commitment `s_esta_comm`

`COMMUNICATION` has to guarantee that `COMMUNICATION_ESTABLISHED_F` will not spontaneously be emitted to `TRAIN`, but only after `TRAIN` has sent `ST_COMMUNICATION_F`.

For each of the events which are transmitted by `COMMUNICATION` a commitment guarantees that the respective receive-event is emitted only after receipt of its corresponding send-event.

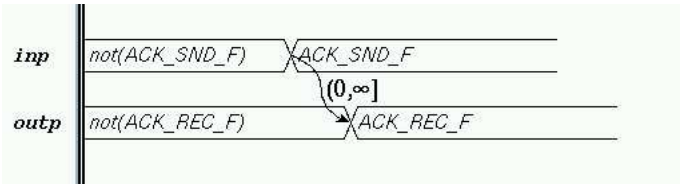


Figure 8.68: `COMMUNICATION`: Commitment `s_ackrec_oa_acksnd`

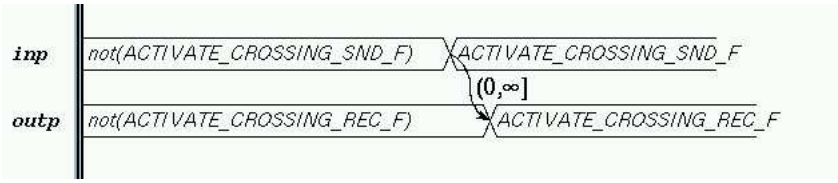


Figure 8.69: `COMMUNICATION`: Commitment `s_actrec_oa_actsnd`

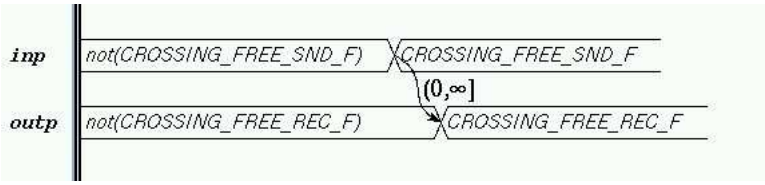


Figure 8.70: `COMMUNICATION`: Commitment `s_crfrerec_oa_crfre_snd`

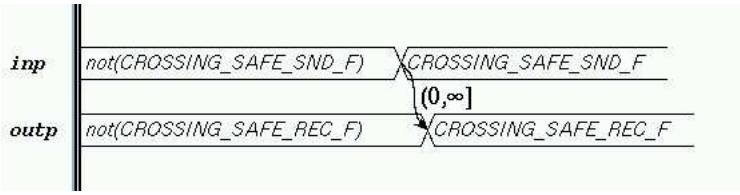


Figure 8.71: COMMUNICATION: Commitment s_crsafrec_oa_crsafsnd

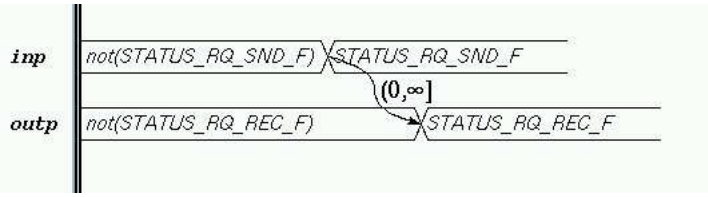


Figure 8.72: COMMUNICATION: Commitment s_strqrec_oa_strqsnd

Figure 8.73: COMMUNICATION: Commitment comm_inits

In a special commitment, it is committed that activity COMMUNICATION emits no event in step 0.

Component-Proofs of COMMUNICATION

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
7i/16s	1i/4s	24i/37s	3sec	3459	12

Commitments: s_esta_comm, s_actrec_oa_actsnd, s_ackrec_oa_acksnd, s_strqrec_oa_strqsnd, s_crsafrec_oa_crsafsnd, s_crfrerec_oa_crfre_snd
Assumptions: s_all_init

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
7i/15s	-	1i/4s	1sec	482	7

Commitments: comm_inits
Assumptions: -

Specification of Activity CROSSING

For figures of CROSSING and its sub-charts, the reader is referred to figures 8.15, 8.21, 8.32, 8.37, and 8.40.

Assumptions for Activity CROSSING

Assumptions cl_oa_lower, never_vacated and not_passed are replications of the respective assumptions at system-level (according to figure 8.46; the individual diagrams are depicted in figures 8.51, 8.52, and 8.53, respectively).

Figure 8.74: CROSSING: Assumption s_crfree_oa_crsaf

CROSSING_FREE_REC_F means abortion of CROSSING's securing procedure by TRAIN due to a time-out of TRAIN's internal timer. Even though, CROSSING_FREE_REC_F is not entirely excluded from the considered protocol, by disabling RELEASED_MAN_IN_S for TRAIN (assumption never_rel_man, cf. figure 8.6 illustrating ACTIVATE_CROSSING_CTRL), CROSSING_FREE_REC_F (translation of TRAIN's CROSSING_FREE_SND_F) can only be received after emitting CROSSING_SAFE_SND_F. Notice, that assumption s_crfree_oa_crsaf involves reactions of TRAIN as well as of COMMUNICATION.

