CARL VON OSSIETZKY universität OLDENBURG

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

An Automated Semantic-Based Approach for Creating Task Structures

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Dipl.-Inform. Matthias Büker

Gutachter:

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

Tag der Disputation: 04. Februar 2013

Author's address: Matthias Büker
OFFIS
Escherweg 2
D-26111 Oldenburg
Germany

E-Mail: matthias.bueker@offis.de
matthias.bueker@gmx.net

Danksagung

Zunächst möchte ich meinem Doktorvater Prof. Dr. Werner Damm sowohl für die fachliche Betreuung meiner Arbeit, die vielen wertvollen Anregungen und die Stellung anspruchsvoller Herausforderungen als auch für die persönlichen Gespräche danken.

Zudem danke ich der Prüfungskommission bestehend aus dem Vorsitzenden Prof. Dr. Ernst-Rüdiger Olderog, den Gutachtern Prof. Dr. Werner Damm und Prof. Dr. Martin Fränzle sowie Dr. Sibylle Fröschle für ihre Bereitschaft meine Arbeit zu begutachten. Insbesondere danke ich Prof. Dr. Martin Fränzle, dass er trotz der unglücklichen Umstände bereit war an meiner Disputation teilzunehmen.

Des Weiteren geht ein ganz großer Dank an meinen "Sparringpartner" Dr. Ingo Stierand, der immer für Fragen und Diskussionen zur Verfügung stand und bereit war große Teile meiner Arbeit Probe zu Lesen. Weiterer Dank gebührt Tayfun Gezgin, der insbesondere bei den formalen Teilen der Arbeit als weiterer Probeleser eine große Hilfe war sowie Günter Ehmen und Matthias Stasch, die mich beim Vorbereiten meiner Evaluation unterstützt haben. Ebenso danke ich Jan-Patrick Osterloh für die Betrachtung meiner Arbeit aus einem anderen Forschungsblickwinkel, um insbesondere die einleitenden Passagen verständlicher zu gestalten.

Ein besonderer Dank gilt auch meinem Gruppenleiter der Forschungsgruppe EEA am OFFIS, Dr. Stefan Henkler, der mir vor allem in der "heißen" Phase immer den Rücken frei gehalten hat um meine Arbeit fertig stellen zu können. Ich danke auch allen weiteren aktuellen und ehemaligen Mitgliedern der Gruppe EEA, die mir ein angenehmes und diskussionsfreudiges Arbeitsumfeld geboten haben bestehend aus Raphael Weber, Philipp Reinkemeier, Eike Thaden, Sunil Malipatlolla, Maike Rosinger, Stefanie Schlegl, Alexander Stühring und Guilherme Baumgarten. Ich danke auch allen Kollegen aus den benachbarten Gruppen HDM, SAV-RSM, SAV-DAT und HCD vom OFFIS sowie den Bereichen "Sicherheitskritische Eingebettete Systeme", "Hybride Systeme" und "Eingebettete Hardware/Software Systeme" der Universität Oldenburg für spannende Seminare, Klausurtagungen, Vorträge und Diskussionen. Ein großer Dank geht auch an alle, die an meinem tollen Doktorhut mitgearbeitet haben!

Einen weiteren Dank möchte ich aussprechen an meinen ehemaligen Gruppenleiter Prof. Dr. Alexander Metzner, der mich besonders in der Anfangsphase der Arbeit bei der Themenfindung und Entwicklung der Grundideen unterstützt hat.

Nicht zuletzt möchte ich meinen Eltern und ganz besonders meiner wundervollen Frau Mirja für ihre mentale und emotionale Unterstützung insbesondere in der nervenbeanspruchenden Endphase der Arbeit danken. Ohne Dich hätte ich das vermutlich nicht durchgestanden. Danke!

Zusammenfassung

Bei der Entwicklung von sicherheitskritischen eingebetteten Systemen müssen verschiedenste Aspekte berücksichtigt werden, um die Korrektheit des Systems nachzuweisen. Neben der rein funktionalen Korrektheit müssen solche Systeme auch Realzeiteigenschaften erfüllen, die typischerweise in so genannten *End-to-End Deadlines* ausgedrückt werden. Eine End-to-End Deadline fordert, dass bestimmte Ereignisse die im System beobachtet werden können innerhalb eines definierten Zeitintervalls auftreten. Dazu müssen alle Berechnungs- und Kommunikationsprozesse die für das Auftreten dieser Ereignisse notwendig sind innerhalb dieser Zeitspanne ausgeführt werden. Um solche Zeitanforderungen zu verifizieren sind verschiedene Techniken verfügbar bestehend aus formalen analytischen Methoden wie beispielsweise der Scheduling-Analyse oder Berechnungsmethoden wie Model-Checking. Um diese Methoden anwenden zu können, werden die Softwareanteile des Systems typischerweise als Tasknetzwerk dargestellt. Ein Tasknetzwerk ist ein gerichteter Graph bestehend aus Knoten die Anwendungsprozesse (Tasks) repräsentieren und Kanten die Abhängigkeiten zwischen Prozessen beschreiben.

In der industriellen Praxis gibt es eine Lücke zwischen der Spezifikation von Modellen durch einen Entwickler mittels eines Modellierungswerkzeugs wie MATLAB Simulink und der Darstellung als Tasknetzwerk, wie es für die Analyse von Zeiteigenschaften benötigt wird. Einerseits muss der Entwickler manuell entscheiden welche Teile des Modells zu einem Task zusammengefasst werden sollen und auf der anderen Seite muss die Semantik des Spezifikationsmodells bei der Erstellung des Tasknetzwerks erhalten bleiben. Andernfalls ist nicht sichergestellt, dass eine Analyse von Zeiteigenschaften tatsächlich das spezifizierte System überprüft. Hinzu kommt, dass in Werkzeugen wie Simulink keine Hardware modelliert und daher auch keine Allokation einzelner Softwareelemente auf Ausführungsressourcen wie Prozessoren betrachtet wird.

Der in dieser Dissertation vorgestellte Ansatz zur Erzeugung von Task-Strukturen (*Task Creation*) bildet den ersten Teil eines Prozessframeworks zur Untersuchung des Entwurfsraums eingebetteter sicherheitskritischer Systeme. Dieses Framework umfasst den gesamten Entwicklungsablauf, angefangen bei der Spezifikation von neuen Funktionen eines Fahrzeugs in Form von Simulinkmodellen, bis hin zu deren verteilter Ausführung auf hierarchischen elektronischen Hardwarearchitekturen. Bei der Task Creation wird in einem ersten Schritt aus einem gegebenen Simulinkmodell automatisch ein Tasknetzwerk abgeleitet, welches die partielle Ordnung von Blockausführungen erhält. Um diese Klasse von Modellen repräsentieren zu können, wird der Formalismus der Tasknetzwerke zu so genannten *Funktionsnetzwerken* erweitert.

Da ein aus Simulink gewonnenes Funktionsnetzwerk jedoch unausgewogen ist hinsichtlich seiner Knotengewichte im Sinne von Rechenintensität, werden in einem zweiten Schritt Knoten zu Tasks verschmolzen mit dem Ziel die so genannte *Kohäsion*

zu minimieren. Dieses Optimierungsmaß definiert, dass sich Knoten mit einer hohen Kommunikationsintensität anziehen und Knoten mit hohen Knotengewichten abstoßen. Das Ziel ist es Tasks mit sehr kleinen Gewichten zu vermeiden, um die Zahl der Taskwechsel zu reduzieren und die Kommunikation zwischen Tasks zu minimieren was die Busse der Hardwarearchitektur potentiell entlastet. Um die Korrektheit der Taskstruktur zu gewährleisten werden formale Kompositionsoperationen zum Verschmelzen von Knoten in einem Funktionsnetzwerk definiert. Für jede Operation wird nachgewiesen, dass diese die Semantik des Spezifikationsmodells im Sinne der Kausalität von Block- beziehungsweise Knotenausführungen erhält.

Abstract

For the design of safety-critical embedded systems, many different aspects have to be considered to guarantee the correctness of the system. Besides functional correctness, these systems also have to meet real-time constraints expressed in terms of end-to-end deadlines. End-to-end deadlines claim that certain system events must occur within a given time interval. This means that all involved computation and communication processes have to be finished within this time bound. To verify such time bounds, different techniques may be used, such as scheduling-analysis as an analytical method, and model-checking as a computational method. For this step, the software parts of the model are typically represented as a task network. A task network is a directed graph of task nodes representing application processes, and edges indicating dependencies between processes.

In practice, there is a gap between the specification model a designer creates in high level modeling tools like MATLAB Simulink, and the task network representation used for timing analysis. On the one hand, the designer has to decide manually which parts of the model should form a task, and, on the other hand, semantics of the specification model has to be preserved when creating the task network. Otherwise, it cannot be assured that the timing analysis really verifies the specified system. Additionally, tools such as Simulink abstract from any concrete target hardware architecture and how the different software parts are mapped to hardware resources for execution.

The approach proposed in this PHD-thesis is called *task creation* and forms the first part of a design space exploration framework for safety-critical embedded systems. This framework addresses the complete design flow from specification models of new automotive features captured in Simulink to their distributed execution on hierarchical bus-based electronic architectures. During task creation, in a first step, a task network is derived automatically from a given Simulink model by preserving the partial order of block executions. To be able to represent Simulink models, the formalism of task networks is extended to so-called *function networks* offering more expressiveness.

As the obtained network is typically unbalanced in the sense of computational node weights, in a second step, nodes are merged to form application tasks following an optimization metric called *cohesion*. This metric is defined such that nodes are attracted by high communication density and repelled by high node weights. The goal is to reduce task switching times by avoiding too lightweight tasks and to relieve the bus by keeping inter-task communication low. To obtain tasks correctly, we define formal composition operations for merging nodes in a function network. For each operation, we prove that it preserves specification semantics in terms of causality of block and node executions, respectively.

Contents

1. Introduction

This work is settled in the area of the design and analysis of safety-critical embedded systems with the focus on real-time properties. Embedded systems are computer systems that are part of other larger systems or devices with a certain purpose, and are found in multiple domains, such as aviation, automotive or automation engineering. Typical tasks of embedded systems are to control and monitor such systems, which may include also mechanical parts.

Because embedded systems often control safety-critical tasks - in particular in the aviation and automotive domain - the verification of certain properties concerning safety, functionality and timing is key to make them available for the productive use. Functional and timing properties are usually strongly related because the correct functionality is only assured if timing constraints are met. Typical timing constraints are end-to-end deadlines between specific system events. For example, the actuator triggering the airbags of a car has to be activated within a time bound of 15 to 30 milliseconds after a collision has been detected by a crash sensor to guarantee that the airbags inflate in time to protect the passengers. To verify timing constraints, there exist different approaches, which can be divided into analytical methods as scheduling analysis [66], and computational methods as model-checking [19]. To be able to apply those analysis techniques, the software parts of the system are modeled as a process or task network [24]. In general, a task network is a graph where nodes represent application tasks implemented as executable program code. Tasks are connected by edges modeling dependencies between tasks. For modeling communication, task networks may also contain signals, which transmit data between tasks. The occurrence of events in task networks is typically described by event streams and event models [73, 40].

The design of safety-critical embedded systems involves several phases, like definition of requirements, system specification, and implementation of the system on a hardware platform. This work is part of a framework [10, 11] that aims at automating significant parts of the design flow in a typical scenario for embedded application development in the automotive domain. It was developed within the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). In this framework, a common hierarchical bus-based target architecture from the automotive domain is considered, where electronic control units (ECUs) are clustered in subsystems. Subsystems are connected by a backbone TDMA (Time Division Multiple Access) bus. In Figure 1.1 an example of such an architecture is depicted with three subsystems connected by a FlexRay backbone bus, where each subsystem contains a set of ECUs, which communicate via a local CAN bus.

Motivated by the iterative design process in industrial practice, we assume that this architecture is already pre-deployed with the functionality of an existing set of applications in terms of software tasks. This is sketched in Figure 1.1, where a part

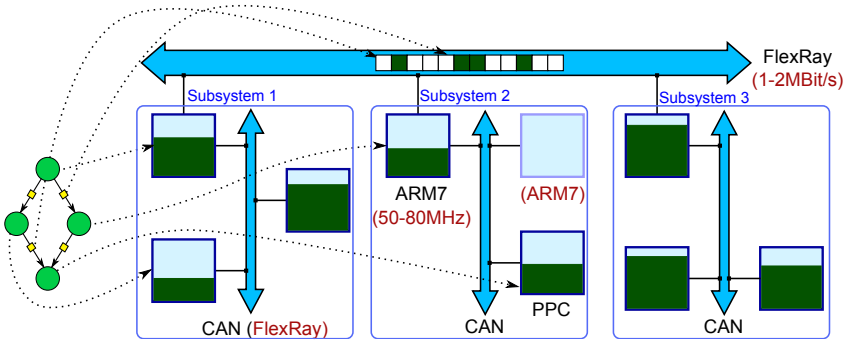


Figure 1.1.: Distributed Hardware Architecture

of the pre-deployed task network is shown on the left. The dotted arrows indicate the allocation of tasks to processors and signals to buses. The utilization of the existing processors, induced by pre-deployed tasks, is indicated by partially filled boxes. For example, in *Subsystem 1*, the ECU at the top left is filled with around 70% utilization.

For the next generation of the system, a new customer feature should be implemented. This might be, for example, a new driver assistance system. For the feature specification of embedded systems in the industrial practice, there exist a number of high-level modeling tools for the individual domains. For this work, we will focus on the automotive domain, where Simulink is a standard tool for system modeling.

To offer sufficient computational capacity for the new feature, the hardware architecture may be modified by adding new ECUs, or by replacing existing ones by more powerful ECU types. In Figure 1.1, allowed modifications are shown in parenthesis. For example, the left ECU of *Subsystem 2* is an ARM7 processor that may be clocked with 50 to 80 MHz. The empty box on the top right of the same subsystem indicates the possible addition of another ARM7 ECU. Each modification induces costs depending on the added or replaced ECU type. The overall goal of the framework is to find a conservative cost-optimized extension of the existing architecture to implement the new feature while meeting all timing constraints.

1.1. Overview and Goals

The approach presented in this work forms the front-end of the design space exploration framework shown in Figure 1.2, where the scope of this work is indicated by a dashed rectangular box. In this first part, a task structure is automatically derived from a Simulink specification model of a new feature. In the succeeding *Design Space Exploration* (DSE), this task structure should be deployed to an existing system in terms of software tasks allocated to a distributed hardware architecture.

The first goal of this work is to define a *Translation (1)* of the structure of a Simulink model and its timing properties into a task network. To be able to represent the semantics of Simulink models, the task network formalism is extended to a so-called *function network*. For each block of the Simulink model, *worst case execution times*

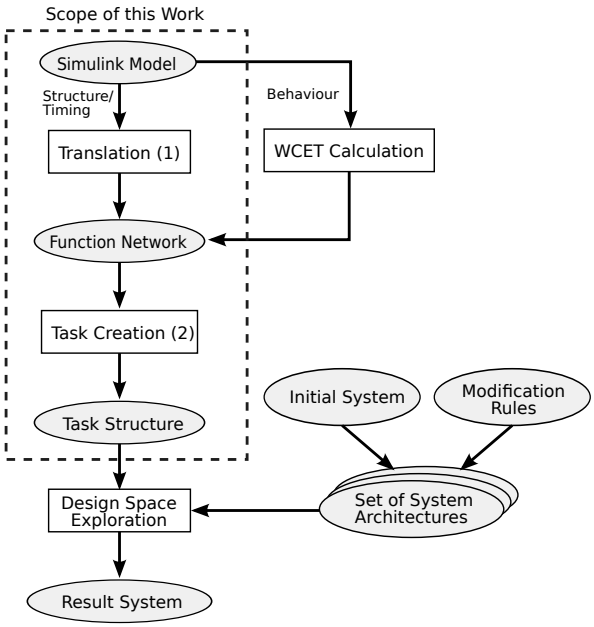


Figure 1.2.: Overview of Design Space Exploration Framework

(WCETs) are estimated based on generated code. These WCETs are used to estimate computational weights for the obtained function network nodes. Please note, that WCET calculation itself is not part of this work.

One scientific challenge for this part is to assure the correctness of the translation from a synchronous language like Simulink to an asynchronous task network formalism meaning that same input values lead to same output values. This is done by showing that the execution semantics of Simulink in terms of the partial order of block executions is preserved such that signals are computed in a valid order. The correct behavior of single blocks is ensured by generating code using existing code generators. Another challenge is to define an extension of the task network formalism that enables to represent the execution semantics of Simulink models and to prove the relevant properties needed to show the correctness of the translation. Because we aim at an implementable system, also the question of boundedness is highly relevant meaning that the software system is implementable with a finite set of (memory) resources. Thus, a class of function networks is defined where boundedness is decidable and which is sufficient to represent Simulink models.

Because the network obtained from a Simulink translation typically consists of a large number of nodes with unbalanced computational weights and a high amount of communication, the second goal of this work is to derive a reasonable task structure from the translated function network. To realize this, in the so-called *Task Creation (2)* step, nodes are iteratively merged into tasks by formal composition operations on function networks. The scientific challenges in this part can be divided into two

categories: First, a methodology and optimization metric for merging nodes needs to be developed with the aim to derive a task structure that is suitable to be mapped to a pre-deployed distributed hardware architecture as it is considered in the design space exploration. Second, it needs to be assured that the execution semantics of the specification model is preserved by the formal composition operations to merge function nodes. This is done by showing that the partial order of Simulink block executions is not violated by any merging operation.