Figure 8.75: CROSSING: Assumption strq_and_act_init

Assumption strq_and_act_init formalizes the expectation that events STATUS_RQ_REC_F and ACTIVATE_CROSSING_REC_F are initially absent.

Commitments for Activity CROSSING

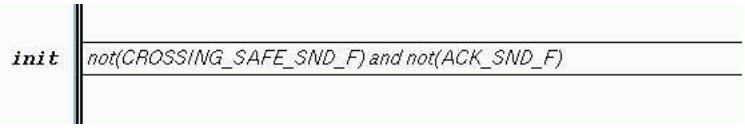


Figure 8.76: CROSSING: Commitment s_all_cross_init

Activity CROSSING has to guarantee that CROSSING_SAFE_SND_F and ACK_SND_F are not emitted in step 0.

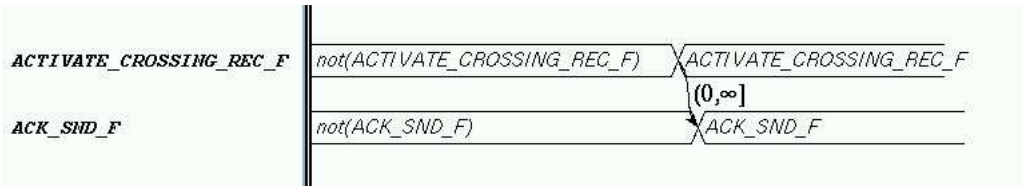


Figure 8.77: CROSSING: Commitment s_ack_oa_act

An ACK_SND_F event can only be emitted after receipt of ACTIVATE_CROSSING_REC_F.

Figure 8.78: CROSSING: Commitment safe_oa_req

CROSSING has to guarantee that CROSSING_SAFE_SND_F will be emitted only after STATUS_-RQ_REC_F.

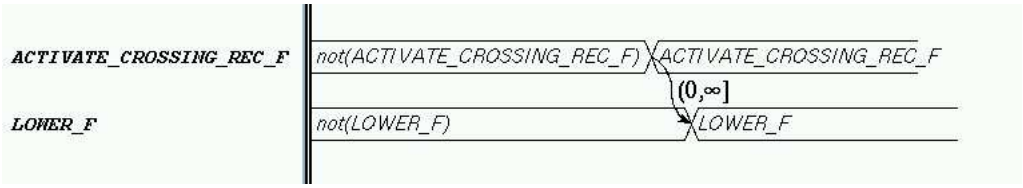


Figure 8.79: CROSSING: Commitment clb_oa_act

clb_oa_act commits that LOWER_F will be sent to the physical barriers only after activation by TRAIN.

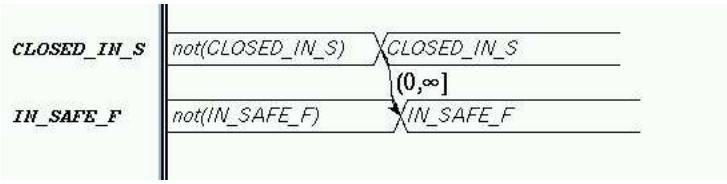


Figure 8.80: CROSSING: Commitment insafe_oa_closed

CROSSING will only enter its state BARRIER_CLOSED and on entering set IN_SAFE_F to true after the physical barriers are sensed to be closed.

CROSSING's central commitment s_allinocomm - committing reaction on a status-request from TRAIN - is a bit too large to be presented as screen-shot. For reasons of limited space, figure 8.81

illustrates a variant of the diagram, for which parameters have been used and the mapping of the parameters to concrete expressions is given below the figure. Since the compositional proof is aimed at establishing an overall safety property no hardware error is disabled by any assumption. Light defects, barrier defects, as well as sensor defects and also a timeout from the operation center can arbitrarily interfere with CROSSING's securing process. Due to this high degree of freedom imposed by possible errors and free inputs modeling the controlled physical components the formalization of commitment s_allinocomm is a bit tricky:

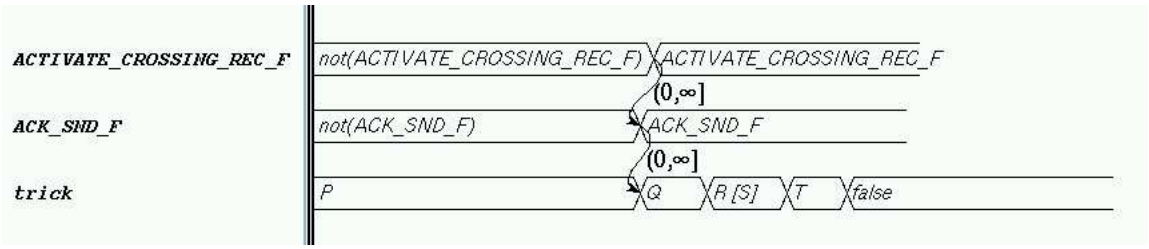


Figure 8.81: CROSSING: Commitment s_allinocomm

The parameters in diagram s_allinocomm are mapped to expressions, according to the mapping:
P ⇒not(CROSSING_SAFE_SND_F or IN_SAFE_F) and not(TIMEOUT_OPCENTER_F)
Q ⇒not(CROSSING_SAFE_SND_F) and IN_SAFE_F and not(TIMEOUT_OPCENTER_F)
R ⇒(not(CROSSING_SAFE_SND_F) and not(IN_SAFE_F) and TIMEOUT_OPCENTER_F)
S ⇒(LAST_STRQ and CROSSING_SAFE_SND_F and IN_SAFE_F)
T ⇒(not(CROSSING_SAFE_SND_F) and not(IN_SAFE_F) and not(TIMEOUT_OPCENTER_F))
CROSSING only answers STATUS_RQ_REC_F from TRAIN by CROSSING_SAFE_SND_F, if CROSSING is entirely secured (IN_SAFE_F=true) and if not yet a timeout has occured (which happens MBCT=40 time-units after entering BARRIER_CLOSED). Only after having sent ACK_SND_F to TRAIN, CROSSING can enter state BARRIER_CLOSED (cf. figure 8.21). On entering this state, condition IN_SAFE_F is set to true indicating a secured status. If neither PASSED_F nor CROSSING_FREE_REC_F is received within MBCT=40 time-units after entering BARRIER_CLOSED a timeout occurs, the state is left and TIMEOUT_OPCENTER_F is emitted, IN_SAFE_F is set to false and remains false unless CROSSING_VACATED_IN_S is received from the environment. Because assumption never_vacated disables CROSSING_VACATED_IN_S, IN_SAFE_F will remain false forever after a timeout. If no STATUS_RQ_REC_F has been received before the timeout, CROSSING_SAFE_SND_F will never be sent[7]. Only if after 'not(CROSSING_SAFE_SND_F) and IN_SAFE_F and not(TIMEOUT_OPCENTER_F)' (Parameter Q) not yet a timeout has occurred and STATUS_RQ_REC_F *has been received in the last step,* the exit condition mapped to parameter S requires sending of CROSSING_SAFE_SND_F in order to fulfill commitment s_allinocomm by prematurely exiting the diagram. If '¬Q and ¬R and S' holds after Q,

[7]Due to assumptions not_passed never a PASSED_F event will be received by CROSSING. Note, that PASSED_F is controlled by SENSOR_CONTROL_CTRL and depends on the free input SENSOR_ON_IN_S. Hence, if it were not disabled by assumptions, indication of a train passing the crossing would occur spontaneously and without correspondence to the perception of TRAIN. In order to only allow meaningful occurences of PASSED_F, assumptions could be formulated, restricting input SENSOR_ON_IN_S in such a way, that PASSED_F can only occur after TRAIN has emitted a PASSED_XING_F. For the compositional proof presented here entirely disabling SENSOR_ON_IN_S is appropriate, because the behavior of CROSSING is regarded only unless a CROSSING_SAFE_SND_F is emitted

`s_allinocomm` is prematurely exited in a satisfied status. Notice, that `LAST_STRQ` in the expression for exit condition `S` refers to a last-variable declaration of the specification:

```
LAST_STRQ = last(STATUS_RQ_REC_F);
```

Component-Proofs of `CROSSING`

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i//104s	11i/26s	15i/24s	2125sec	17109	275

Commitments: clb_oa_act, insafe_oa_closed, s_allinocomm
Assumptions: cl_oa_lower, never_vacated, not_passed, strq_and_act_init, s_crfree_oa_crsaf

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i/100s	-	1i/4s	250sec	15846	132

Commitments: s_allcross_init
Assumptions: -

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i/99s	1i/4s	4i/7s	130sec	19176	132

Commitments: s_ack_oa_act
Assumptions: strq_and_act_init

Model	Assumption	Commitment	Time	BDD-Nodes	Image-Comps
13i/99s	1i/4s	4i/7s	522sec	12208	134

Commitments: safe_oa_req
Assumptions: st_rq_init_false

8.3.4 Summary of the Application of STDx Verification

The application examples demonstrate the necessity of combining step and super-step constraints in STDx-specifications. The proof for crossing illustrates compositional verification of real-time requirements using compositional verification, while the proof for system establishes a safety critical system-requirement.

By the real-time proofs of `ACTIVATE_CROSSING_CTRL` and `CROSSING` it could be shown that the constant capturing the crossing closing time (`CCT`) does not always allow crossing to complete its securing procedure, before train emits a status request. `CCT` determines the time to wait by train between receipt of acknowledge and sending a status request. Therefore, the constant should capture the maximal crossing closing time instead of only a mean crossing closing time, as it is the case in the example.

For both presented compositional proofs manual interaction has been necessary in order to reduce the complexity of the respective verification task. The proof-obligation generator supports the user with a proposal of possible implications, which can automatically be verified to determine a set of implications can be applied for assumption substitution. Unfortunately, selection of valid

implications for assumption substitution is not yet guided by the graphical user interface and should be supported by a more comfortable selection mechanism in the future.