The resulting task structure serves as input to the design space exploration process, which has the goal to find a feasible task allocation on the distributed architecture with minimum costs. The set of available system architectures is comprised of the initial system and a set of allowed modification rules. However design space exploration is no conceptional part of this work, it is used to evaluate the proposed approach in the overall context of the framework.

Overview of Related Work A lot of work has been done concerning the translation of Simulink into other synchronous or asynchronous languages. In [81], Simulink models are translated to Lustre to partition the generated code into modules that are executed on different processors communicating via a time-triggered bus. Here, the focus lies on efficiently generating modular code and separating it into different modules respecting a global partial order. In our approach, we use existing code generators to generate code for Simulink blocks and thus the question of optimal code generation is not in the scope of this work but can be considered as supplementary.

A further work was presented in [80], where a synchronous model is implemented on a loosely time-triggered architecture. The authors only consider single-rate models while for our approach the semantic preservation for multi-rate models is one major part. Furthermore, they cannot always guarantee that no data is lost because they allow Simulation steps of Simulink to overlap. Similar problems occur in [71], where tasks are identified manually from a Simulink model and scheduled in a fixed-priority preemptive scheduling. In our translation, we follow the Simulink simulation semantics and forbid overlapping executions by defining respective end-to-end deadlines to assure data consistency.

In [6] and [7], an overview is given on the basic idea of synchrony and the most important synchronous languages and it is discussed how to translate a synchronous language to an asynchronous one in general. In contrast to these approaches, we do not aim at representing the complete functional behavior of a synchronous model in our translation to an extended task network model. Instead, we assure the correctness of our translation by preserving the partial order of signal updates induced by block executions. The functional correctness of the computations of single blocks is assured by applying existing code generators to generate code for those blocks.

An approach comparable to the idea of task creation was proposed in [23], where an optimization of the multi-task implementation of Simulink models with real-time constraints is considered. The optimization goal is to reduce the use of rate transition blocks between different synchronous sets to minimize buffering and latencies. The tasks for the scheduling analysis are either determined by the synchronous sets or

are also part of the optimization problem. Beside the optimization goal, the main difference to our work is the target hardware in terms of a single processor, while we consider a distributed bus-based architecture.

Another work from Kugele et al. [46] is also based on synchronous languages and presents a way to deploy clusters in terms of tasks on a multi-processor platform. This allocation process is completed by a scheduling analysis. The authors also raise the question of how to generate clusters of nodes to form tasks but assume that this is a decision that is taken manually by the user.

A more detailed discussion of related work can be found in the summary of each main chapter of this work.

1.2. Concept and Approach

In Simulink, the functionality of a system is modeled in terms of synchronous block diagrams, where each block fulfills a specific function. Blocks are connected by signals delivering values from one block to another. A block may first be executed if all its input signals have been updated by the execution of preceding blocks. Sample times are used to determine at which points in time a block is executed, and consist of a period and an initial phase offset. *Synchronous sets* are defined as a set of connected blocks with the same sample time. Blocks of different synchronous sets may also be connected by *rate transition blocks* if their periods are integer multiples. Those blocks ensure that the respective signal values are available if the succeeding block is executed and that the partial order is guaranteed if both connected blocks are executed in one simulation step. To structure the model, subsystems may be used, which are hierarchical blocks containing other blocks and signals. In Figure 1.3, an example of a Simulink block diagram is shown consisting of three synchronous sets with sample times ST_1, ST_2, and ST_3. A rate transition block named RTB connects the synchronous sets with sample times ST_1 and ST_2.

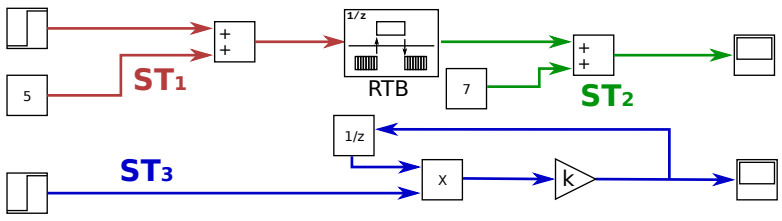


Figure 1.3.: Example of a Simulink Block Diagram

In Simulink, it is possible to model discrete systems in terms of the actual embedded controller as well as continuous systems, which may be used to model the environment. Simulink models can be simulated to evaluate the functionality of the system. During simulation, block execution in Simulink does not consume time. Instead, simulation is performed in discrete steps for discrete controller models. Thus, it is implicitly assumed that in each step all needed blocks can be executed to deliver results at their

output ports. Obviously, the assumption of instant block execution cannot hold for any implementation because task execution on an ECU always consumes time.

This leads to a gap between specification models, as they are designed in Simulink, and task networks, which are the basis for timing analysis. To close this gap, it first needs to be assured that a Simulink model can be correctly translated into a task network. Because we aim at analyzing timing properties, we abstract from the concrete functional behavior of blocks. Instead, we define a translation to be correct if it preserves the partial order of Simulink block executions and satisfies the assumption that all blocks are executed within one simulation step.

To be able to represent the semantics of a Simulink model in terms of synchronization of multiple input signals and consistent behavior of blocks with different sample times, we define an extended task network called *function network*, which allows more expressiveness. In a function network, tasks are denoted as *function nodes* and edges as *channels*. As a first extension, we introduce AND-activation (synchronization) of incoming channels of function nodes to translate blocks with multiple input signals. To model rate transition blocks that convert from a slower to a faster period, we further define OR-activation (superposition) to model the additional activations. A rate transition from a faster to a slower block must only write each $n'th$ signal update, where n is the integer multiple of the respective sample times. To realize this, we define an internal state transition system for function nodes, which allows to produce different output events depending on the state and received input events.

As a further block type, in Simulink *data store memory blocks* are used to model data exchange between blocks without inducing a partial order relation. To allow the modeling of such data dependencies, we define special communication nodes called *data nodes*. Data nodes may be of different types as shared memory and FIFO buffers. To be able to show preservation of certain semantic properties during translation and task creation, semantics of function networks needs to be defined formally. This is done by defining atomic components in terms of timed automata that are composed to function network elements like function nodes and channels.

Based on the function network formalism, we define a translation from Simulink specification models, where blocks are translated to function nodes and signals to channels. Because the hierarchy of the model in terms of subsystems does not influence the semantics but is only used to structure the model, we translate the flattened block diagram. To show that the partial order of block executions is preserved in the function network translation, we relate Simulink signal updates to function network events. Furthermore, it has to hold that all block executions of a simulation step are finished before the next simulation step starts.

After the translation, the process continues with the decision which parts of the specification model should form a task. The code generator of Simulink offers a multi-task implementation, where all blocks with the same sample time are realized as one task. However, this is not necessarily the best choice if tasks should be allocated to a distributed hardware architecture with pre-deployed tasks. In this scenario, ECUs usually have a high utilization by the existing tasks in order to use hardware resources efficiently. Thus, tasks that need a high amount of computational capacity might not be deployable to any existing ECU. Hence, we need to find a partitioning of the

specification functionality into a set of tasks with "reasonable" computational needs. To describe the computational capacity a task needs for its execution, we introduce *node weights*. Thus, one goal for task creation is to avoid tasks with too large weights. On the other hand, task weights should also not be too small. If we would, for example, assume each atomic block of a Simulink model to be represented as one task, this would lead to a very high number of tasks with comparatively small weights. If such tasks were deployed to an ECU, this would lead to frequent task switching, also denoted as *thrashing*. In summary, task weights should be balanced to avoid both too heavy and too lightweight tasks.

Another important aspect is that usually many blocks of a Simulink model are connected by signals leading to a high amount of communication. If such a fine-granular task network would be spread over a distributed hardware architecture, this would lead to a high bus utilization. But buses are often the bottleneck of distributed systems and can hardly be upgraded, or only with very high costs. Thus, a further goal for task creation is to find a task set with a minimum inter-task communication to relieve the buses. To cover all these issues, we define an optimization metric for task creation called *cohesion*. It defines that nodes are attracted by edges with a high communication density and repelled by high node weights. This leads to minimizing inter-task communication and balancing task weights as well.

To perform task creation, an algorithm is defined that partitions function nodes into task sets while minimizing cohesion. This is done in two steps, where first an initial solution is created that is improved in a second step by a variant of the Kernighan-Lin heuristic [43]. The result of the algorithm is a set of node partitions, where each partition represents a task. To finish task creation, all function nodes within the same partition are merged to one node in the function network. For this, we define a set of formal composition operations. A merging of two nodes means that they will be considered as one task and thus will be executed on one computation resource. This leads to less task activations, and thus also to less task switches. As for the translation from Simulink, we also show for the task creation operations to preserve the original execution semantics of the specification model.

1.3. Outline

First, we introduce in Chapter 2 the fundamentals that are needed for this work in terms of formalisms like timed automata, task networks, and tools like Simulink. In Chapter 3, we define function networks as an extension of task networks. To be able to show semantics preservation of the Simulink translation and task creation process, we define causality and timing properties of function networks that should be preserved. Furthermore, we show for a class of function networks that boundedness is decidable by propagation event models through the network.

In Chapter 4, we define a translation of Simulink block diagrams into function networks and show that this translation preserves the partial order semantics of Simulink. In Chapter 5, we present the task creation approach starting with defining the optimization metric cohesion and weights of function nodes and channels. For merging

nodes, we define formal composition operations and show that they preserve the specification semantics in terms of causality. Furthermore, an algorithm is proposed to perform node partitioning and the approach is evaluated with a case study.

In Chapter 6, we shortly present the design space exploration process of the framework and discuss the role of task creation. Furthermore, we evaluate the task creation approach by applying the design space exploration with different task networks of the same specification model. Finally, in Chapter 7, we conclude this work by summarizing and discussing the results with respect to the defined goals.

2. Basics

In the context of this work, we make use of some formalisms and tools, which are introduced in this chapter. We start with general definitions and notations in Section 2.1. In Section 2.2, we define timed languages and timed automata, which build the base to define semantics of function networks.

To describe the occurrence of events in a function network, event streams and event models are used, which are presented in Section 2.3. In Section 2.4, task networks are introduced which can be considered as predecessor of function networks. Finally, in Section 2.5, we introduce the high level modeling tool suite MATLAB Simulink, which we assume as starting point for our design process.

2.1. General Definitions and Notations

First, we define the following notations for sets of numbers:

- $\mathbb{N}_0$: Set of natural numbers including zero.
- $\mathbb{N}^+$: Set of natural numbers except zero.
- $\mathbb{R}$: Set of real numbers.
- $\mathbb{R}_0^+$: Set of positive real numbers including zero.
- $\mathbb{R}^+$: Set of positive real numbers except zero.
- $\mathbb{Q}_0^+$: Set of positive rational numbers including zero.
- $\mathbb{Q}^+$: Set of positive rational numbers except zero.

Because execution times of tasks and function nodes, respectively, will be defined as intervals of natural numbers, we define the following operations on intervals.

Definition 2.1.1 (Interval Arithmetic) *Let $[a, b]$ and $[c, d]$ be intervals with $a, b, c, d \in \mathbb{N}_0$. We define addition, minimum and maximum of intervals as follows:*

- *Addition:* $[a, b] + [c, d] := [a + c, b + d]$
- *Minimum:* $\min([a, b], [c, d]) = [\min(a, c), \min(b, d)]$
- *Maximum:* $\max([a, b], [c, d]) = [\max(a, c), \max(b, d)]$

$\diamond$

A basic concept to show semantics preservation of the Simulink translation is the partial order relation. Based on [82], we define a partial order as follows:

Definition 2.1.2 (Partial Order) *A (strict) partial order $PO(\Sigma)$ is a binary relation $<$ over a set Σ that is irreflexive, antisymmetric, and transitive, i.e. $\forall a, b, c \in \Sigma$:*

$$(1)\ \neg(a < a)\ \textit{(irreflexive)}$$
$$(2)\ a < b\ \wedge\ b < a\ \implies\ a = b\ \textit{(antisymmetric)}$$
$$(3)\ a < b\ \wedge\ b < c\ \implies\ a < c\ \textit{(transitive)}$$

We write $(a, b) \in PO(\Sigma)$, if $a < b$ where a is called predecessor of b, and b is called successor of a. ◇

2.2. Timed Languages and Timed Automata

Timed automata are an extension of finite state machines with time and were defined by Alur and Dill [3]. Timed automata offer a set of special variables named clocks, which increase uniformly when time passes. Based on values of clocks, guards of transitions allow to define constraints when a transition may be taken. Clocks may also be reset to zero by transitions.

Furthermore, invariants allow to constrain how long an automaton is allowed to stay in a specific state. States are often referred to as *locations* because the actual state of a timed automaton is determined by the location *and* the value of all clocks.

The language of timed automata is defined in terms of timed words forming a timed language. Based on [3] and [85], we define timed words and timed languages as follows:

Definition 2.2.1 (Timed Word and Timed Language) *Let Σ be a set of events, and let $\mathbb{T} = \mathbb{R}_0^+$ be a time domain. An infinite sequence $\omega = (\sigma_i, t_i)_{i \in \mathbb{N}^+}$ where $\sigma_i \in \Sigma$, $t_i \in \mathbb{T}$ is a* timed word *if and only if:*

1. *$\forall i < j : t_i \leq t_j$ (Monotonicity)*
2. *$\forall c \in \mathbb{T}\ \exists i : c \leq t_i$ (Progress)*

Let $\Omega(\Sigma, \mathbb{T})$ be the set of timed words over Σ. Then $L \subseteq \Omega(\Sigma, \mathbb{T})$ is a timed language *over Σ. For a timed word $\omega = (\sigma_i, t_i)_{i \in \mathbb{N}^+}$, we also write $(\sigma_1, t_1)(\sigma_2, t_2)...(\sigma_i, t_i)...$. For each element of a word ω we write $(\sigma_i, t_i) \in \omega$.* ◇

For system modeling, network of timed automata are defined, which are used in tools like UPPAAL [5, 49]. In such a network, a set of timed automata communicate via *synchronization channels*, where a sender channel is denoted as $c!$ and its respective receiver channel as $c?$. Furthermore, *urgent* synchronization channels are defined, where delays must not occur if a synchronization transition on an urgent channel is enabled. Edges using urgent channels for synchronization cannot have clock guards. Also locations may be urgent, meaning that time is not allowed to pass when the system is in such a location. In *committed* locations it additionally has to hold that

they are left with the next transition. If more than one automaton is in a committed location, this has to hold for at least one of the committed locations.

We define clock constraints and clock valuations as done in [33] as follows:

Definition 2.2.2 (Clock Constraint [33]) *Let C be a set of clocks. A clock constraint is defined by the syntax*

$$\varphi ::= \ c_1 \sim t \mid c_1 - c_2 \sim t \mid \varphi \wedge \varphi,$$

where $c_1, c_2 \in C$, $t \in \mathbb{Q}_0^+$ and $\sim \in \{\leq, <, =, >, \geq\}$. The set of all clock constraints over the set of clocks C is denoted by $\Phi(C)$. ◇

Definition 2.2.3 (Clock Valuation [33]) *Let C be a set of clocks. Clock valuation is a function*

$$\nu : C \to \mathbb{R}_0^+$$

assigning each clock in C a non-negative real number.

We denote $\nu \models \varphi$ the fact that a clock constraint φ evaluates to true under the clock valuation ν. We use 0_C to denote the clock valuation $\{c \mapsto 0 \mid c \in C\}$, abbreviate the time shift by $\nu + d := \nu(c) + d$ for all $c \in C$, and clock resets by $\nu[\varrho \mapsto 0]$ with $\nu[\varrho \mapsto 0](c) = 0$ if $c \in \varrho$, and $\nu[\varrho \mapsto 0] = \nu(c)$ else, where $\varrho \subseteq C$. ◇

For the definition of timed automata, we rely on the definition of [85], while we additionally allow the set of locations to be infinite.

Definition 2.2.4 (Timed Automaton)
A Timed Automaton *(TA) is a tuple $A = (L, L^c, l^0, \Sigma, \Sigma^u, C, Inv, R)$ where*

- *L is a non-empty set of locations, $L^c \subseteq L$ is the set of committed locations and $l^0 \in L$ is an initial location,*
- *Σ is a finite alphabet of channels, inducing the action set $\Sigma_{?!} = \{a? \mid a \in \Sigma\} \cup \{a! \mid a \in \Sigma\} \cup \{\tau\}$, where τ denotes internal actions,*
- *$\Sigma^u \subseteq \Sigma$ is the set of urgent channels,*
- *C is a finite set of clocks.*
- *$Inv : L \to \Phi(C)$ is a mapping which assigns an invariant to each location,*
- *$R \subseteq L \times \Sigma_{?!} \times \Phi(C) \times 2^C \times L$ is a set of transitions. A tuple $t = (l, \sigma, \varphi, \varrho, l')$ represents a transition from location l to location l' annotated with the action σ, the constraint φ and a set of clocks ϱ which have to be reset.*

  For t we also write $l \xrightarrow{\sigma, \varphi, \varrho} l'$. ◇

As introduced in [85], we define a mapping *chan* "such that each element of the action set is mapped to the corresponding channel, e.g., $chan(\sigma!) = chan(\sigma?) = \sigma$. If $chan(\sigma) \in \Sigma^u$, then $\varphi = true$ i.e. no guards are allowed on transitions synchronizing on urgent channels" [85].