Complexity of compositional verification only indirectly depends on modeling style and modularity of a STATEMATE design. The less inter-dependences between the considered activities of a design exist, the less specifications have to be regarded in a compositional proof. Since complexity of a compositional proof depends on the specifications involved in the implication of the top-level commitment, compositional verification benefits from a clear localization of functionality and a loosely coupled communication structure. Even though no scheduling of controlled activities has to be considered in case of the case-study, there are some aspects of the design that aggravate compositional verification:

- Nearly all communication between the activities of the case-study is realized using events, which are visible only one step. Very few indications of the respective local status of the activities are observable at system-level. E.g. communication only indicates an established connection by sending an event to train, the status of communication is not indicated at system-level. The lack of status information available in the interfaces of the activities and the momentariness of event communication aggravates specification of useful activation conditions for iterative STDx-specifications.
  Due to the synchronous parallel composition of the Statemate semantics and the event-based communication, the timing of the protocol is very strict. It seems rather unrealistic to us for modeling such an interaction of such autonomous components that train, communication and crossing communicate step-synchronously with each other instead of using at least a communication using conditions, which would store their values for several steps and thus fit better to a protocol between such independent sub-systems, by allowing a more loosely coupling of triggers and reactions.

- The behavior of the system involves a high degree of freedom. Important parts of the system are modeled using free inputs of the design, such as for example the controlled physical barriers. Assumption have to suppose correct response to lower and raise commands within strict time limits. Also light and sensor errors have to be excluded from consideration by assumptions in order to focus on the normal operation of crossing.
  The components of the design do not share their perception. Train determines passing the crossing according to a position which is not known by crossing, while the perception of crossing of a passing train is modeled using a free input which has no correspondence to the actual position or speed of the train. Hence, the crossing can perceive a train passing, even though the train has been stopped due to an emergency stop.

- The activities of the design interact very closely. All communication between train and crossing takes place via activity communication, which is activated and deactivated by train. Train and crossing both react on events delivered by communication with emission of events to communication. All these triggers and reactions are hidden from the system-level view-point and can hence not be referred to in the specification of the system. The securing protocol involves circular dependence of the behaviors of crossing and train.
  Also the sub-activities of crossing communicate in a very close cooperation, as can be seen in the dependence graph (figure 8.13) for the compositional proof of crossing. In particular communication of `CROSSING_CTRL` with its sibling activities involves mutual triggering of transitions by events.

- The behavior of train depends on the computation of speed and position. Thus, control depends on complex arithmetic computations of data. In order to verify the contribution of train to the securing protocol, propositional abstraction from these computations has to be applied, because otherwise train is too complex for verification. This way, being stopped by an emergency stop appears to be a non-deterministic event after application of propositional abstraction. On the other hand, the interaction of sub-activities of train is too close and data-dependent to apply compositional verification.

Compositional verification using STDx-specifications is in principle independent from model complexity and thus scales up also to very large systems. For example, if the computation of the breaking curve in `TRAIN` was replaced by a quite more complicated and concrete function, the verification complexity for the compositional proof for `SYSTEM` would not be affected. Moreover, compositional verification provides a valuable enhancement of the repertoire of assessment techniques for quality assurance in a model-development process. Both presented compositional proofs demonstrate that compositional verification provides a deep insight into activity interaction and dependences among activities in a design. Only if this interaction is entirely captured by specifications for the involved activities, fulfillment of a top-level requirement specification can successfully be concluded from local specifications. This way, compositional verification provides also a measure for the completeness of requirement specifications. All conditions for correct behavior of the sub-activities of a design have to be explicitely considered and formalized. The allowed and required activity interaction has to be explicitely specified in order to establish a system requirement using compositional verification.

9 Conclusion and Outlook

In this work, we have presented the STATEMATE Verification Environment (STVE). The STVE aims at supporting a model based development process of safety critical embedded systems with formal verification techniques. Overall product quality is critically dependent on the familiarity of system and software designers with the established process, and any change, in particular the introduction of a technology completely novel to designers, can potentially cause significant process degradation. It is thus essential to tune the handling to use-cases well understood and easily appreciated by designers. Robustness analyses and formal debugging offered by the STVE are powerful, easy to handle and push-button techniques which do not confront the developer with the underlying mathematical verification technology. Application of robustness analyses already during development increases the overall product quality by revealing conflicts in the model early in the design of a reference model. Formal debugging supports the developer in examining the behavior of the model. Witness paths found by robustness analyses and formal debugging are translated into simulation control programs, which can be used to drive the STATEMATE simulator. We have presented the realization and the integration of analyses and formal debugging with the STVE and demonstrated their application for the case-study of a radio based signaling system.

For verification of basic safety requirements, specification patterns are offered which can easily be instantiated by providing a mapping of the formal parameters of the pattern to user defined expressions. Creation of proof-obligations and execution of proof tasks is integrated with a graphical user interface that hides away all control aspects of verification from the developer. All verification related activities including application of several abstraction techniques can be initiated directly from appropriate icons. Verification using observer-pattern has been demonstrated for the case-study.

Verification of functional and safety properties of the design under consideration is supported using the intuitive graphical specification formalism of Symbolic Timing Diagrams (STDx). We have presented and explained the constructs and features of STDx and enhanced the formalism to be capable of real-time specifications in terms of the super-step semantics of STATEMATE models. Specifications can quantitatively refer to steps as well as to the virtual time of super-step execution. As common semantical basis of observer pattern and STDx verification we have presented a class of discrete-timed symbolic automata and their representation by observer modules for application of verification. A sub-class of these automata has been identified for which the acceptance criterion can be captured by a invariant. For this relevant sub-class of timed automata it has been shown that efficient verification using invariance checking is applicable.

We have defined a formal semantics of STDx-diagrams in a constructive way by unwinding into timed symbolic automata. Only few restrictions have to be adhered to, in order to translate STDx-diagrams into observer modules with invariant acceptance condition, s.t. invariance checking is applicable to the verification of STDX-specifications. Application of verification of STDx-specifications has been demonstrated for components of the case study with a set of component proofs.

Compositional verification of STDx-specifications as supported by the STVE has been formally described and has been illustrated by two compositional proofs regarding the case-study. One of the compositional proofs demonstrates verification of a real-time specification for a sub-component of the case-study, while the other one establishes a safety critical requirement for the entire system.

Part of the STVE is a proof-manager that offers proof-obligation construction and proof task execution by simple graphical operations. The management of proof-results as provided by the proof-manager has been formally defined. The proof-manager keeps track of the verification results and provides an overview of the proof-state of the design at every time. It has been explained how proof-results are automatically invalidated according to changes in the design or modifications of the requirement specifications. Proofs can be established again w.r.t. the changes by re-executing the affected proof-tasks.

Although compositional reasoning and hierarchical conclusions are offered as a fully automated technique, compositional verification remains a difficult task that requires significantly more expert knowledge than the pure push-button techniques offered by the STVE. Decomposition of a system requirement specification into requirement specifications of its sub-components can not be automated and thus remains a creative task.

Because complexity of compositional verification only depends on the number and complexity of the involved specifications but not on the complexity of the model, compositional verification can in principle be applied to arbitrarily large models. It only depends on the modularity of the system, how many and how complex specifications are required in order to establish a top-level requirement by compositional verification. Close interaction of the sub-components of a design may involve circular dependencies between specifications and lead to reasonable complexity of the compositional proof task. As a solution for the complexity issues of compositional verification substituting local assumptions by local commitments and top-level assumptions, respectively, has been integrated with the proof-obligation generation. Assumption substitution has been applied to both compositional proofs presented in this work.

Besides being a technique for complexity reduction for specification verification, compositional verification also provides a deep insight into activity interaction of a system. Only if the contributions and conditions of all considered sub-components are explicitely specified, requirements regarding their interaction can be verified using compositional verification. This involves intensive examination and requires unambiguous specifications of the behavior of sub-components, as observable at their interfaces. Hence, compositional verification using STDx-specifications is a valuable technique for quality assurance in a model-based development process.

9.1 Outlook

Currently no automatic techniques support the complexity reduction for the stabilization bound analysis integrated with the STVE. Since neither RCOI reduction nor propositional abstraction are applicable because both abstraction techniques do not preserve the stabilization properties of the design, future work should identify abstraction techniques applicable for this use-case. We feel that the presented hand-abstraction of action parts of time triggered transitions by slow inputs is a promising direction for future work.

Even though the formalism STDx has been proven to be successful applicable for specification verification of embedded systems, there are still some directions for future improvements. In section 7.4 we have pointed out that the limitation of referring only one commitment declaration in STDx-

specification limits the use of parameterized diagram templates. By abolishing this limitation re-use of diagrams could be improved, and capturing requirement specifications could be made easier by offering specific diagram template libraries. In order to improve use of diagram templates not only the annotations of symbolic events should be parameterizable, but also the time-interval annotation of constraints in the templates, which is currently not supported. We expect that offering libraries of predefined diagram templates of which event annotations and constraint intervals can be mapped to user-defined expressions and values, respectively, will ease formalization of requirements and contribute to the intuitiveness of the formalism.

Regarding component verification using STDx, integration with abstraction techniques should be improved. Currently STDx verification is integrated only with manual selection of variables for propositional abstraction. The promising results of automatic abstraction refinement [Bie03] for pattern verification strongly suggest application also in the context of verification using STDx. Besides targeting maximal complexity reductions for verification, integration of STDx verification with automatic abstraction refinement could significantly increase user-friendliness.

Concerning the support of compositional verification using STDx specifications, future work should improve the application and handling of assumption substitution. Although assumption substitution is already integrated with the proof-obligation generation, a list of substitutions has to be provided, which is not yet guided by the graphical user-interface.

An interesting approach to improve the applicanility of compositional verification could be the support of *mixed* proof-obligations, where parts of the composition are represented by specifications and other parts are represented by sub-component behavior representations. This could permit compositional derivation of specifications, from a parallel composition of the less complex components of a design with observers replacing complex components by their specification. Even though we can not forecast the verification complexity of such an approach, we expect that the effort spent for specification verification for all involved sub-component can be reduced significantly.