Following [85], we can model a system by a network of timed automata with pairwise disjoint clock sets as follows: "A network of n timed automata is denoted by $A_1||...||A_n$ and modeled by a timed transition system. During computation, each automaton is in a specific location. The locations of all automata are collected in a control vector denoted by $l = (l_1, \ldots, l_n)$. A change from location l_i to l_i' of the ith automaton of a given network is denoted by $l[l_i/l_i']$ "[85]. A timed transition system modeling a network of timed automata is defined in [85] as follows:

Definition 2.2.5 (Timed Transition System [85]) *Let A_i be a network of timed automata with $A_i = (L_i, L_i^c, l_i^0, \Sigma_i, \Sigma_i^u, C_i, Inv_i, R_i)$, $i \in \{1, ..., n\}$ and pairwise disjoint sets of clock variables. The semantics of such a network is defined in terms of a timed transition system denoted as*

$$\mathcal{T}(A_1||...||A_n) = (\Sigma_{out}, Conf, Conf^0, C, \rightarrow)$$

where

- $\Sigma_{out} = \bigcup_{i=1}^n (\Sigma^u(A_i) \setminus \bigcup_{j=1, j\neq i}^n \Sigma^u(A_j))$ *the so-called open synchronization channels of the network, inducing the action set* $\Sigma_{?!,out} = \{a? \mid a \in \Sigma_{out}\} \cup \{a! \mid a \in \Sigma_{out}\}$
- $Conf = \{(l, \nu) \mid l_i \in L_i \ \wedge \ \nu \models \bigwedge_{j=1}^n Inv_j(l_j)\}$ *is the set of configurations,*
- $Conf^0 = (l^0, 0_C)$, *where* $l^0 = (l_1^0, ..., l_n^0)$ *is the initial location vector and* 0_C *is the initial clock valuation, i.e.* $\{c \mapsto 0 \mid c \in C\}$,
- $C = \bigcup_{j=1}^n C_i$ *is the set of clock variables,*
- $\rightarrow \subseteq Conf \times (\bigcup_{j=1}^n \Sigma_{?!,j} \dot\cup \mathbb{R}_0^+ \cup \{\tau\}) \times Conf$ *is the transition relation. A transition* $((l, \nu), \lambda, (l', \nu'))$, *also denoted by* $(l, \nu) \xrightarrow{\lambda} (l', \nu')$, *has one of the following types.*
  - *Delay transitions* $(l, \nu) \xrightarrow{t} (l, \nu + t)$ *with* $t \in \mathbb{R}_0^+$ *can occur, if* $\nu + t \models \bigwedge_{j=1}^n Inv_j(l_j)$.*Moreover, neither an urgent channel is active nor an automaton is in a committed state, i.e.*
    - *(i) there exist no* $i, j \in \{1, ..., n\}$ *and* $\sigma \in \Sigma^u$ *with* $(l_i, \sigma!, \varphi_i, \varrho_i, l_i') \in R_i$ *and* $(l_j, \sigma?, \varphi_j, \varrho_j, l_j') \in R_j$
    - *(ii) there exist no* $i \in \{1, ..., n\}$ *with* $l_i \in L_i^c$.
  - *Local transitions* $(l, \nu) \xrightarrow{\lambda} (l', \nu')$ *with* $\lambda \in \Sigma_{?!,out} \cup \{\tau\}$ *can occur, if for some* $i \in \{1, ..., n\}$ *it holds that* $(l, \lambda, \varphi, \varrho, l') \in R_i$, *such that* $\nu \models \varphi$, $l' = l[l_i/l_i']$, $\nu' = \nu[\varrho_i \mapsto 0]$ *and* $\nu' \models Inv_i(l_i')$. *Furthermore it holds that if* $l_k \in L_k^c$ *for some* $k \in \{1, ..., n\}$ *then* $l_i \in L_i^c$.
  - *Internal transitions* $(l, \nu) \xrightarrow{\tau} (l', \nu')$ *can occur, if for some* $i, j \in \{1, ..., n\}$ *with* $i \neq j$ *and* $\sigma \in \Sigma_i \cap \Sigma_j$ *there are transitions* $(l_i, \sigma!, \varphi_i, \varrho_i, l_i') \in R_i$ *and* $(l_j, \sigma?, \varphi_j, \varrho_j, l_j') \in R_j$ *such that* $\nu \models \varphi_i \wedge \varphi_j$, $l' = l[l_i/l_i'][l_j/l_j']$, $\nu' = \nu[\varrho_i \mapsto 0, \varrho_j \mapsto 0]$ *and* $\nu' \models Inv_i(l_i') \wedge Inv_j(l_j')$.

    Furthermore it holds that if $l_k \in L_k^c$ *for some* $k \in \{1, ..., n\}$ *then* $l_i \in L_i^c$ *or* $l_j \in L_j^c$.

◇

Following [85] "given a configuration (l, ν) of a network of n timed automata M, we say M can *move* to configuration (l', ν') by synchronizing on $\sigma \in \Sigma_{?!,out}$ after delay $d \in \mathbb{R}_0^+$, denoted by $(l, \nu) \xrightarrow[d]{\sigma} (l', \nu')$, if and only if there exists a sequence

$$(l, \nu) \xrightarrow{\rho_1} (l_1, \nu_1) \xrightarrow{\rho_2} \ldots \xrightarrow{\rho_n} (l_n, \nu_n) \xrightarrow{\sigma} (l', \nu'), n \in \mathbb{N}_0,$$

with $\rho_i \in \mathbb{R}_0^+ \cup \{\tau\}$, $1 \leq i \leq n$, such that $\sum_{i=1}^{n} \rho_i = d$ "[85] where $\tau + d = d$.

For the definition of the language of a timed transition system, we slightly differ from [85] and define the language also on internal symbols and not only on open synchronization channels as follows:

Definition 2.2.6 (Language of TTS) *Let $M = (\Sigma_{out}, Conf, Conf^0, C, \rightarrow)$ be a network of n timed automata. A finite sequence $\bar{\gamma} = ((l_i, \nu_i), \lambda_i)_{0 \leq i < n}$ of pairs of configurations of M and actions $\lambda_i \in \Sigma_{?!,out} \dot{\cup} \mathbb{R}_0^+ \cup \{\tau\}$ for all $0 \leq i < n$ is called* partial computation *of M of length n if and only if $((l_0, \nu_0), \lambda_0) = ((l^0, 0_C), \tau)$ and if adjacent pairs are* consecutive*, i.e. $(l_i, \nu_i) \xrightarrow{\lambda_{i+1}} (l_{i+1}, \nu_{i+1})$ for $0 \leq i < n$. An infinite sequence γ of pairs of configurations of M and actions is called* computation *of M if each prefix of γ is a partial computation.*

We say a partial computation $\bar{\gamma}$ computes *the (finite) timed word $\omega = ((\sigma_i)_{0 \leq i < n}, (t_i)_{0 \leq i < n})$, denoted by $M \overset{\omega}{\twoheadrightarrow}_n (l', \nu')$, if and only if there exists a configuration (l', ν') of M such that $(l^0, 0_C) \xrightarrow[t_0 - t_{-1}]{\sigma'_0} \ldots \xrightarrow[t_{n-1} - t_{n-2}]{\sigma'_{n-1}} (l', \nu')$ with $\sigma_i = chan(\sigma'_i)$ for all $i \in \{0, ..., n-1\}$. We set $t_{-1} = 0$ to uniformly cover the case $i = 0$.*

The TTS M computes *the timed word ω if and only if it computes each prefix of ω. The* language *$L(M)$ of M is the set of timed words computed by M.*

◇

We apply two further extensions introduced for UPPAAL timed automata in [5, 49]. First, guards may be also defined over bounded integer variables, which are considered as part of the state space. Second, we allow broadcast channels where one sender $c!$ synchronizes with an arbitrary number of receivers $c?$.

2.3. Event Streams and Event Models

Event streams are used in real-time theory and in particular in scheduling analysis to describe the occurrence of task activations in terms of events within a specific time interval. They are defined by four characteristic functions: The η^- and η^+ functions give for a specific time interval the minimum and maximum number of events that may occur i.e., $\eta^-(\Delta) = n$ means that within a time interval of Δ at least n events may be observed. The δ^- and δ^+ functions give for a specific number of events the minimum and maximum time distance where this number of events may occur. Hence, $\delta^+(n) = \Delta$ means that the maximum time distance between n events is bounded by Δ. According to [73], both the pair of η and δ functions are each sufficient to represent the characteristics of an event stream because they can be derived from each other.

2.3.1. Event Models

Based on event streams, *event models* were defined to represent a certain class of event streams. In [40, 66], Jersak and Richter describe an approach to perform compositional scheduling analysis using standard event models. The most common models are the *strictly periodic* event model and the *periodic with jitter* event model where activations may be delayed by a jitter. Jitter may also lead to overlapping invocations of events if it becomes greater than the period, which is known as burst.

The periodic event model with jitter was defined by Jersak in [40] as follows while we use slightly different notations to avoid ambiguities:

Definition 2.3.1 (Periodic with Jitter Event Model) *Let $Period \in \mathbb{R}^+$ be a period and $Jitter \in \mathbb{R}_0^+$ be a jitter. Periodic event streams with jitter are defined as*

$$\eta^+(t)_{P+J} = \left\lceil \frac{t + Jitter}{Period} \right\rceil$$
$$\eta^-(t)_{P+J} = \max\left(0, \left\lfloor \frac{t - Jitter}{Period} \right\rfloor\right)$$

◇

An extension of this model was proposed in [24], where for the η^- and η^+ functions different periods P^+ and P^- are defined. Thus, the period may vary between these bounds. This additionally allows to model *sporadic* streams where a minimum inter-arrival time d^- between events is defined. This is realized by choosing $P^+ = \infty$ and $P^- = d^-$.

The periodic/sporadic event model with jitter was defined in [24] as follows:

Definition 2.3.2 (Periodic/sporadic Event Model with Jitter) *The* periodic/sporadic event model with jitter *is a function $\mathcal{EM} : (\Sigma \times \mathbb{T}^3) \to 2^{\Omega(\Sigma)}$, where $\mathcal{EM}(e, P^-, P^+, J)$ characterizes the set of timed traces with occurrences of events e with lower and upper periods bounds $P^- \in \mathbb{R}^+$, $P^+ \in \mathbb{R}^+ \cup \{\infty\}$ and jitter $J \in \mathbb{R}_0^+$ is defined for $0 \leq J < P^- \leq P^+$ as*

$$\eta^+(\Delta) = \left\lceil \frac{\Delta + J}{P^-} \right\rceil,$$
$$\eta^-(\Delta) = \max\left(0, \left\lfloor \frac{\Delta - J}{P^+} \right\rfloor\right)$$

◇

In Figure 2.1, the η functions for the periodic and sporadic event model are depicted on the left and the δ functions on the right.

Another event model has been introduced by Thiele et al. in [78, 47], where a performance analysis technique is proposed that is based on a formalism named Real-Time Calculus (RTC). Here, so-called arrival curves are used to model the computation that is requested by a process. In the following we give the definition of arrival curves

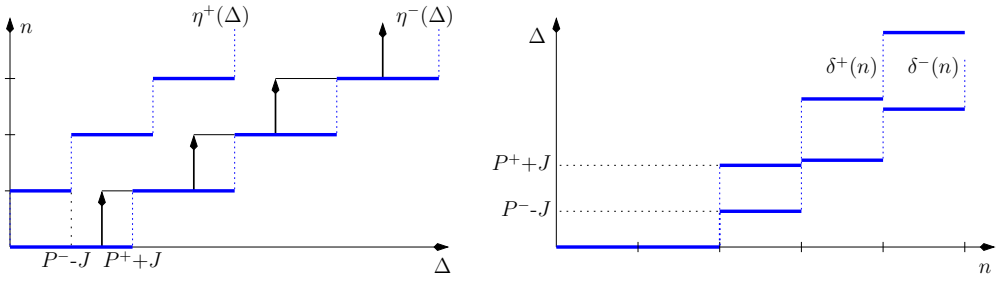


Figure 2.1.: Periodic/Sporadic event stream model (Source: [24])

from [78, 47] while we differ from the original notation by denoting the minimum and maximum distance (period) between event occurrences as P^- and P^+ instead of δ^u and δ^l (where $P^- = \delta^u$, $P^+ = \delta^l$).

Definition 2.3.3 (Real Time Calculus (RTC) Arrival Curves) *In RTC an arrival curve is defined as upper and lower staircase function as follows:*

$$\alpha^u(\Delta) := N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor$$
$$\alpha^l(\Delta) := N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor$$

describing traces which contain at least $\alpha^l(\Delta)$ and at most $\alpha^u(\Delta)$ event signals for any interval of length $\Delta \in \mathbb{R}_0^+$. The parameter N^u can be understood as burst capacity, which describes the number of events producible in zero time according to curve α^u. The parameters P^- and P^+ specify the minimum/maximum distance of two successive events. The absolute values of parameter N^l of the lower curve can be understood as the fictitious numbers of periods (P^+) before event emission has to be enforced every P^+ time unit. ◇

This model has some differences to the previous models because it does not define a jitter. But it also allows different periods P^- and P^+ for the upper and lower curves and additionally defines initial values N^l and N^u for both curves. The initial value for the upper curve may be modeled by the model of Jersak as well by choosing a jitter greater than the period. Another difference is the definition of the η^+ and α^u function. If we assume $Jitter = 0$ and $N^u = 1$ (which is implicitly the case in the model of Jersak), the value of $\eta^+(k \cdot Period)$ is k in the model of Jersak and Richter, while in the RTC model $\alpha^u(k \cdot P^-)$ is defined to be $k + 1$. Thus, analysis results may differ depending on the utilized model.

2.3.2. AND- and OR-Operations on Event Models

Jersak defined in [40] operations for the synchronization (AND) and superposition (OR) of event models where we focus again on the periodic event model with jitter.

AND-activation of event streams occurs if a task needs multiple inputs to be activated leading to the need to synchronize all input streams and to buffer events until all synchronization partners have arrived. To ensure that an AND-activated task is ever activated these input buffers need to be bounded, which is assured for periodic event models with jitter if the period of all input models is the same. We will later prove in Section 3.1 that this is a sufficient *and* necessary condition to decide boundedness of buffers for periodic event models with jitter and an additional offset. According to Jersak [40], AND-activation of periodic event models with jitter is defined as follows:

Definition 2.3.4 (AND-Activation - Periodic with Jitter) *AND-activation of n input event models described by the parameters $(Period_i, Jitter_i)$ with $Period_i = Period_j$ for all $i, j \in \{1, ..., n\}$ is defined as the event model*

$$(Period_{AND}, Jitter_{AND}),$$

where $Period_{AND} = Period_i$ and $Jitter_{AND} = \max_i\{Jitter_i\}$. ◇

OR-activation of event streams describes that a task is activated if on any of its input streams an event occurs. This leads to a superposition of all input event models. According to Jersak [40], OR-activation of such event models is defined as follows:

Definition 2.3.5 (OR-Activation - Periodic with Jitter) *OR-activation of n input event models described by the parameters $(Period_i, Jitter_i)$ with $i \in \{1, ..., n\}$ is defined as the event model*

$$(Period_{OR}, Jitter_{OR}), \text{ where}$$

- $Period_{OR} = \frac{1}{\sum_{i=1}^{n} \frac{1}{Period_i}}$
- *$Jitter_{OR}$ is the minimum jitter which fulfills the following equations:*

$$\left\lceil \frac{t + Jitter_{OR}}{Period_{OR}} \right\rceil \geq \sum_{i=1}^{n} \left\lceil \frac{t + Jitter_i}{Period_i} \right\rceil$$

$$\max\left(0, \left\lfloor \frac{t - Jitter_{OR}}{Period_{OR}} \right\rfloor\right) \leq \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - Jitter_i}{Period_i} \right\rfloor\right)$$

◇

In [40] it is shown that the minimum jitter values for the lower and upper curve are always identical. Furthermore, an approach is proposed to calculate such a minimum jitter by considering all constant segments of the upper curve for one macro period. The overall minimum jitter can then be determined as the smallest value which satisfies the equation for all constant segments.

2.4. Task Networks

A task network is a common formalism to model the software parts of a system in performance and scheduling analysis of real-time systems. It is a graph formalism consisting of nodes that are referred to as tasks or processes, and edges between tasks. Edges typically model control-flow dependencies and thus also precedences between tasks. Task activations are usually modeled using event streams, which have been introduced in Section 2.3.

While there exist different variants of task networks, we rely on task networks as they have been defined in [24]. There, a task network is defined as a graph consisting of trigger nodes where events are produced, sinks where events are consumed, task nodes representing processes, and edges between tasks representing control-flow dependencies. Edges are sometimes also referred to as *signals*. Furthermore, tasks are annotated with response times and execution times, respectively. They are defined as an interval of a minimum and maximum time bound. Please note, that for this definition, the event model from Def. 2.3.2 is used.