Another direction of further work concerns the integration of alternative and complementary specification formalisms. Live Sequence Charts (LSC) [DH99, Klo03] are already integrated with the proof-manager in a prototypical way. Combining the graphical formalisms STDx and LSC is interesting in more than one way. First, compositional derivation of LSC-specifications from STDx-specifications of a decomposed view and derivation of *black-box* STDx-specifications of the top-level from *grey-box* LSC specifications extend the richness of the available repertoire of offered specification formalisms and verification techniques. Second, LSC specifications could be decomposed into sub-component STDx-specifications and this way be used to define compositional proof-obligations. Since LSCs are used to specify the interaction of components, while STDx is used to specify the interface behavior of activities, both specification formalisms are in a way complementary. It seems a promising approach, to automatically decompose protocol specifications given by LSCs into sub-component specifications for the involved components. STDx-diagrams could be derived from the instances of LSCs and be offered to the user, this way utilizing LSCs for the definition of compositional proof-obligations.

Bibliography

[ABC+99] A. Allara, M. Bombana, S. Comai, B. Josko, R. Schlör, and D. Sciuto. Specification of embedded monitors for property checking. In *2nd Forum on Design Languages, FDL'99, Lyon*, pages 117–126, 1999.

[AD91] R. Alur and D. Dill. The Theory of Timed Automata. In de Bakker, Henzinger, and de Roever, editors, *Real Time : Theory in Practice, Proceedings of Rex 1991*, number 600 in LNCS, pages 45–73, 1991.

[AdAG+01] R. Alur, L. de Alfaro, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Kirsch, and B. Wang. Mocha: A model checking tool that exploits design structure, 2001.

[AEKN00] Nina Amla, E. Allen Emerson, Robert P. Kurshan, and Kedar S. Namjoshi. Model checking synchronous timing diagrams. In *Formal Methods in Computer-Aided Design*, pages 283–298, 2000.

[AEN99] Nina Amla, E. Allen Emerson, and Kedar S. Namjoshi. Efficient decompositional model checking for regular timing diagrams. In *Conference on Correct Hardware Design and Verification Methods*, pages 67–81, 1999.

[AFF+02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AHM+98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[Alu98] R. Alur. Timed Automata, In NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems, 1998.

[BB91] Gerard Berry and Albert Beneviste. The synchronous approach to reactive and real-time systems. In *Another Look at Real Time Programming, Proceedings of the IEEE,79*, pages 1270–1282, 1991.

[BBD+99] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlör, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking. In Felix Redmill and Tom Anderson, editors, *Towards System Safety – Proceedings of the Seventh Safety-critical Systems Symposium, Huntingdon, UK*, pages 150–173. Safety-Critical Systems Club, Springer Verlag, 1999.

[BBEH99] Jürgen Bohn, Udo Brockmeyer, Claus Essmann, and Hardi Hungar. SMI – system modelling interface, draft version 0.1. Technical report, Kuratorium OFFIS, e.V., Oldenburg, 1999.

[BBEW98] Jürgen Bohn, Udo Brockmeyer, Claus Essmann, and Gunnar Wittich. SMI – system modelling interface, technical report. Technical report, Universität Oldenburg, 1998.

[BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS 99*, LNCS. Springer, 1999.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[BDW00] Tom Bienmüller, Werner Damm, and Hartmut Wittke. The STATEMATE Verification Environment – Making it real. In E. Allen Emerson and A. Prasad Sistla, editors, *12th international Conference on Computer Aided Verification, CAV*, number 1855 in LNCS, pages 561–567. Springer Verlag, 2000.

[Bec99] Peter Bechberger. Model-Based Software Development for Electronic Control Units (ECUs). *ATZ/MTZ*, Special Issue 'Automotive Electronics':2–7, 1999.

[Bie03] Tom Bienmüller. *Reducing Complexity for the Verification of Statemate Designs*. PhD thesis, Carl von Ossietzky Universität/OFFIS Oldenburg, 2003.

[Bit00] F. Bitsch. Classification of safety requirements for formal verification of software models of industrial automation systems. In *13th Conference on Software & Systems Engineering and their Applications - ICSSEA 2000, Paris*, 2000.

[Bit01] F. Bitsch. Safety-patterns - the key to formal specification of safety-requirements. In *SAFECOMP 2001 - Computer Safety, Reliability and Security, 20th International Conference*. Springer-Verlag, 2001.

[BPR98] Juergen Broede, Hugo Pfoertner, and Klaus Richter. The Importance of Testing for Successful Life Usage Monitoring Systems. In *19th International Symposium on Aircraft Integrated Monitoring Systems (AIMS98), Garmisch-Partenkirchen, Germany*, 1998.

[Bro99] Udo Brockmeyer. *Verifikation von* STATEMATE *Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, December 1999.

[Bry92] Randal E. Bryant. Symbolic boolean Manipulation with ordered Binary-Decision Diagrams. *ACM Comp. Surveys*, ??(24):293–318, 1992.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CES83] Edmund M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Procceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[CGH97] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, February 1997.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press,Cambridge, Massachusets, London, England, 1999.

[CLM89] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362. IEEE Press, 1989.

[CM81] K. Mani Chandy and Jayadev Misra. Proofs of networks of processes. In *IEEE Transaction on Software Engineering*, volume 7(4), pages 417–426, 1981.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DAC98a] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, AUTHORS: Matthew B. Dwyer (1), George S. Avrunin (2) and James C. Corbett (3) AFFILIATIONS: Department of Computing and Information Sciences (1) Kansas State University Department of Mathematics and Statistics (2) University of Massachusetts Department of Information and Computer Science (3) University of Hawai'i, , 1998.

[DAC98b] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[DC01] Werner Damm and Moshe Cohen. Formal checker verifies software, June 2001.

[DD97] H. Dierks and C. Dietz. Graphical Specification and Reasoning: Case Study Generalized Railroad Crossing. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME'97*, volume 1313 of *LNCS*, pages 20–39. Springer, 1997.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[Die96] Cheryl Dietz. Graphical formalization of real-time requirements. In *FTRTFT*, pages 366–384, 1996.

[DJHP97] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In *COMPOS, Copositionality: The Significant Difference, International Symposium COMPOS'97, edt. H.Langmaack, A. Pnueli and W.-P. de Roever, LNCS 1536, Springer-Verlag*, pages 186–238, 1997.

[DJS95] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods*, pages 331–410. Oxford Univ. Press, 1995.

[DK01] Werner Damm and Jochen Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, 19(2):121–141, 2001.

[DL02] Henning Dierks and Marc Lettrari. Constructing test automata from graphical real-time requirements. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 433–454, London, UK, 2002. Springer-Verlag.

[DS93] W. Damm and R. Schlör. Specification and verification of system-level hardware designs using timing diagrams. In *The European Conference on Design Automation, Paris, France*, pages 518–524. IEEE Computer Society Press, 1993.

[EMSS90] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proc. 2nd International Computer Aided Verification Conference*, pages 136–145, 1990.

[EOKX95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[ESt97] Bundesministerium des Inneren EStdIT. V-Model, Development Standard for IT-Systems of the federal Republic of Germany, 1997.

[fEA01] ITEA Information Technology for European Advance. Guideline for Validation and Verification Real-Time Embedded Software Systems - Software Development Process for Real-Time Embedded Systems (dess), Dec 2001.

[fES97] European Committee for Electronical Standardization. Railway Applications: Software for Railway Control and Protection Systems, EN 50128, 1997.

[Fey96] Konrad Feyerabend. Real time Symbolic Timing Diagrams - Technical Report, 1996.

[Fis96] Kathryn Fisler. *A Unified Approach to Hardware Verification through heterogeneous Logic of Design Diagrams.* PhD thesis, Department of Computer Science, Indiana University, August 1996.

[Fis00] K. Fisler. On tableau constructions for timing diagrams, 2000. In NASA Langley Workshop on Formal Methods.

[FJ97] Konrad Feyerabend and Bernhard Josko. A visual formalism for real time requirement specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97, Lecture Notes in Computer Science 1231*, pages 156–168, 1997.

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[Gre02] Bertrand Gregoire. Automata oriented program verification. Master's thesis, Facultes Universitaires Notre-Dame de la Paix, Namur, Belgium, September 2002.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403 – 414, 1990.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions of Software Engineering Methods*, 5(4):1–36, Oct 1996.

[Hol00] Leszek Holenderski. Compositional verification of synchronous networks. In *FTRTFT*, pages 214–227, 2000.

[Hol05] H.J. Holberg. Erfahrungsbericht über formale methoden in den bereichen model checking und automatic test vector generation im industriellen umfeld. Technical report, Virtuelles Engineering Kompetenzzentrum (VISEK), To Appear 2005.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, Inc., New York, NY, USA, 1998.

[iL00] i Logix. Statemate MAGNUM, 2000.

[iL04] i Logix. www.ilogix.com, 2004.

[JEKD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.

[Jon91] Capers Jones. *Applied software measurement: assuring productivity and quality.* McGraw-Hill, Inc., 1991.

[Jos87] B. Josko. MCTL: An extension of CTL for modular verification of concurrent systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 165–187. Springer, 1987.

[Jos93] Bernhard Josko. Modular Specification and Verification of Reactive Systems. Carl von Ossietzky Universität Oldenburg, 1993. Habiltationsschrift.

[Klo03] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behaviour*. PhD thesis, Department für Informatik, C.v.O. Universität Oldenburg, 2003.

[KT00] Jochen Klose and Andreas Thums. The Statemate Reference Model of the Reference Case Study 'Verkehrsleittechnik'. Technical report, University of Augsburg, 2000.

[KV98] Orna Kupferman and Moshe Y. Vardi. Modular model checking. *Lecture Notes in Computer Science*, 1536:381–401, 1998.

[KV00] Kupferman and Vardi. An automata-theoretic approach to modular model checking. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 22, 2000.