Definition 2.4.1 (Task Network [24]) *A task network $\mathcal{TN}$ is a tuple (Σ, C, E) where Σ is a set of event labels, $C = Q \cup T \cup S$, Q is a set of triggers, T is a set of tasks, S is a set of sinks, and $E \subseteq C \times \Sigma \times C$ is a set of edges. The components are defined as follows:*

- *A* trigger *$q \in Q$ is a tuple $(e, P^-, P^+, J, \pi_e^{out})$ where $e \in \Sigma$, and $P^-, P^+, J \in \mathbb{T}$. π_e^{out} is an event model, defined as $\mathcal{EM}(e, P^-, P^+, J)$.*
- *A* task *$\tau \in T$ is a tuple $(e, f, P^-, P^+, J, R^-, R^+, \pi_e^{in}, \pi_f^{out})$ where $e, f \in \Sigma$, and $P^-, P^+, J, R^-, R^+ \in \mathbb{T}$. The event model π_e^{in} is defined as $\mathcal{EM}(e, P^-, P^+, J)$ and π_f^{out} is defined as $\mathcal{EM}(f, P^-, P^+, J + R^+ - R^-)$.*
- *A* sink *$s \in S$ is a tuple $(e, P^-, P^+, J, \pi_e^{in})$ where $e \in \Sigma$, and $P^-, P^+, J \in \mathbb{T}$. The event model π_e^{in} is defined as $\mathcal{EM}(e, P^-, P^+, J)$.*

$\mathcal{TN}$ has to satisfy the following constraints:

- *(C, E) must be an acyclic graph.*
- *For each $q = (e, P^-, P^+, J, \pi_e^{out}) \in Q$ there is at least one edge $(q, f, c) \in E$ with $c \in T$.*
- *For each $\tau = (e, f, P^-, P^+, J, R^-, R^+, \pi_e^{in}, \pi_f^{out}) \in T$ there is a unique edge (c, e, τ) with $c \in Q \cup T$ and at least one edge $(\tau, g, d) \in E$ with $d \in T \cup S$.*
- *For each $s = (e, P^-, P^+, J, \pi_e^{in}) \in S$ there is a unique edge $(c, e, s) \in E$ with $c \in T$.*
- *Components connected by an edge $(c, e, d) \in E$ must be compatible, i.e., for $\pi_c^{out} = \mathcal{EM}(e', P_c^-, P_c^+, J_c)$, and $\pi_d^{in} = \mathcal{EM}(e, P_d^-, P_d^+, J_d)$, it must hold that $[P_c^-, P_c^+] \subseteq [P_d^-, P_d^+]$, and $J_c \leq J_d$.* ◇

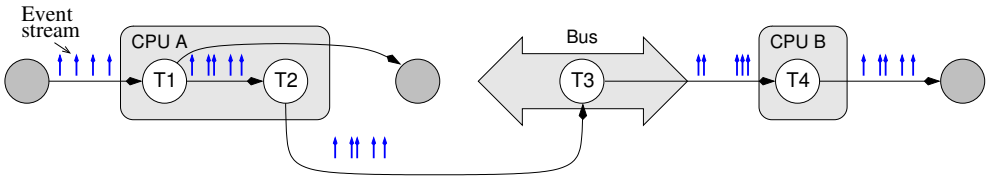


Figure 2.2.: Example for a Task Network (Source: [24])

For this work, we will always refer to worst case execution times (WCETs) when reasoning about task runtimes instead of response times. This is due to the fact that we derive task networks from specifications in Simulink, where no hardware resources are available and thus no interaction of tasks on a shared resource is considered.

In Figure 2.2, a task network is shown, where tasks have already been deployed to a processor (CPU) or a bus. Tasks are depicted as white-filled circles with a label such as $T1$ and edges between them are drawn as directed arcs. The gray-shaped circles model the interface to the environment in terms of triggers and sinks. Trigger model inputs, where events are produced with a specific event model and sinks represent outputs, where events are consumed.

2.5. MATLAB Simulink

Simulink is a high level system modeling tool, which allows the modeling and simulation of dynamic systems from a multitude of domains including continuous and discrete systems. It is based on *MATLAB*, which is a numerical computing environment. Simulink also offers a framework for code generation named *Embedded Coder*. MATLAB, Simulink and Embedded Coder are distributed by *MathWorks* [1]. Another code generator for Simulink is *TargetLink* distributed by *dSpace* [2].

In Simulink, systems are modeled in terms of block diagrams, where blocks represent specific functions and signals are used to connect output ports of one block with input ports of another block. Simulink offers a rich library of standard modeling blocks but also the possibility to create user-defined blocks such as so-called *S-Functions*. Block diagrams may be simulated to observe the system behavior. For simulation a solver needs to be chosen where different discrete and continuous solvers are available depending on the characteristics of the system. Timing characteristics of blocks are defined by *sample times*, which are defined by the user or by Simulink. Sample times can be also classified as either being discrete or continuous. Discrete sample times consist of a period *per* and an initial phase offset *init*. Thus, a block is executed in the simulation steps $init, init + per, init + 2 \cdot per, \ldots$. For simulation, continuous sample times are transformed into discrete sample times depending on the simulation step size, which is determined by the chosen Simulink solver.

Discrete systems are, for example, used to model a controller that should be implemented on a hardware platform whereas continuous systems are typically used to

[1] www.mathworks.com
[2] www.dspace.com

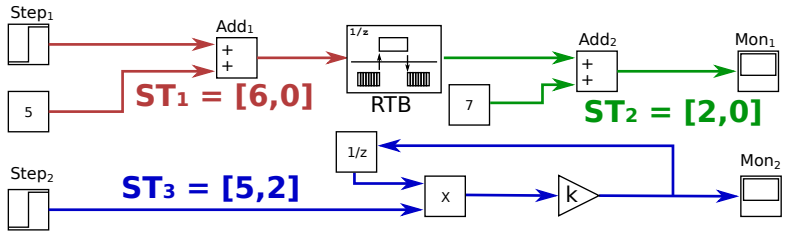


Figure 2.3.: Example for a Simulink Block Diagram

model the environment of the discrete system. For this work, we only consider discrete systems. For discrete systems and a discrete solver with a fixed step-size, code may be produced by using code generators such as Embedded Coder or TargetLink.

In Figure 2.3, an example of a Simulink block diagram is shown. This model contains three different sample times that are denoted as ST_1, ST_2, and ST_3. For example, sample time $ST3 = [5, 2]$ means that those blocks are executed first in simulation step 2 and afterwards periodically with a period of 5 leading to the steps 2,7,12,17,... . A set of connected blocks with the same sample time is referred to as a *synchronous set*. Assuming a fixed-step solver, blocks of different synchronous sets may only be connected by a signal if a *rate transition block* is inserted in between. This block guarantees a deterministic data transfer by holding the last value of a preceding producer block and offering it to a succeeding consumer block if needed. The concrete behavior depends on whether the producer or the consumer block has the larger period. If the consumer block runs with a smaller period (meaning a higher rate), it is executed in more steps than the producer block. Thus, the signal value of the producer block must be available also when this block is not executed. In the other case, where the producer block is executed more often, it is not necessary to store the value, but it must be guaranteed that the order of block executions is preserved when both blocks are executed in a simulation step.

Simulink also allows to model loops. For this work, we only allow models with so-called *non-algebraic loops* meaning that each loop contains a delay block as e.g. a *Unit Delay* block. In Figure 2.3, the part with sample time ST_3 contains a non-algebraic loop with a unit delay block denoted as $1/z$. We also allow that the behavior of a block may be modeled by finite state machines by using *Stateflow*. Stateflow is integrated in Simulink and offers an environment to model so-called *state charts*.

3. Function Networks

As formal base for the Simulink translation and task creation, an extended task network formalism called *function network* is introduced. To be able to show semantics preservation, formal semantics of function networks is defined and properties about causality and timing are derived that should be preserved when translating a Simulink model and performing merging operations. As a necessary condition to implement a system, also boundedness of function networks is discussed and a class of function networks is defined where boundedness is decidable.

A function network is a graph that consists of nodes called *function nodes*, edges called *channels* and trigger nodes called *event sources*, and is an extension of the task network formalism presented in Section 2.4. In a task network, nodes represent tasks that are pieces of software in terms of executable code, which should be executed on a hardware processor. Tasks may have several properties, where from the real-time point of view the most interesting are the worst case execution time (WCET), the activation pattern describing when the task is triggered for an execution, and the (hard) deadline when this task has to finish its execution. Furthermore, there may exist end-to-end deadlines giving timing constraints over task chains. Deadlines and activation patterns are properties that originate from the system specification, which is given as a Simulink model for this work. Contrastingly, the worst case execution time depends on the software code of the task and the target processor where the task is executed. In the scenario we consider here, there exists a set of target processors and thus the WCET can be determined for each target processor, for example, by using aiT [30]. Thus, we get one WCET for each task and each target processor.

The need to extend the model of task networks has several reasons leading to different kind of extensions. First, it is often not sufficient to assume that a task is triggered by a single channel or trigger. In practice, we may have, for example, *synchronous* activations, which means that a task is only activated if on all its incoming edges a trigger event has occurred. This is typically the case in synchronous languages, such as synchronous block diagrams used in Simulink. Please note, that for synchronization, we always need a buffer to store events until all synchronization partners have arrived. Furthermore, there may be also overlapping activations, denoted as *superposition* meaning that a task may either be activated by a trigger event a or a trigger event b. We will need this kind of activation also when translating Simulink block diagrams to model rate-transitions between blocks with different sample times. A further need to represent superposition of task activations is the more general case, where a task is part of different task chains that are executed independently from each other.

In function networks, we introduce *input ports*, which represent activation points of function nodes and may have several incoming channels. To activate an input port on each channel an event is needed realizing a synchronization of these events. Super-

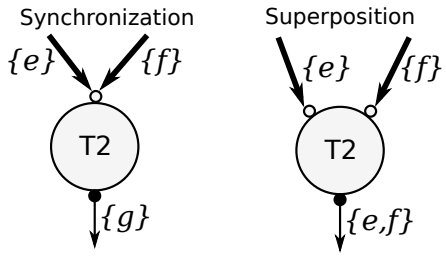


Figure 3.1.: Task Activation by Synchronization (AND) and Superposition (OR)

position may be modeled by a function node with multiple input ports, where each input port activation leads to an activation of the node. This kind of activations can already be found in existing task models as e.g. in [68]. To describe event occurrences at ports in a function network, we introduce *event patterns*, which are an extension of classic periodic event models [40].

In Figure 3.1 on the left, an example for a synchronization is depicted. Task $T1$ receives events from two incoming channels that lead to the same input port (depicted as small circle). $T1$ is activated whenever an event e was received on the left channel *and* an event f was received on the right channel leading to an event g at the output port. On the right of Figure 3.1, an example for a superposition is shown. Task $T2$ is activated if an event e on the left channel *or* an event f at the right channel is received. The received event is forwarded to the output port.

Another important reason to extend task networks is the fact that the WCET of a task may depend on the input data it processes and on the internal state of the task, which may change due to previous executions. In classic task networks, the WCET of a task is a safe upper bound for all possible kinds of input data and internal state configurations. This often leads to over-approximations in timing analysis and thus possibly to a negative result in terms of a deadline violation although the assumed WCET will never be reached under specific input and state conditions. There already exist analysis approaches to cover these issues such as state-based scheduling [56].

As an example, we could think of a simple adaptive cruise control (ACC) system equipped with speed sensors ϕ_1 and ϕ_2 and a distance sensor ϕ_3, which measures the distance to preceding cars as depicted in Figure 3.2. The sensor values are aggregated by the tasks f_v (speed sensors) and f_d (distance sensor). The resulting values are forwarded to a controller task f_c, which decides whether it is necessary to initiate a braking action to avoid a collision. As long as the distance is not critical, denoted by the event *ok*, the task forwards the values to nodes of the car periphery via task f_e. This only needs a comparatively small execution time. If the distance to the vehicle ahead becomes critical, denoted by the event *crit*, a braking measure is initiated by an event b triggering task f_b. This deceleration process must be controlled leading to a mode change of the controller task involving a larger execution time. We will consider this example in more detail in Section 3.2.

If we were able to represent this kind of information in a task network model, this may reduce over-approximations in timing analysis and thus improve the quality of

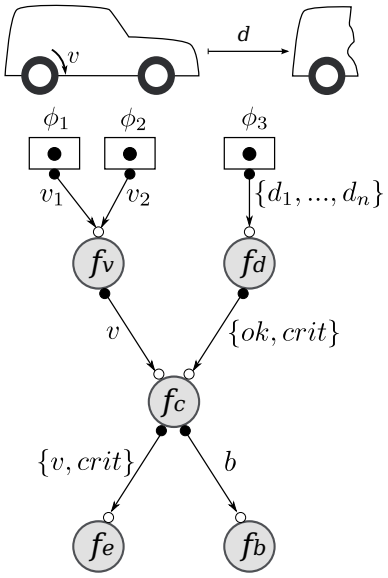


Figure 3.2.: Simple Example of an Adaptive Cruise Control (ACC) System

the results. Thus, in design space exploration, task allocations may become feasible that would be rejected with classic task networks. This may lead to more possible solutions for task allocations and hence also to lower modification costs.

Accordingly, we extend the model of a task in a function network by introducing an internal state transition system that is sensitive to incoming data in terms of events and the internal state of the task. Each transition is annotated with a delay, which represents the worst case execution time for a specific input and state configuration assumed by the considered transition. Another extension is to allow transitions that produce events at different output ports with different delays leading to a set of delays per transition. Furthermore, we may observe different events on the same channel leading to different behavior of the function node. If we consider again the example of the ACC system, there may be two different events sent on the same channel to distinguish between sensor values below and above a certain threshold value denoting a critical distance.

Another important reason to introduce internal state transition systems in task nodes arises from one of the essentials of this work: the merging of nodes. If nodes are merged to build a task, we need to deal with the question how this may be represented in the task network formalism. In doing so, we need to assure that we do not violate the semantics of the model in terms of causality of task executions and events. Thus, if we want to reason about what merging of nodes semantically means, we need to have some abstract representation of the behavior of the task. In Figure 3.3 on the left, an extract of a task network with two sequential tasks T_1 and T_2 is depicted where T_1 has one outgoing channel triggering T_2, and T_2 has no other incoming channels. For this simple case, it would be sufficient to say that merging T_1 and T_2 leads to a task

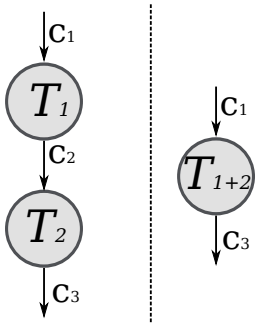


Figure 3.3.: Merging Example for a Simple Task Chain

T_{1+2} that has all incoming channels of T_1 and whose WCET is the sum of the single WCETs of T_1 and T_2. The result is shown in Figure 3.3 on the right. But as soon as T_1 has more than one outgoing channel, or T_2 has more than one incoming channel, we get into difficulties to represent the behavior correctly as illustrated in Figure 3.4. If we would perform the merging in the same manner as for the previous case, the events on outgoing channels of T_1 not leading to T_2 would automatically be delayed, because the WCET of T_{1+2} is the sum of the single WCETs. But actually, the output of T_1 would be available already after the execution of the program code that belongs to T_1. Thus, we need the possibility to annotate different WCETs for different outputs of a task to represent delays correctly.

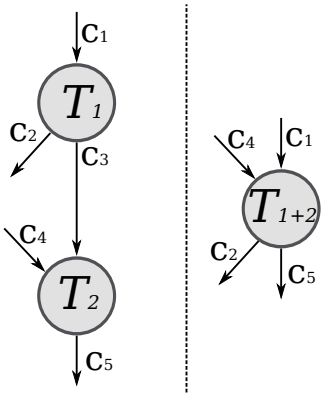


Figure 3.4.: Merging Example for a Complex Task Chain

If T_2 has more incoming channels than the one from T_1, as c_4 in Figure 3.4, these channels would also be input channels of T_{1+2}. This means that an execution of T_{1+2} can only start if on each input channel an event is available. But actually, the code that belongs to T_1 does not depend on the input channels of T_2 and thus could be executed without c_4. Hence, we obviously change the activation dependencies and delays implicitly by doing such an operation. Thus, we cannot represent and keep the

previous semantics after the merging operation due to missing expressiveness in the formalism. The problem becomes even worse if we consider more complex activation patterns with several activation ports. An internal state transition system within a node offers the possibility to express the original semantics without doing implicit changes to activation dependencies or delays.

The third major extension of function networks over task networks is the definition of *data nodes*. In principle, the concept of data nodes is an extension of signals, which we already know from some task network formalisms. Signals are special tasks modeling communication between ordinary computational tasks. This means that signals do not add semantic expressiveness but are an explicit entity to model communication. We extend this approach because in practice there may exist further concepts beside signals to model communication in a specification. For example, in Simulink, it is possible to define *data store memory blocks*. Other Simulink blocks may access those blocks without inducing a partial order on read or write operations. Thus, if different blocks share a data store memory block, Simulink does not make any guarantees when and in which order data is written into or read from a block. This is due to the fact that in Simulink blocks do not need any time to execute, and thus it cannot be determined if the reading or writing block is executed first. This kind of data store blocks cannot be modeled explicitly in classic task networks. First, there is no node type that thought to represent a data store. Second, edges always induce a partial order between nodes, which would not meet the semantics of the Simulink model.

This leads to the introduction of a further element of function networks that is a special channel type called *read channel*. Read channels allow to read from a data node when the reader node is executed without having any activation dependencies from the writing nodes. We define different types of data nodes: A *shared* data node stores an abstract value in terms of the last event that has been written, and allows other tasks to read this value. A value may be consumed infinitely often. This is the counterpart to a data store memory block in Simulink. In a *FIFO* data node, event values are read in the same order as they were written and each value may be only consumed once. Another data node type is *signal*, whose semantics and role is identical to the signals known from task networks i.e., they forward incoming activations from tasks to one or more target tasks with a specific delay. We define a further data node type called *finite source*, which is used to model synchronization cycles in function networks. A finite source provides an output event at system startup, but for each further output event it needs an input event to occur before i.e., the previous activation must have reached the finite source before the next activation may start. Its semantics is very similar to those of pre-allocated events in task networks with cyclic dependencies [41].

Although data nodes are only special function nodes, they are important and essential for system modeling and analysis because they have a different role. While function nodes represent computing processes in terms of software tasks that are assigned to hardware processors to be executed, data nodes represent communication processes. For example, *signals* are mapped to a bus in a distributed system to model the message transmission on the bus. Thus, the delay of a signal represents the time that is needed to transfer the data from one processor to the other via a bus, and not the execution time of a piece of code on a processor. Data nodes like *shared* or *FIFO*

can be regarded as dedicated processes that handle the memory access to a shared memory located on a processor chip.