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[KW01] Jochen Klose and Hartmut Wittke. An Automata Based Representation of Live Sequence Charts. In *Proceedings of TACAS 2001,number 2031 in LNCS*. Springer Verlag, 2001.

[LIS02] Andrew Lapping, Mark Irving, and Andy Stringer. De-Mystifying Signalling Priciples Through Modelling and Simulation., 2002.

[LSU95] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1995.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM99] K. L. McMilan. Circular compositional reasoning about liveness. In *Conference on Correct Hardware Design and Verification Methods*, pages 342–345, 1999.

[MJH+98] In-Ho Moon, Jae-Young Jang, Gary D. Hachtel, Fabio Somenzi, Jun Yuan, and Carl Pixley. Approximate reachability don't cares for CTL model checking. In *ICCAD*, pages 351–358, 1998.

[OS19] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, SRI, Menlo Park, CA, Menlo Park, CA, 19.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, sub-series F: Computer and System Science, pages 123–144. Springer-Verlag, 1985.

[PS97] A. Pnueli and E. Singermann. Fair synchronous transition systems and their lifeness proofs. Technical report, Dept. of Comp. Science, Weizmann Institute, 1997.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th International Symposium on Programming*, pages 195–220, 1981.

[R.E86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[RTC92] RTCA. RTCA/DO-178B, Software Consideration in Airborne Systems and Equipment Certification, RTCA-Requirements and Technical Concepts for Aviation, 1992.

[SAC97] SACRES. Syntax of the System Specification Language within SACRES. In *Deliverable Report I1.4.A, (Esprit 20897 SACRES)*, 1997.

[SAC99] R. Schlör, A. Allara, and S. Comai. System verification using user-friendly interfaces. In *Design, Automation and Test in Europe / User Forum*, pages 167–172. IEEE Computer Society Press, 1999.

[Sch00] Rainer Schlör. *Symbolic Timing Diagrams: A Visual Formalism for Model Verification*. PhD thesis, Carl von Ossietzky Universität/OFFIS Oldenburg, 2000.

[Som98] Fabio Somenzi. CU Decision Diagram Package, 1998. CUDD is available from `http://vlsi.Colorado.EDU/~fabio`.

[SRS+03] Bernhard Schätz, Jan Romberg, Martin Strecker, Oscar Slotosch, and Katharina Spies. Modeling embedded software: State of the art and beyond. In *Proceedings of ICSSEA 2003, 16th International Conference on Software and Systems Engineering and their Applications*, 2003.

[Tho90] Wolfgang Thomas. Automata On Infinite Objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–191. Elsevier Science Publishers, 1990.

[Tho02] Filip Thoen. Enabling Early Software Development Through Virtual System Prototyping. *EDN System Design Series*, A Special Advertising Section to EDN(2):32–36, 2002.

[vHBK98] Reinhard v. Hanxleden, Ali Botorabi, and Slawomir Kupczyk. A Co-Design Approach for Safety-Critical Automotive Applications. *IEEE Micro Special Issue on Embedded Fault-Tolerant Systems*, 18(5):66–79, Sep/Oct 1998.

[VIS96a] The VIS Group, VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996. VIS is available from the VIS home-page: `http://www-cad.eecs.Berkeley.EDU/~vis`.

[VIS96b] The VIS Group, VIS: A System for Verification and Synthesis. In *FMCAD*, 1996.

[Wit99] Gunnar Wittich. *Ein problemorientierter Ansatz zum Nachweis von Realzeiteigenschaften eingebetteter Systeme*. PhD thesis, Carl von Ossietzky Universität Oldenburg, August 1999.

Curriculum Vitae

19.8.1965 Geboren in Hannover als Sohn von Erika (geb. Westphal) und Dieter Wittke.

1970 Einschulung in die Grundschule Trenknerweg in Hamburg.

1974 Übergang in das Gymnasium Hohenzollernring in Hamburg.

7.6.1984 Erlangung der Allgemeinen Hochschulreife.

1.2.1985-7.1987 Ausbildung zum Maschinenschlosser im Hamburger Berufsbildungszentrum HBZ e.V. in Hamburg.
Bestehen der Gesellenprüfung am 12.6.1987.

1.9.1987-31.12.1988 Zivildienst als Rettungshelfer beim Rettungsdienst Friesland.

1.10.1989-7.1997 Studium der Informatik an der C.v.O.-Universität in Oldenburg. Ab WS 94/95 Tätigkeit als wissenschaftliche Hilfskraft in den Projekten KORSYS und FORMAT.

10.7.1997 Abschluss des Studiums mit der Diplomarbeit "Eine graphische Design-Umgebung für STD-Spezifikationen".

1.7.1997 Einstellung bei OFFIS e.V. im Bereich "Sicherheitskritische Systeme" (SC) als wissenschaftlicher Mitarbeiter.
Seitdem Mitarbeit in den Projekten VFORMAT, SACRES, SAFEAIR und Autogen. In diesen Projekten zuständig für graphische Spezifikations-Formalismen und die Integration der entwickelten Werkzeuge.

18.11.2005 Disputation

1.1.2006 Mitarbeiter der OSC Embedded Systems AG