Due to the expressive power of function networks, the question arises whether or for which kind of networks boundedness is decidable. A function network is bounded if it is implementable with a finite amount of resources. This is in particular relevant when a system should be executed on real hardware as it is the case for this work. Thus, we will answer this question for a specific class of function networks, where we use event pattern propagation to decide boundedness.

Outline We start with defining event patterns in Section 3.1 as an event model to describe event occurrences in function networks. We define operations on event patterns to realize superposition and synchronization, and define essential properties to be able to decide boundedness.

In Section 3.2, syntax of function networks is defined. First, we define a basic function network consisting of function nodes, event sources and channels. In a second step, extended function networks are defined containing data nodes and read channels and a translation into basic function networks is given.

Semantics of basic function networks is defined in Section 3.3 in terms of a composition of timed automata for each basic function network component. In particular, buffers are defined as a part of a function node to store activation events. Furthermore, we define properties to describe the behavior of each component in terms of causality and timing delays. These properties are used to show semantics preservation for the translation from Simulink and merging of nodes during task creation. Semantics of extended function networks is defined implicitly by the previously given translation. Causal dependencies including timing delays are derived and shown for basic and extended function network components as well.

Based on the defined semantics, we prove the decidability of boundedness for a specific class of function networks in Section 3.4. To be able to do this, we show how event patterns can be propagated to determine sufficient and finite sizes for buffers in a function network. Finally, we summarize this chapter in Section 3.5 and discuss related work for function networks.

3.1. Event Patterns

Event patterns are an extension of periodic event models as introduced in Section 2.3. An event model defines a class of event streams, which describe the occurrence of events at particular observation points in a model. For function networks these observation points are input and output ports. Event models are described by four characteristic functions that are the η^-, η^+, δ^- and δ^+ functions. For this work, we will mainly consider the η^- and η^+ functions, which are sufficient to describe event streams because the respective δ^- and δ^+ functions can be derived from them [73].

As already stated in Section 2.3, there exist some variants of event models in the literature as periodic streams with jitter, streams with a minimum and a maximum period and initial start values for the η^- and η^+ functions. But there is currently no

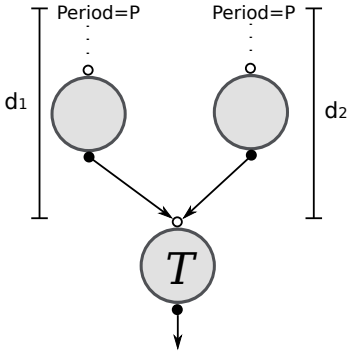


Figure 3.5.: Motivation for Offset in Event Models

event model available that considers all these aspects and thus we define *event patterns* to be able to cover the common event models.

3.1.1. Definition of Event Patterns

We define an event pattern as an event model having as parameters a lower and upper period bound $P^-, P^+ \in \mathbb{R}^+$, a jitter $J \in \mathbb{R}_0^+$, and an initial offset $O \in \mathbb{R}_0^+$. We do not consider sporadic streams for this work where P^+ would be allowed to be set to ∞.

Period and jitter are well established and known in particular from the periodic event model of Richter and Jersak [66, 40]. The introduction of a lower and upper period bound has been done in [24] and also in [47]. In the latter, also initial values for the η^- and η^+ functions are defined where the initial value for the η^+ function determines the maximum number of events that may occur simultaneously, which is known as burst. The initial value for the η^- function is smaller or equal to zero and thus allows that the occurrence of an event may be delayed more than the sum of period and jitter, which moves the η^- curve to the right.

In our event pattern model, the initial burst value of the η^+ function is modeled in the same way as done by Richter and Jersak by increasing the jitter. The initial value for of the η^- function is represented by an offset O. This offset is able to represent a phase shift, which means that the occurrence of an event may be delayed more than the period and jitter allow. Such a phase shift occurs, for example, in Simulink models where an offset, denoted as *initial phase*, is part of the sample time of a block. Another case where an offset is useful and also necessary is the propagation of event models between connected nodes of a task network.

This is shown in Figure 3.5, where a task T is depicted with one input port and two incoming channels from two different task chains. T has to wait with its execution until it has received both an event from the left and the right task chain. The delays of these task chains are denoted as d_1 and d_2 and are determined by the execution times of their respective tasks. Both task chains are activated with the same period P and

no jitter. For simplicity, we assume fixed execution times and thus the jitter stays zero for all succeeding tasks of each chain. Hence, in an event model without offset, the delays d_1 and d_2 are not considered at all to determine the event stream at the input port of task T. But if d_1 and d_2 differ, events of the one chain arrive at task T earlier than events of the other chain. If this difference becomes greater than the minimum inter-arrival time of events on any input stream, we would need an additional place in the input buffer of T. Thus, an offset is needed to determine a safe capacity for the activation buffer of a task to avoid buffer overrun. We define event patterns as follows:

Definition 3.1.1 (Event Pattern) *An event pattern is a tuple* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *with a set of event symbols* Σ^{EP}*, a minimum and maximum period* $P^-, P^+ \in \mathbb{R}^+$ *with* $P^+ \geq P^-$*, an initial offset* $O \in \mathbb{R}_0^+$ *and a jitter* $J \in \mathbb{R}_0^+$ *defining two monotonic increasing functions called* Eta-functions $\eta^{+/-} : \mathbb{R}_0^+ \to \mathbb{N}_0$ *as follows:*

$$\eta^+(t) = 1 + \left\lfloor \frac{t+J}{P^-} \right\rfloor,$$
$$\eta^-(t) = \max\left(0, \left\lfloor \frac{t-O-J}{P^+} \right\rfloor\right)$$

$\eta^+(t)$ *and* $\eta^-(t)$ *are called upper and lower Eta-function respectively. The timed language* $L(EP)$ *of an event pattern* EP *is defined as follows:*

$$\begin{aligned} L(EP) = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \\ & \mid\ \sigma_i \in \Sigma^{EP}, \\ & \quad \eta^-(t_i) \leq i \leq \eta^+(t_i), \\ & \quad \forall m : \eta^-(t_{i+m} - t_i) \leq m+1 \leq \eta^+(t_{i+m} - t_i) \\ & \} \end{aligned}$$

where $i, m \in \mathbb{N}^+$.

$\diamond$

The respective η^- and η^+ functions are depicted in Figure 3.6 where $\eta^{+/-}(t) = n$ means that within a time interval of t, n events from the set Σ^{EP} can be observed. We write $EP(\Sigma^{EP})$ for the event pattern over the event set Σ^{EP}. If for an event pattern $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ holds that $P^+ = P^-$, then it is called *periodic*.

The characteristic functions of event patterns can be represented in an equivalent form, which will be used for proofs that reason about the language of event patterns.

Lemma 3.1.1 (Equivalent Representation of Event Patterns) *Let* EP *be an event pattern with* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$*. An equivalent representation of this event pattern is the following:*

$$\eta^+(t) = \begin{cases} \left\lfloor \frac{J}{P^-} \right\rfloor + 1 & , \text{if } t \in [0,\ P^- - (J \bmod P^-)) \\ n+1 & , \text{if } t \in [n \cdot P^- - J, (n+1) \cdot P^- - J),\ n > \left\lfloor \frac{J}{P^-} \right\rfloor \end{cases}$$

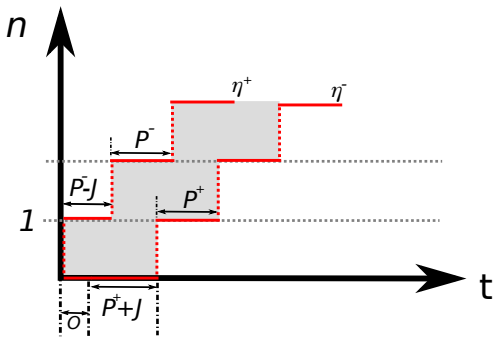


Figure 3.6.: η^+ and η^- Functions of Event Patterns

$$\eta^-(t) = \begin{cases} 0 & ,if\ t\ \in [0, O + P^+ + J) \\ n & ,if\ t\ \in [O + n \cdot P^+ + J,\ O + (n+1) \cdot P^+ + J),\ n > 0. \end{cases}$$

Proof: see Lemma A.1.1 in the appendix on page 201. □

As already mentioned before, the η^- and η^+ functions are only two of the four characteristic functions to define event streams. Based on [73] delta functions for event patterns are defined as follows:

Definition 3.1.2 (Delta Functions for Event Patterns) *Let EP be an event pattern with $EP = (\Sigma^{EP}, P^-, P^+, J, O)$. The δ^+ and δ^- functions are defined as follows:*

$$\delta^-(n) = \max(0, (n-1) \cdot P^- - J)$$
$$\delta^+(n) = O + (n-1) \cdot P^+ + J$$

◇

To be able to show that the language of a specific function network component may be correctly abstracted by an event pattern, we show the timed language of event patterns based on the previous alternative event stream representation.

Lemma 3.1.2 (Event Pattern Language) *Let $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ be an event pattern. Then the language is defined as follows:*

$$\begin{aligned} L(EP) = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma^{EP}, \\ & (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\ \}\ & \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

Proof: see Lemma A.1.2 in the appendix on page 203. □

As a next step, we define a translation from event patterns to the periodic event model with jitter of Jersak and show that the result is a correct abstraction. This enables us to use all operations of their work, such as synchronization and superposition, even if the abstraction leads to a loss of precision. This may lead to over-approximations when determining sufficient buffer sizes.

Please note, that in the following translation, we always have to add a small value greater zero to the jitter to get a correct representation in the event model of Jersak. This is due to the fact that we defined the η^+ function similar to the one of Thiele et al. [47], which slightly differs from the model of Jersak. This concerns the time instance where the η^+ function reaches its next step. In our definition a time instance of $t = (n-1) \cdot P - J$ leads to $\eta^+ = n$ while in the definition of Jersak this first holds for $t = (n-1) \cdot P - J + \epsilon$ where $\epsilon > 0$ is a small number greater than zero. This is needed to consider the fact that in the Jersak model the η^+ is defined using the *ceiling* function instead of the *floor* function.

Definition 3.1.3 (Translation to Jersak Model) *Let $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ be a periodic event pattern i.e. $P^- = P^+$ and let $\epsilon > 0$. It is translated to the periodic event model with jitter from Def. 2.3.1 as follows:*

- $Period = P^- = P^+$
- $Jitter = J + O + \epsilon$

$\diamond$

In Lemma 3.1.3 we show that the translation into the periodic event model of Jersak is valid because the resulting η-curves are a valid abstraction of the original ones.

Lemma 3.1.3 (Valid Translation to Jersak Model) *Let $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ be a periodic event pattern with $P^- = P^+$ and its Eta-curves η^+ and η^-. The translation from Def. 3.1.3 is a valid abstraction i.e., the resulting η-curves $\eta^-(t)_{P+J}$ and $\eta^+(t)_{P+J}$ contain all streams that the event pattern contains i.e.*

$$\eta^+(t)_{P+J} \geq \eta^+(t) \ \wedge \ \eta^-(t)_{P+J} \leq \eta^-(t)$$

Proof: see Lemma A.1.3 in the appendix on page 204. $\square$

3.1.2. Properties and Operations

We will define and prove some properties and operations on event patterns that we will need later for event pattern propagation and to show decidability of boundedness. We will use event patterns to determine sufficient buffer sizes for a specific class of function networks and thus need to be able to reason about maximum distances between Eta-curves of different input streams.

We start by defining a simple relation between Eta-functions that holds if the value of the one function is always less or equal to the values of the other function.

Definition 3.1.4 (Relation on Functions) *Let η_1, η_2 be two Eta-functions*

$$\eta_1 \leq \eta_2 \Leftrightarrow \forall t \in \mathbb{R}_0^+ : \eta_1(t) \leq \eta_2(t).$$

$\diamond$

An important property for streams to show boundedness is that their periods are equal, which is defined as follows:

Definition 3.1.5 (Period Equivalence) *Let $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ be two event patterns. EP_1 and EP_2 are called* period-equivalent *with a period of P written as $EP_1 \overset{P}{=} EP_2$ if the upper and lower period bounds are equal i.e.*

$$EP_1 \overset{P}{=} EP_2 \iff P = P_1^- = P_1^+ = P_2^- = P_2^+$$

$\diamond$

To be able to show boundedness, it is essential to have upper bounds for the maximum time distance between events described by two Eta-functions. Thus, we define the maximum time distance as follows:

Definition 3.1.6 (Maximum Time Distance) *Let η_1 and η_2 be two Eta-functions. The maximum time distance δ_t of events of η_1 and η_2 is given by the following:*

$$\delta_t(\eta_1, \eta_2) = \sup_{t_1, t_2 \in \mathbb{R}_0^+} (\{|t_2 - t_1| \mid \eta_1(t_1) = \eta_2(t_2)\})$$

$\diamond$

Please note that this function is very similar to the delay function from Jersak [40].

The next lemma shows that the maximum time distance between an upper Eta-curve of an event pattern EP_1 and the lower Eta-curve of a period-equivalent event pattern EP_2 is always bounded by a value determined by the event pattern parameters.

Lemma 3.1.4 (Time Distance is Bounded) *Let η_1^+, η_2^- be Eta-functions of two period-equivalent event patterns $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ where $EP_1 \overset{P}{=} EP_2$. Then the following holds:*

$$\delta_t(\eta_1^+, \eta_2^-) \leq O_2 + 2 \cdot P + J_2 + J_1$$

Proof: see Lemma A.1.4 in the appendix on page 205. $\square$

The notion of time distance can be extended to event patterns by considering the maximum of both combinations of lower and upper Eta-curves of the two patterns.

Definition 3.1.7 (Maximum Event Pattern Distance) *Let EP_1 and EP_2 be two period-equivalent event patterns i.e. $EP_1 \overset{P}{=} EP_2$. Their maximum time distance is defined as follows:*

$$\delta_t(EP_1, EP_2) = \sup(\delta_t(\eta_1^+, \eta_2^-), \delta_t(\eta_2^+, \eta_1^-))$$

$\diamond$

Now we can show that the time distance between period-equivalent event patterns is determined by the offset, period and jitter of both streams in the following manner.

Lemma 3.1.5 (Event Pattern Distance is Bounded) *Let EP_1 and EP_2 be two period-equivalent event patterns with $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ where $EP_1 \overset{P}{=} EP_2$. Then the following holds.*

$$\delta_t(EP_1, EP_2) \leq \max(O_1, O_2) + 2 \cdot P + J_1 + J_2$$

Proof:

$$\begin{aligned} \delta_t(EP_1, EP_2) &\overset{Def.\ 3.1.7}{=} \sup(\delta_t(\eta_1^+, \eta_2^-), \delta_t(\eta_2^+, \eta_1^-)) \\ &\overset{Lemma\ 3.1.4}{\leq} \sup(O_2 + 2 \cdot P + J_2 + J_1,\ O_1 + 2 \cdot P + J_1 + J_2) \\ &= \max(O_1, O_2) + 2 \cdot P + J_1 + J_2 \end{aligned}$$

because $O_1, O_2, P, J_1, J_2 < \infty$. □

Thus, the time distance between events of two period-equivalent streams is always bounded. We will now prove that in case of two periodic streams that are not period-equivalent, there does not exist an upper bound for the time distance as it always grows with increasing time.

Lemma 3.1.6 (Infinite Distance) *Let η_1^+, η_2^- be Eta-functions of two periodic event patterns $EP_1 = (\Sigma_1^{EP}, P_1, P_1, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2, P_2, J_2, O_2)$ that are not period-equivalent i.e., $P_1 = \frac{q}{r} \cdot P_2$ where $q, r \in \mathbb{R}^+$ and $q \neq r$. The time distance until the same number of events is produced grows with every periodic step and is thus not bounded i.e. $\delta_t(\eta_1^+, \eta_2^-) = \infty$ leading with Def. 3.1.6 to*

$$\lim_{t_1, t_2 \to \infty} \left(\sup_{t_1, t_2 \in \mathbb{R}_0^+} (|t_2 - t_1| \ \mid \eta_1^+(t_1) = \eta_2^-(t_2)) \right) = \infty$$

Proof: see Lemma A.1.5 in the appendix on page 206. □

As a next step, we will define some typical operations on event patterns, which have already been defined in [40] for periodic streams with jitter but without an offset. As we have already shown before, we can translate our model to the model of Jersak at the price of losing precision. To avoid such over-approximations, we show how these definitions may be easily extended to also handle event patterns with offsets. The precision gain lays here in the fact that we do not add the offset to the jitter as we do when translating to the model of Jersak. If we increase the jitter, this influences also the η^+ function (by moving it to the left) while the offset only affects the η^- function. Thus, the η^+ function of our extended synchronization definition is more precise than the translation to the model of Jersak. Furthermore, the definition of

these operations for event patterns enables us to propagate event pattern through a function network without leaving the formalism of event patterns. Otherwise, we always had to switch between different event models and with each translation step we might loose information and precision.

The first operation we define, is event pattern synchronization. Here, on the synchronized stream we first see an event if we have seen an event on each of the streams to be synchronized. This occurs in function networks at input ports with more than one incoming channel. A sufficient condition for event patterns to be synchronized is that they are period-equivalent. We define the synchronization of n period-equivalent event patterns as follows:

Definition 3.1.8 (Event Pattern Synchronization) *Let $EP_1 ... EP_n$ be n period-equivalent event patterns with $EP_i = (\Sigma_i^{EP}, P_i^-, P_i^+, J_i, O_i)$ where $\forall i, j \in \{1, ..., n\} : EP_i \overset{P}{=} EP_j$. The event pattern resulting from their synchronization is defined as follows:*

$$\begin{aligned} sync(EP_1, ..., EP_n) &= ((\Sigma_1^{EP} \times ... \times \Sigma_n^{EP}), P, P, J_{max}, O_{max}) \\ where\ J_{max} &= \max(J_1, ..., J_n), \\ O_{max} &= \max(O_1, ..., O_n) \end{aligned}$$

$\diamond$

In Lemma 3.1.7, we show that the synchronization operation leads to a correct abstraction i.e., all input languages of synchronized event streams are contained in the resulting synchronization language.

Lemma 3.1.7 (Synchronization is Correct Abstraction) *Let $EP_1 ... EP_n$ be n period-equivalent event patterns with $EP_i = (\Sigma_i^{EP}, P_i^-, P_i^+, J_i, O_i)$ and the respective eta curves $\eta_i^{-/+}$ where $\forall i, j \in \{1, ..., n\} : EP_i \overset{P}{=} EP_j$. Let $EP_s = sync(EP_1, ..., EP_n)$ be the synchronization of these event patterns with the eta curves $\eta_s^{-/+}$. Then the following holds:*

$$\forall i \in \{1, ..., n\}, t \in \mathbb{R}_0^+ : \ \eta_s^-(t) \leq \eta_i^-(t) \ \wedge \ \eta_s^+(t) \geq \eta_i^+(t)$$

Proof: see Lemma A.1.6 in the appendix on page 207. $\square$

Another relevant operation is the superposition of event streams, which also has been defined for streams with period and jitter by Jersak. Superposition occurs in the function network when a function node is activated by more than one input port. Superposition of Eta-curves is defined as the sum of the single Eta-curves. To find a valid abstraction of this sum in terms of a single event pattern, we make use of the definition of Jersak and define how this definition is extended for event patterns.

First, the period bounds can be each determined in the same way as for pure periodic streams. The same holds for the jitter. The offset only affects the η^- curve and moves it to the right and thus allows an event to occur more delayed than just by period and jitter. From the work of Jersak we know how superposition of two streams without

offset is defined. If we now add an offset to all streams that are to be superposed, we move the η^- function of the superposition by the maximum offset to the right. This leads to the following definition of superposition for event patterns.

Definition 3.1.9 (Event Pattern Superposition) *Let $EP_1 ... EP_n$ be n event patterns with $EP_i = (\Sigma_i^{EP}, P_i^-, P_i^+, J_i, O_i)$. The event pattern resulting from their superposition is defined as follows:*

$$super(EP_1, ..., EP_n) = (\Sigma_1^{EP} \cup ... \cup \Sigma_n^{EP}, P_s^-, P_s^+, J_s, O_s)$$

where

- $P_s^- = \frac{1}{\sum_{i=1}^{n} \frac{1}{P_i^-}}$, $P_s^+ = \frac{1}{\sum_{i=1}^{n} \frac{1}{P_i^+}}$
- J_s *is determined as defined in Def. 2.3.5.*
- $O_s = \max_{i=1}^{n}(O_i)$ ◇

In Lemma 3.1.8, we show that the previous superposition definition with offset is a valid abstraction of the sum of the single Eta-curves. We assume for this lemma only two streams to be superposed while the result can be easily generalized to n streams because the sum and also the maximum operation are commutative. Furthermore, we ignore the jitter here by setting the jitter of all streams to zero. How a valid abstraction for the superposition jitter can be determined has already been shown by Jersak [40]. The results still hold for η^- curves with an offset because both parameters independently move the η^- curve to the right. The η^+ curves are not affected by the offset at all. Thus, we only need to show that the abstraction is valid for the η^- curves.

Lemma 3.1.8 (Superposition is Valid Abstraction) *Let EP_1, EP_2 be two event patterns with $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ where $J_1 = J_2 = 0$ and the respective lower Eta-curves η_1^- and η_2^-. Let further be $EP_s = super(EP_1, EP_2)$ be the superposition of both event patterns with the lower Eta-curve η_s^-. Then the superposition offset is a valid abstraction because the following holds:*

$$\forall t \in \mathbb{R}_0^+ : \eta_s^-(t) \leq \eta_1^-(t) + \eta_2^-(t)$$

Proof: see Lemma A.1.7 in the appendix on page 207. □

For event pattern propagation, we further need a function that transforms an input event pattern of a function node into an output event pattern by considering the delay interval of the function node execution. Thus, we define an event pattern delay function where the minimum execution time (transition delay) is added to the offset because the first event is at least delayed by this value. The jitter is increased by the difference between the maximum and minimum execution time as it is also defined in the model of Jersak [40].

Definition 3.1.10 (Event Pattern Delay) *Let* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *be an event pattern and* $[min, max]$ *a delay interval with* $min, max \in \mathbb{N}_0$ *and* $max \geq min$. *The delay function for event pattern is defined as follows:*

$$delay(EP, [min, max]) = (E, P^-, P^+, J', O')$$
$$where\ J' = J + max - min,$$
$$O' = O + min$$

◇

Now it remains to show that the language of an event set Σ^{EP} where each event is delayed by a specific time interval $[min, max]$ can be correctly abstracted by applying the delay function on the respective event pattern $EP(E)$.

Lemma 3.1.9 (Correct Event Pattern for Delayed Language) *Let* Σ^{EP} *be a set of events with an event pattern* $EP(\Sigma^{EP}) = (\Sigma^{EP}, P^-, P^+, J, O)$. *If each event* $e \in \Sigma^{EP}$ *is delayed by a time interval of* $[min, max]$ *with* $min, max \in \mathbb{N}_0 \times \mathbb{N}_0$ *and* $min \leq max$, *then the resulting language* $L(\Sigma^{EP})'$ *can be abstracted by applying the delay function. Following Def. 3.1.1, the delayed language* $L(\Sigma^{EP})'$ *is defined as*

$$\begin{aligned} L(\Sigma^{EP})' = \{\ & (\sigma_1, t_1 + [min, max])....(\sigma_i, t_i + [min, max])... \\ & (\sigma_{i+m}, t_{i+m} + [min, max])...\ |\ \ \sigma_i \in \Sigma^{EP}, \\ & (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\ \}\ \ & where\ i, m \in \mathbb{N}^+ \end{aligned}$$

Then it holds

$$L(\Sigma^{EP})' \subseteq L(delay(EP(\Sigma^{EP}), [min, max]))$$

Proof: See Lemma A.1.8 in the appendix on page 208. □

When propagating an event pattern from one port to another also the events that are described by this event pattern may change and thus we need a renaming function, which is defined as follows:

Definition 3.1.11 (Event Pattern Renaming) *Let* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *be an event pattern and* Σ^{EP}_{new} *a set of event symbols. The renaming function for event patterns is defined as follows:*

$$ren(EP, \Sigma^{EP}_{new}) = (\Sigma^{EP}_{new}, P^-, P^+, J, O)$$

◇

3.2. Function Network Definition and Properties

The formalism of function networks was published first in [16] while the syntactic definition evolved since that time to mainly improve readability. Nevertheless, the basic structure remained the same while it turned out to be useful to define a *basic function network* to show certain properties such as boundedness as already done in [13]. This basic function network contains the minimum set of elements that is needed to model the intended semantics, which simplifies many proofs. Afterwards, the definition of an extended function network follows and a translation from extended to basic function networks is given.

3.2.1. Basic Function Networks

A *basic function network* is defined as a directed graph where function nodes represent processing elements, or tasks, and edges represent (basic) channels transmitting events between nodes. In Figure 3.7 an example for a basic function network is depicted modeling a simple adaptive cruise control system where the velocity v of a car and the distance d to a preceding car is measured and used to make different controller decisions. Function nodes are depicted as circles and channels as directed arcs. A *channel* may transport different events where, for example, channel c_3 in Figure 3.7 transmits events from the set $\{d_1, ..., d_n\}$. Additionally, *event sources* allow to model events sent by the environment to the network. They are depicted as rectangles with filled circles where in Figure 3.7 ϕ_3 represents a distance sensor delivering values $d_1, ..., d_n$ and ϕ_1 and ϕ_2 are both speed sensors. The production of events is defined in terms of event patterns defined as a tuple $(\Sigma^{EP}, P^-, P^+, J, O)$ where Σ^{EP} is the set of event symbols the source node produces.

The connection between nodes and channels is realized by ports representing the observation points in the system. Ports describe which events flow into and out of the corresponding nodes. Activation of nodes is captured by their input ports depicted as small white circles. An input port activates a node whenever at least one event is available at each of its incoming channels. Each event can only release one activation i.e., events are consumed by an activation. For example, node f_v in Figure 3.7 is activated when both an event v_1 on channel c_1 and v_2 on channel c_2 occurs meaning that values are available from both speed sensors. We call this kind of activation AND-activation or *synchronization*. A function node having multiple input ports is activated whenever any of its input ports is activated. Node f_c, for example, is activated for each event on channel c_4 and for each event on channel c_5. We refer to this kind of activation as OR-activation or *superposition*. The combination of multiple input ports and input channels allows the modeling of complex node activations.

Function nodes are sensitive to incoming events. To this end, nodes employ internal state-transition systems. Depending on the current state and the received events, output events are emitted at output ports depicted as small black circles. Each activation causes a delay for processing, depending on the input event, the current state, and the particular output port. Delays are taken from intervals with best-case and worst-case bounds. Thus, a transition $t = (p, E, s \to \Psi, s')$ with a set of so-called *output speci-*

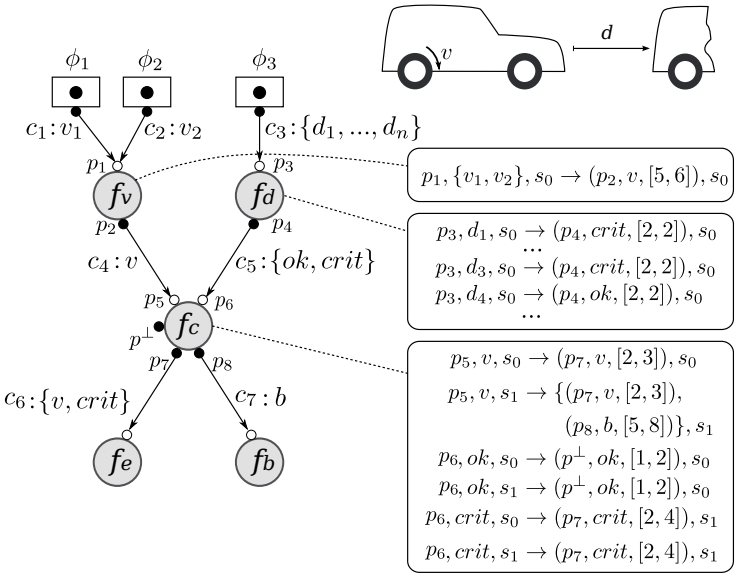


Figure 3.7.: Simple Function Network Model of an Adaptive Cruise Control System

fications $\psi = (p'_i, e'_i, [\delta_i^-, \delta_i^+]) \in \Psi$ means, that if a set of events $E = \{\sigma_1, ..., \sigma_n\}$ has occurred at input port p and the function node is in state s, an event e' is produced at output port p'_i with a delay of $\delta_i = [\delta_i^-, \delta_i^+]$ time units and the function node state is changed to s'.

For example, if node f_c in Figure 3.7 receives an incoming event v in state s_0, it forwards this event to its output port p_7 with a delay between 2 and 3 time units and stays in state s_0. If it receives an event *crit* in state s_0, meaning that the distance sensor has identified a critical distance to the preceding car, this event is forwarded to port p_7 with a delay between 2 and 4 time units and the state is changed to s_1. When in state s_1 a speed value v is received, it is still sent at port p_7 but additionally a brake event b is produced at port p_8 after a delay between 5 and 8 time units to initiate a braking action. It is also allowed that output ports are not connected to any channel. We refer to those ports as *sink ports* and indicate them by the $\perp$ symbol. As an example, we consider again node f_c, which sends an event *ok* to port $p^\perp$ if an event *ok* was received at input port p_5 resulting in an execution delay between 1 and 2 time units. An event produced at a sink port will not be received by any function node.

With this knowledge, we will now describe and motivate the whole functionality of the example system in Figure 3.7. The source nodes ϕ_1 and ϕ_2 periodically deliver values of two redundant speed sensors denoted as v_1 and v_2, which are aggregated by the function node f_v to a single speed value v. To assure that the time distance between two values v_1 and v_2 that are needed to determine the next aggregated value v is bounded, both sensors deliver their values with the same period. The source node ϕ_3 emits events from the event set $\{d_1, ..., d_n\}$, which represent the different values a distance sensor may measure. Depending on this value, the function node f_d decides

whether this distance is critical resulting in the events *ok* and *crit*. Here, the values d_1 to d_3 represent critical distances and all distances greater than d_3 are interpreted as noncritical. The function node f_c is a controller node that receives the speed value v and the distance events *ok* and *crit* and decides if a braking action needs to be initiated. The states are used to store if the latest distance value was *crit* (s_1) or *ok* (s_0). If it was *crit*, a braking is necessary and an additional brake event b is sent to the actuator node f_b. This induces a longer execution time between 5 and 8 time units because the intensity of the braking action needs to be computed. The speed value v is forwarded regularly to a function node f_e, which represents the environment e.g. an interface to the driver. Additionally, an event *crit* is sent to f_e if it was received to make the critical situation also available to other systems and the driver.

In the following, a basic function network is defined where we call channels *basic channels*. The set of events that may activate a function node at a specific input port p is called *activation set* and denoted as $\Sigma^{act}(p)$.

Definition 3.2.1 (Basic Function Network) *A basic function network is a tuple* $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ *where:*

- Σ *is a finite set of events.*
- $\mathcal{P} = \mathcal{P}^I \uplus \mathcal{P}^O$ *is a finite set of input and output ports. We define* $\Sigma(p) \subseteq \Sigma$ *as the set of events that can be observed at port* $p \in \mathcal{P}$*, and* $\Sigma(P) = \bigcup_{p \in P} \Sigma(p)$ *for* $P \subseteq \mathcal{P}$*.*
- $\mathcal{C} \subseteq \mathcal{P}^O \times \mathcal{P}^I$ *is a set of* basic *channels such that each input port is connected to at least one channel, and each output port is connected to at most one channel:*

$$\forall p_i \in \mathcal{P}^I : \exists (p_o, p_i) \in \mathcal{C},$$
$$\forall (p_o, p_i), (p'_o, p'_i) \in \mathcal{C} : p_o = p'_o \implies p_i = p'_i$$

  Furthermore, the output ports of any two channels connected to the same input port have disjoint event sets:

$$\forall (p_o, p_i), (p'_o, p_i) \in \mathcal{C} : p_o \neq p'_o \implies \Sigma(p_o) \cap \Sigma(p'_o) = \emptyset,$$

  and it holds $\Sigma(p_i) = \bigcup_{(p_o, p_i) \in \mathcal{C}} \Sigma(p_o)$*.*

- Φ *is a finite set of event sources* $\phi = (EP, \mathcal{P}^{out})$ *where* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *is an event pattern with* $J < P^-$*,* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports,* $\mathcal{P}^{out} \neq \emptyset$*, and all ports share the same event set:*

$$\forall p_o \in \mathcal{P}^{out} : \ \Sigma(p_o) = \Sigma^{EP}$$

  We define $\mathcal{P}^{out}(\phi) := \mathcal{P}^{out}$*.*

- $\mathcal{F}$ *is a finite set of function nodes* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ *where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports,* $\mathcal{P}^{in} \neq \emptyset$*, and* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports. We define the* activation set *of each input port* $p_i \in \mathcal{P}^{in}$ *as follows:*

$$\Sigma^{act}(p_i) = \{\{\sigma_1, ..., \sigma_n\} \mid \exists p_o \in \mathcal{P}^{out} : (p_o, p_i) \in \mathcal{C} \wedge \sigma_j \in \Sigma(p_o)\}$$

$\mathcal{A} = (S, s_0, T)$ is a timed transition system where $S \neq \emptyset$ is a finite set of states, $s_0 \in S$ is the initial state, and T is a transition function

$$T = \bigcup_{p_i \in \mathcal{P}^{in}} T_{p_i}, \quad T_{p_i} := p_i \times \Sigma^{act}(p_i) \times S \to \Psi \times S$$

$$\text{and } \Psi = \bigcup\nolimits_{\emptyset \neq P \subseteq \mathcal{P}^{out}} \{\psi : P \to (\Sigma \times \mathbb{N}^+ \times \mathbb{N}^+) \mid \psi(p_o) = (\sigma, \delta^-, \delta^+) \implies \sigma \in \Sigma(p_o), \delta^- \leq \delta^+\}$$

where each transition in T maps an input port, an activation event and a state to an output specification *and a successor state. An output specification $\psi \in \Psi$ maps output ports to output events and delay intervals. We define $\mathcal{P}^{in}(f) := \mathcal{P}^{in}$ and $\mathcal{P}^{out}(f) := \mathcal{P}^{out}$.*

It must further hold that each the sets of input and output ports of nodes and sources are disjoint i.e.

$$\begin{aligned} \forall \phi_1, \phi_2 \in \Phi, f_1, f_2 \in \mathcal{F}: \ & \mathcal{P}^{in}(f_1) \cap \mathcal{P}^{in}(f_2) = \emptyset, \\ & \mathcal{P}^{out}(f_1) \cap \mathcal{P}^{out}(f_2) = \emptyset \\ & \mathcal{P}^{out}(\phi_1) \cap \mathcal{P}^{out}(\phi_2) = \emptyset \\ & \mathcal{P}^{out}(\phi_1) \cap \mathcal{P}^{out}(f_1) = \emptyset \end{aligned}$$

◇

Please note, that a set Ψ of output specifications for a function node f denotes all non-empty subsets of output ports of f in combination with output events and delay intervals $[\delta^-, \delta^+]$. Hence, the transition function of a function node allows to send events to any combination of output ports, depending on the current state and input event. Also the delays can be chosen freely for each such output event. In the following we write functions ψ as sets $\{(p'_1, e'_1, [\delta^-_1, \delta^+_1]), ..., (p'_n, e'_n, [\delta^-_n, \delta^+_n])\}$ or even shorter as $\{(p'_1, e'_1, \delta_1), ..., (p'_n, e'_n, \delta_n)\}$ where $\delta_i = [\delta^-_i, \delta^+_i]$. For sets with one element we write abbreviatory $(p'_i, e'_i, [\delta^-_i, \delta^+_i])$ instead of $\{(p'_i, e'_i, [\delta^-_i, \delta^+_i])\}$.

3.2.2. Extended Function Networks

Specification languages such as Simulink often allow the explicit modeling of data objects and data access to capture not only control flow but also data flow. In order to be able to represent such models of specifications in function networks, the possibility to model data flow is an important feature. Function networks capture internal task states and complex execution patterns based on event values, which can be employed to model also data access. We can define for example function nodes modeling storage of variables, and also FIFO buffers.

To simplify modeling of data flow, we however want to be able to model explicit data storage. Furthermore, we would like to explicitly distinguish between computation and communication nodes. To this end, the function network formalism is

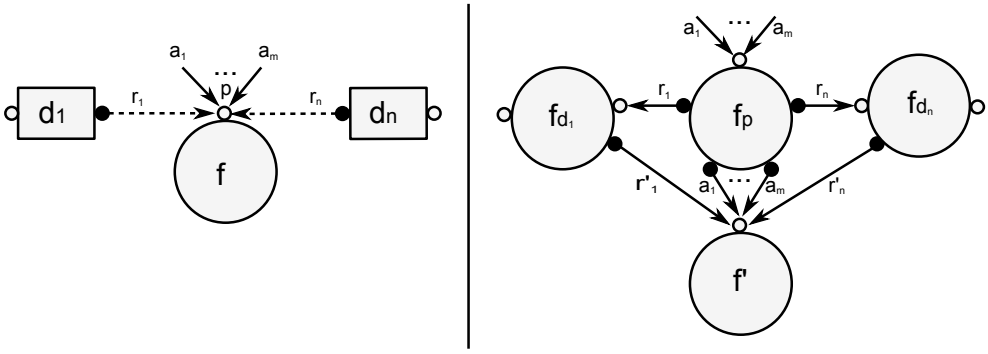


Figure 3.8.: Left: Read channels from Data Nodes d_1 to d_n
Right: Translation of Read Channels into Basic Function Network

extended by data nodes, that are special function nodes modeling specific data objects for communication between tasks. We define different types of data nodes as persistent ones like shared variables and FIFO buffers, and volatile ones like signals.

A further enrichment is the introduction of *delay channels* allowing to model communication between tasks in a more abstract way. In a later refinement step, when a function network becomes deployed and analyzed, these communication delays may be replaced by those without delays, and additional signals may be inserted that capture the respective delays. We define two types of delay channels namely activation and read channels. While *activation channels* model control flow and cause an activation at their target function node, *read channels* model data dependencies, that is a reading access by a function node to a data node. Read channels are depicted as dotted arcs and can be represented using the basic function network definition as shown in Figure 3.8. On the left, a function node f reads data from n data nodes d_1 and d_n at its activation. The coordination of the read process is hidden in the complex input port p, which is defined as a special function node f_p as shown on the right of Figure 3.8. This function node is activated before f is activated and triggers the data nodes d_1 to d_n whose translation to function nodes is described later. If all data nodes have been executed and have sent an event to the input port of the actual function node f', this node is activated.

Four common kinds of data nodes have been identified. A *shared* data node, as depicted in Figure 3.9 at the upper left, stores incoming events from its input ports and returns the currently stored event when it is requested via an output port. In this example, there are two incoming events a and b that may be stored. The internal transition system, shown at the bottom of Figure 3.9, contains one state per input event that may be stored resulting in the states s_a and s_b in this example. As soon as a new event a or b is written, the state is updated accordingly to s_a or s_b. When a read request r occurs, the respective event is returned depending on the current state.

A *FIFO* data node (see Figure 3.10, upper left) stores a bounded queue of events and returns them in FIFO order if requested. In contrast to the shared data node, here an event may only be read once. Hence, the transition system contains one state

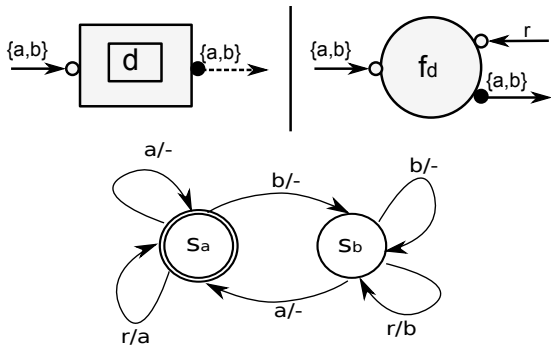


Figure 3.9.: Translation of Shared Data Node

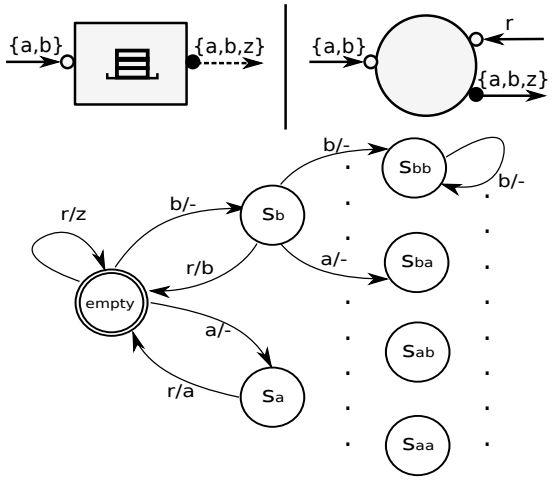


Figure 3.10.: Translation of FIFO Data Node

for each possible order and number of events in the event queue. At the bottom of Figure 3.10, an example transition system for a FIFO with 2 places is depicted. If any input event a or b occurs, it is stored and the state is changed to represent the current queue state. A read event r returns the 'oldest' event and changes the state accordingly. In principle, different ways exist to deal with queue under- and overflow, for example by introducing an error state in case of an overflow. In the context of this work, a buffer sends a special event z when a read request occurs and the buffer is empty. In the case that the buffer is full and another event should be written, this event is ignored.

In Figure 3.11, a *signal* data node is shown that does not store any events permanently but immediately forwards them to all output ports. This results in a transition system with a single state and one transition for each incoming event (not depicted in Figure 3.11). In this example, the data node has two input ports where on each port two events may arrive. If an event is received, it is forwarded to both output ports.

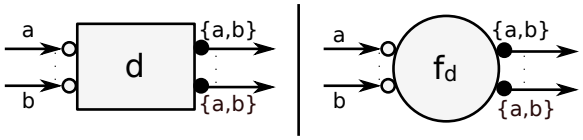


Figure 3.11.: Translation of Signal Data Node

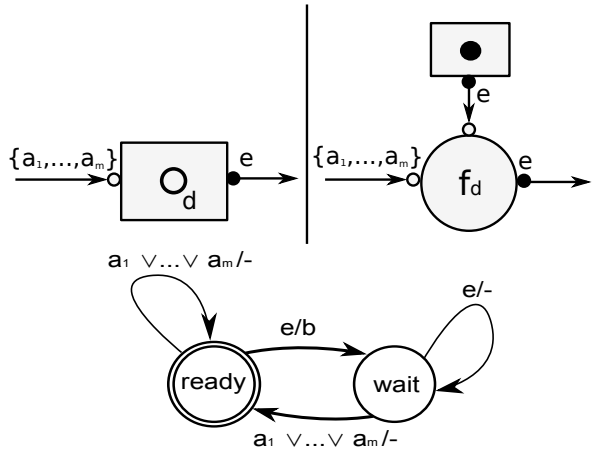


Figure 3.12.: Translation of Finite Source Data Node

Another data node type is a *finite source*. It produces an initial event at its output port at system startup while emitting the next event not before it has received an event at its input port. This node type is used to model cycles in function networks and is depicted in Figure 3.12 on the left. Its translation into basic function networks is shown on the right. At the first activation by an event e from an event source, an initial event is sent by the function node. Then, an event b is first emitted at each output port if an event a_i was received on the input port. This is realized by the internal transition system, which is shown at the bottom of Figure 3.12. It starts in the 'ready' state where an event e from the source node leads to an event b at the output port and a change to the state 'wait'. Whenever an input event a_i arrives, the state is changed back to 'ready'. During the 'wait' state an event from the source does not lead to an output event b i.e., it has to be assured that an event a_i arrives in time to not loose any source events.

These extensions lead to the following definition of an extended function network.

Definition 3.2.2 (Extended Function Network) *An extended function network is a tuple* $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *where* Σ, $\mathcal{P}$ *and* Φ *are defined as for the basic function network and*

- $\mathcal{C} = \mathcal{C}^A \uplus \mathcal{C}^R \subseteq (\mathcal{P}^O \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathcal{P}^I)$ *is a set of delay channels* $c = (p^{out}, \delta, p^{in})$ *where* $\delta = [\delta^-, \delta^+]$ *is a delay interval with* $(\delta^+ > 0) \Longrightarrow (\delta^- > 0)$, $\mathcal{C}^A$ *is a set of* activation *channels and* $\mathcal{C}^R$ *is a set of* read *channels.*

- $\mathcal{D} = \mathcal{D}_{signal} \uplus \mathcal{D}_{fifo} \uplus \mathcal{D}_{shared} \uplus \mathcal{D}_{fsource}$ *is a set of data nodes, where*
  - $d_{sig} = (\mathcal{P}^{in}, \delta, \mathcal{P}^{out}) \in \mathcal{D}_{signal}$ *is a signal data node, where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports with* $\mathcal{P}^{in} \neq \emptyset$, $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval and* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports with* $\mathcal{P}^{out} \neq \emptyset$.
  - $d_{fifo} = (\mathcal{P}^{in}, \delta, c, \mathcal{P}^{out}) \in \mathcal{D}_{fifo}$ *is a FIFO data node, where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports with* $\mathcal{P}^{in} \neq \emptyset$, $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval,* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports with* $\mathcal{P}^{out} \neq \emptyset$ *and* $c \in \mathbb{N}^+$ *is a capacity.*
  - $d_{shared} = (\mathcal{P}^{in}, \delta, \sigma_0, \mathcal{P}^{out}) \in \mathcal{D}_{shared}$ *is a shared data node, where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports with* $\mathcal{P}^{in} \neq \emptyset$, $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval,* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports with* $\mathcal{P}^{out} \neq \emptyset$ *and* $\sigma_0 \in \bigcup_{p^{in} \in \mathcal{P}^{in}} \Sigma(p^{in})$ *is the event that is stored initially.*
  - $d_{fsrc} = (\{p^{in}\}, \delta, EP, \{p^{out}\}) \in \mathcal{D}_{fsource}$ *is a finite source data node, where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports,* $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval,* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports and* EP *is an event pattern.*

  For a data node $d \in \mathcal{D}$, *we define* $\mathcal{P}^{in}(d) = \mathcal{P}^{in}$ *and* $\mathcal{P}^{out}(d) = \mathcal{P}^{out}$.

The input ports of data nodes and function nodes are each disjoint. The same holds for output ports.

$$\forall n_1, n_2 \in \mathcal{F} \cup \mathcal{D} : \; \mathcal{P}^{in}(n_1) \cap \mathcal{P}^{in}(n_2) = \emptyset, \\ \mathcal{P}^{out}(n_1) \cap \mathcal{P}^{out}(n_2) = \emptyset$$

Each input port of a function node has at least one incoming activation channel

$$\forall p \in \mathcal{P}^{in}(f), f \in \mathcal{F} : \exists \; (p', p) \in \mathcal{C}^A$$

Each input port of a data node has exactly one incoming activation channel

$$\forall p \in \mathcal{P}^{in}(d), d \in \mathcal{D} : \exists! \; (p', p) \in \mathcal{C}^A$$

Read channels lead exclusively from FIFO or shared data nodes to function nodes.

$$\forall (p, p') \in \mathcal{C}^R : \; \exists! \; d \in \mathcal{D}_{fifo} \cup \mathcal{D}_{shared} \mid p \in \mathcal{P}^{out}(d) \; \wedge \\ \exists! \; f \in \mathcal{F} \mid p' \in \mathcal{P}^{in}(f)$$

◇

As mentioned before, any extended function network can be uniquely translated into a basic function network. Thus, we will define semantics only for basic function networks and implicitly define the semantics of extended function networks by the respective translation. For simplicity, we assume that each FIFO and each shared data node has exactly one input port and one output port. Nevertheless, the translation can be generalized to an arbitrary numbers of ports.

Definition 3.2.3 (Extended Function Network Translation) *Given an extended function network* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *where,* $\forall d \in \mathcal{D}_{fifo} \cup \mathcal{D}_{shared}$ *holds* $|\mathcal{P}^{in}(d)| = |\mathcal{P}^{out}(d)| = 1$. *The translation of fn into a basic function network bfn is defined as*

$$bfn = (\Sigma_b, \mathcal{P}_b, \mathcal{C}_b, \Phi_b, \mathcal{F}_b)$$

1. *We first define the translation of data nodes as follows:*
   a) *Each signal data node* $d = (\mathcal{P}^{in}, \delta, \mathcal{P}^{out}) \in \mathcal{D}_{signal}$ *with* $\mathcal{P}^{out} = \{p_1^{out}, ..., p_m^{out}\}$ *is translated to a function node* $f_d = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out}) \in \mathcal{F}_b$ *with*
      - $S = \{s_0\}$
      - $T = \{(p^{in}, e, s_0 \to \{(p_1^{out}, e, \delta), ..., (p_m^{out}, e, \delta)\}, s_0) \mid p^{in} \in \mathcal{P}^{in}, e \in \Sigma(p^{in})\}$

   b) *Each FIFO data node* $d = (\{p^{in}\}, \delta, c, \{p^{out}\}) \in \mathcal{D}_{fifo}$ *is translated into a function node* $f_d = (\{p^{in}, p_d^r\}, (S, s_0, T), \{p^{out}, p_d^\perp\}) \in \mathcal{F}_b$ *where*
      - $S = \{empty\} \cup \{s_{\sigma_1,...,\sigma_k} \mid \sigma_i \in \Sigma(p^{in}), i \in \{1, ..., k\}, k \in \{1, ..., c\}\}$
      - $s_0 = empty$
      - *for each state* $s_{\sigma_1,...,\sigma_k}$ *and each input symbol* $\sigma \in \Sigma(p^{in})$, *we define a transition* $t = (p^{in}, \sigma, s_{\sigma_1,...,\sigma_k} \to (p_d^\perp, \sigma, \delta), s') \in T$ *where*

$$s' = \begin{cases} s_{\sigma_1,...,\sigma_k,\sigma} & ,if\ k < c \\ s_{\sigma_1,...,\sigma_k} & ,else \end{cases}$$

      - *for each symbol* $\sigma \in \Sigma(p^{in})$, *we define a transition* $t = (p^{in}, \sigma, empty \to (p_d^\perp, \sigma, \delta), s_\sigma) \in T$
      - *for each state* $s_{\sigma_1,\sigma_2,...,\sigma_k}$ *and each read event* $r \in \Sigma(p_d^r)$, *we define a transition* $t = (p_d^r, r, s_{\sigma_1,\sigma_2,...,\sigma_k} \to (p^{out}, \sigma_1, \delta), s_{\sigma_2,...,\sigma_k}) \in T$
      - *for the empty state, we define a transition that returns the z symbol to denote that the buffer is empty i.e.* $\forall r \in \Sigma(p_d^r) : \exists t = (p_d^r, r, empty \to (p^{out}, z, \delta), empty) \in T$

   c) *Each shared data node* $d = (\{p^{in}\}, \delta, \sigma_0, \{p^{out}\}) \in \mathcal{D}_{shared}$ *is translated into a function node* $f_d = (\{p^{in}, p_d^r\}, (S, s_0, T), \{p^{out}, p_d^\perp\}) \in \mathcal{F}_b$ *where*
      - $S = \{s_\sigma \mid \sigma \in \Sigma(p^{in})\}$
      - $s_0 = s_{\sigma_0}$
      - *for each* $\sigma \in \Sigma(p^{in})$ *and each* $s \in S$ *there exists a transition* $t = (p^{in}, \sigma, s \to (p_d^\perp, \sigma, \delta), s_\sigma)$
      - *for each state* $s_\sigma \in S$ *and each read event* $r \in \Sigma(p_d^r)$ *exists a transition* $t = (p_d^r, r, s_\sigma \to (p^{out}, \sigma, \delta), s_\sigma)$

   d) *Each finite source data node* $d = (\{p^{in}\}, \delta, EP, \{p^{out}\}) \in \mathcal{D}_{fsource}$ *is translated into a function node* $f_d = (\{p^{in}, p_d^{tr}\}, (S, s_0, T), \{p^{out}, p_d^\perp\}) \in \mathcal{F}_b$ *where*
      - $S = \{ready, wait\}$

- $s_0 = ready$
- $T =$

$$\begin{aligned}
&\{\ (p^{in}, a, ready \to (p_d^{\perp}, a, \delta), ready),\\
&\quad (p^{in}, a, wait \to (p_d^{\perp}, a, \delta), ready)\ \mid\ a \in \Sigma(p^{in})\ \}\\
\cup\ &\{\ (p_d^{tr}, e, ready \to (p^{out}, e, \delta), wait)\\
&\quad (p_d^{tr}, e, wait \to (p_d^{\perp}, e, \delta), wait)\ \mid\ e \in \Sigma(p_{tr}^{in})\ \}
\end{aligned}$$

and a source node $\phi_d = (EP, \{p_{\phi_d}^{out}\})$ *with a channel* $c = (p_{\phi_d}^{out}, p_d^{tr}) \in \mathcal{C}_b$

2. *Each activation channel* $c = (p^{out}, [\delta^-, \delta^+], p^{in}) \in \mathcal{C}^A$ *is translated as follows:*
   a) *If* $\delta^-, \delta^+ > 0$, *c is translated into a function node* $f_c = (\{p_c\}, (\{s_0\}, s_0, T), \{p_c'\}) \in \mathcal{F}_b$ *where* $T = \{(p_c, e, s_0 \to (p_c', e, [\delta^-, \delta^+]), s_0) \mid e \in \Sigma(p)\}$ *and two basic channels* $(p^{out}, p_c) \in \mathcal{C}_b$ *and* $(p_c', p^{in}) \in \mathcal{C}_b$.
   b) *If* $\delta^- = \delta^+ = 0$, *c is translated into a basic channel* $c_b = (p^{out}, p^{in}) \in \mathcal{C}_b$.
3. *Function nodes with read channels are translated as follows:*

   Let $f \in \mathcal{F}$ *be a function node. Then* $f \in \mathcal{F}_b$ *and each input port* $p \in \mathcal{P}^{in}(f)$ *with a non-empty set of incoming read channels* $\mathcal{C}_p^R = \{c_{r_1}, ..., c_{r_n}\} \in \mathcal{C}^R$ *and activation channels* $\mathcal{C}_p^A = \{c_{a_1}, ..., c_{a_m}\} \in \mathcal{C}^A$ *is translated to a function node* $f_p = (\{p'\}, (\{s_0\}, s_0, T), \{p_{r_1}^{out}, ..., p_{r_n}^{out}, p_{a_1}^{out}, ..., p_{a_m}^{out}\})$, *which initiates the reading of all events from the connected data nodes where*
   a) *each activation channel* $c_{a_j} = (p^*, p) \in \mathcal{C}_p^A$ *is changed to* $c_{a_j} = (p^*, p')$ *and translated as described under 2.*
   b) *for each* $E = \{r_1, ..., r_n, a_1, ..., a_m\} \in \Sigma(p')$, *we define a transition*

$$(p', E, s_0 \to \Psi, s_0) \in T \text{ with}$$

   $\Psi = \{(p_{r_1}^{out}, r_1, [\epsilon, \epsilon]), ..., (p_{r_n}^{out}, r_n, [\epsilon, \epsilon]), (p_{a_1}^{out}, a_1, [\epsilon, \epsilon]), ..., (p_{a_m}^{out}, a_m, [\epsilon, \epsilon])\}$ *where* $\epsilon > 0$ *denotes a negligible small execution time.*
   c) *for each read channel* $c_{r_i} = (p_d^{out}, \delta_{r_i}, p) \in \mathcal{C}_p^R$ *from a data node* $d \in \mathcal{D}_{fifo} \cup \mathcal{D}_{shared}$ *that is translated following (1b) and (1c) to a function node* $f_d \in f_b$ *with* $\mathcal{P}^{in}(f_d) = \{p_d^{in}, p_d^{r_i}\}$ *and* $\mathcal{P}^{out}(f_d) = \{p_d^{out}\}$
      i. *an activation channel* $(p_{r_i}^{out}, \delta_{r_i}, p_d^{r_i})$ *is created and translated as described under 2.*
      ii. *a basic channel* $(p_d^{out}, p) \in \mathcal{C}_b$ *is created leading from the output port of d to the input port p of f.*
   d) *for each activation channel* $c_{a_j} \in \mathcal{C}^A$, *a basic channel* $(p_{a_j}^{out}, p) \in \mathcal{C}_b$ *is created forwarding events to f to be synchronized with read events.*
4. *Each source* $\phi \in \Phi$ *is also contained in* Φ_b. ◇

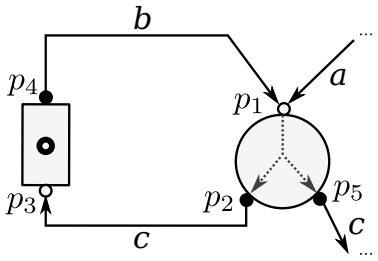


Figure 3.13.: Example of a Synchronization (AND) Loop

Please note, that the output port $p_\perp$ of the translation of FIFO, shared and finite source data nodes is skipped for brevity in the respective figure. For the rest of the document, we will shortly write *function network* for an extended function network. Otherwise, we explicitly refer to a basic function network.

Modeling Loops Loops are an essential part of system modeling and often used in practice to specify for example a controller unit. Thus, it is important to allow the modeling of such control loops also in function networks. We give an understanding how loops may be modeled using function networks with the help of some simple examples. There mainly exists two ways for modeling loops that are synchronization (AND) and superposition (OR) loops.

In Figure 3.13, a simple synchronization loop is depicted, which means that a control flow starts and ends in a synchronization at an input port of a function node. This kind of loop has to contain a finite source data node because otherwise we would need a buffer of unbounded size due to a missing initial event from the loop. The function node in this example has one input port p_1 and is activated if both an event a and b is received. The transition of the function node is indicated by a split dotted arrow meaning that with each activation an event c is emitted at both output ports. The first event b is guaranteed to be available at system start-up by the finite source data node. As soon as also an a event is present, the function node is executed emitting an event c at both its output ports p_2 and p_5 after specific delays. The next activation cannot occur before the event c emitted at port p_2 has reached the finite source data node, which then emits the next instance of the event b. This way, a minimum distance between succeeding b events is guaranteed, which equals the minimum delay of the complete loop. In general, such a loop may be more complex involving other function and data nodes.

The second type of loop is a superposition loop, which is shown in Figure 3.14. Actually, this is not a real control loop because the function node is activated at different ports and thus no port is involved more than once in the cycle. This is also the reason why we do not need a finite source data node here. The function node is activated first when an event a occurs at its input port p_1, and then executes and produces an event b at its output port p_2 after a specific delay. This immediately leads to another activation at input port p_3 and an output event b at output port p_4.

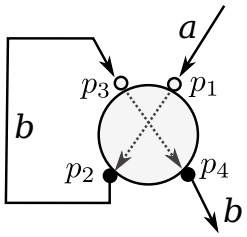


Figure 3.14.: Example of a Superposition (OR) Loop

To decide if such a loop is bounded may become harder as in this example as soon as a function node has different states leading to different output events. A class of function networks where boundedness is decidable is presented in Section 3.4. The most important property for this class is called state-independence and is introduced in the next section.

3.2.3. Properties of Function Networks

We will define some basic properties of function networks that are needed for this work. We start with the definition of a path between ports to describe how events may flow through a function network. We do this for basic function networks only while the paths for extended function networks can be derived from the translation given in Def. 3.2.3.

Definition 3.2.4 (Path) *Let $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ be a basic function network. There exists a path between two ports $p, p' \in \mathcal{P}$ written as $path(p, p')$ if and only if one of the following conditions holds*

1. *there exists a function node with a transition t from p to p' i.e.*

$$\exists f = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out}) \in \mathcal{F} : \exists t = (p, E, s \to \Psi, s') \in T \wedge \exists (p', e'_0, \delta) \in \Psi$$

2. *there exists a channel from p to p' i.e.*

$$\exists c = (p, p') \in \mathcal{C}$$

3. *there exists a shortest sequence of paths from p to p' i.e.*

$$path(p, p_1) \wedge path(p_1, p_2) \wedge ... \wedge path(p_n, p')$$

such that $\nexists (path(p, p_{1'}) \wedge path(p_{1'}, p_{2'}) \wedge ... \wedge path(p_m, p'))$ with $m < n$. ◇

Based on the definition of paths, we can argue about reachability of ports in a function network. A port is called *reachable* if there exists a path from a source node to that port. A function network is called reachable if each of its ports is reachable.

Definition 3.2.5 (Reachability) *Let $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ be a basic function network. A port $p \in \mathcal{P}$ is called* reachable *if there exists a path from a source node to p or p belongs to an event source i.e.*

$$p \text{ is reachable} \iff \exists \phi = (EP, \mathcal{P}^{out}) \in \Phi \wedge \exists p_{src} \in \mathcal{P}^{out} \wedge path(p_{src}, p) \vee \exists \phi = (EP, \mathcal{P}^{out}) \in \Phi \wedge p \in \mathcal{P}^{out}$$

bfn is called reachable *if each of its ports $p \in \mathcal{P}$ is reachable.* ◇

A usual property of graph formalisms are cycles. We define cycles on ports of function networks based on the definitions of paths.

Definition 3.2.6 (Cycles) *Let $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ be a basic function network. There exists a cycle starting from port $p \in \mathcal{P}$ written as $cycle(p)$ if there exists a path that starts and ends at p i.e.*

$$cycle(p) \iff path(p, p)$$

bfn is called cyclic *if there exists at least one cycle in bfn and* acyclic *otherwise.* ◇

Please note, that a cycle does not necessarily mean that there also exists a cyclic causal dependency in a function network. This is due to the fact, that the definition of cycles is static and it cannot be determined if a path is ever taken. For example, there may exist paths that will never be taken because the necessary combination of input events and states does never occur.

An important property that we will need to define a class of function networks where boundedness is decidable is called *state-independence*. A function node is called state-independent if its output behavior in terms of events that occur at its output port does not depend on its internal state while the transition delays may still vary. This means that, independently from the state, an activation of a function node by any set of input events leads to events at the same output ports.

Definition 3.2.7 (State-Independence) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node with $\mathcal{A} = (S, s_0, T)$ and $p^{out} \in \mathcal{P}^{out}$ be an output port. p^{out} is state-independent if for each input port $p \in \mathcal{P}^{in}$ with a transition producing an event $b \in \Sigma(p^{out})$ also each other transition triggered by an event from $\Sigma^{act}(p)$ produces an event from $\Sigma(p^{out})$ i.e.*

$$\forall p \in \mathcal{P}^{in} \mid \exists\ (p, E, s \to \{...(p^{out}, b, \delta)...\}, s') \in T \text{ with } b \in \Sigma(p^{out}) :$$
$$\forall s_j \in S, E' \in \Sigma^{act}(p) : \exists (p, E', s_j \to \{...(p^{out}, b_j, \delta_j)...\}, s'_j) \in T \text{ with } b_j \in \Sigma(p^{out})$$

A function node is state-independent if all its output ports are state-independent. ◇

Timing Constraints We shortly introduce the type of timing constraints we consider for this work that are end-to-end deadlines. End-to-end deadlines define upper bounds for executions of function node chains. A function node chain is the set of function nodes on all paths from an input port of a start node to the output port of an end node. In Figure 3.15, an example for an end-to-end deadline of 120 milliseconds is

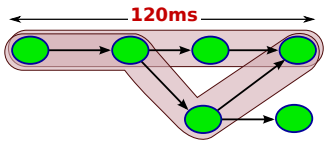


Figure 3.15.: End-to-end Deadline

depicted. Here, two paths exist from the start node to the end node of the deadline emphasized by the red shaped area.

We restrict to end-to-end deadlines for simple linear node chains. More complex chains where, for example, the control flow is split and joined again, are divided into sets of linear chains, which is always possible. The corresponding deadline reasons about the time span between the activation of the first node and finishing the execution of the last node of the chain, which coincides with sending the respective output event.

A deadline $D \in \mathbb{T}$ defines a maximum delay between two events e_1, e_2 like activations or completions of function nodes. If we take an arbitrary execution of a function network, let $t_i(e_1)$ be the time of the i^{th} occurrence of its input event e_1, and $t_i(e_2)$ for its output event e_2, respectively. We say the deadline is satisfied if and only if $|t_i(e_2) - t_i(e_1)| \leq D$ for all $i \in \mathbb{N}^+$, and all such executions.

3.3. Semantics of Function Networks

There are mainly two reasons to define semantics of function networks formally. First, we need a formal semantic representation to be able to analyze a function network, for example to show timing properties in terms of end-to-end deadlines.

Second, we aim at showing semantic preservation when we translate Simulink models into function networks. Thus, we need formal semantics for both formalisms. Additionally, we need to specify which properties we want to preserve. Here, we are mainly interested in causality and time delays between events and introduce a set of patterns to capture those properties. The same properties should be preserved when merging nodes to create tasks. Thus, we start with defining such patterns before defining the actual semantics of function networks.

3.3.1. Causality and Timing Patterns

In order to be able to reason about semantic properties, we want to preserve, we define three types of patterns to capture causal and timing properties of function networks. First, *causality patterns* describe a conditional causal dependency with a time delay between two single events or sets of events. Second, *condition patterns* are used to show that a condition or property holds during a specific time interval that is defined by two event occurrences. Finally, we define the *waiting time pattern*, which states how long we have to wait after the occurrence of one event to see another.

Causality Pattern

For Simulink models, a partial order [48, 82] on input and output signals of a block is sufficient to express valid orderings of block executions under the assumption that the internal behavior of blocks is not considered. This is due to the fact that all input signals of a block are synchronized because a block must first be executed if all its inputs are available. Thus, there are no alternative ways to activate a block in terms of an OR-activation. In function networks, the internal transition system of a function node is able to model more complex causal dependencies between its input and output events which may not be covered by a partial order. Beside the non-determinism induced by an OR-activation, also different states may lead to different output behavior.

This is exemplified in Figure 3.16, where a function node is depicted on the left. It has two input ports where input events are received to activate the node and two output ports where events are produced. On the right, the internal transition system of this function node is shown. It has two states s_1 and s_2 where s_1 is the initial state. If an event a is received, the node is activated, stays in state s_1 and produces two events b and c on different output ports. For simplicity, we omit the concrete delays here and assume that the output events may be produced in an arbitrary order. If an event x is received in state s_1, the state is changed to s_2 and an output event b is produced. In state s_2 an event a leads to an output event b as well and an event x leads back to state s_1 and produces an output event c.

If we try to describe this behavior by a partial order, we recognize that this cannot be done without losing information or getting conflicts. In this example, we would get the following partial order relations:

$$a < b,\ a < c,\ x < b,\ x < c$$

This would induce that a and x need to occur before b may occur, which is not correct. Furthermore, we cannot derive from this partial order whether the output events b and c may occur concurrently or not because this depends on the state of the function node. This is because a partial order is not able to express non-determinism in terms of a superposition. A formalism where this may be covered are event structures [61], which offer an additional relation to the partial order called conflict relation. Two events in a conflict relation must not occur concurrently.

However, we want to capture the non-determinism induced by states of the internal transitions systems more explicit and decided to rely on a pattern from the requirement specification language (RSL) [65]. The idea of the RSL is to define patterns that are typically used to describe requirements a designer defines for a system under development. Beside an intuitive natural language description it was the main goal to also define a formal representation in terms of timed automata to enable formal analysis methods such as model-checking.

We use a functional pattern from the RSL to reason about causal dependencies in function networks. The so-called 'F1' pattern exists in different variants while we rely on the one with the following natural language representation: 'whenever e occurs, f occurs during $[min, max]$' where e and f are events and $min, max \in \mathbb{N}_0$ describe

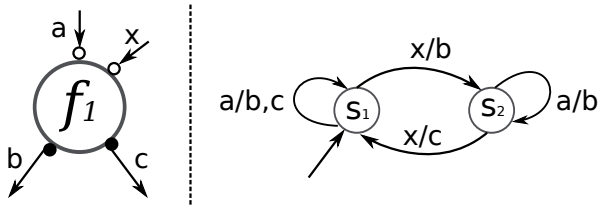


Figure 3.16.: Example for Causal Dependencies in Function Nodes

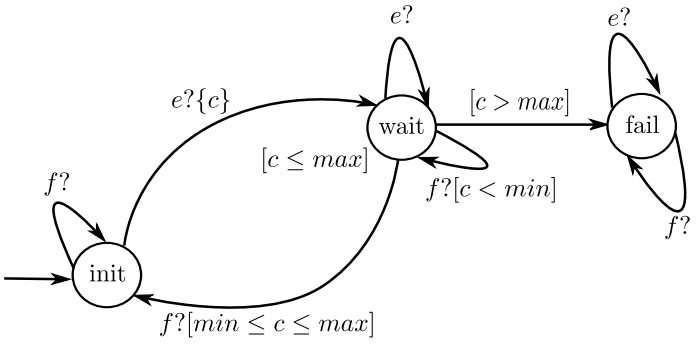


Figure 3.17.: Observer Automaton for RSL 'F1' Pattern [65]

a time interval with $min \leq max$. Its semantics is defined as a timed automaton as depicted in Figure 3.17 [65].

It starts in the 'init' state and waits for an event e to occur. As soon as an e occurs, clock c is reset and state 'wait' is entered. Here, the transition back to state 'init' is taken if an event f arrives within a time interval of $[min, max]$. If an event f is received and $c < min$, the automaton remains in state 'wait'. As soon as the clock value is greater than max, state 'fail' is entered due to the invariant $c \leq max$ of state 'wait'. The property is violated if the state 'fail' is reached and satisfied otherwise. The automaton is receptive meaning that it is always able to consume any input event e or f. This is needed to avoid that overlapping invocations of events lead to a deadlock. In case of a deadlock, the 'fail' state may never be reached although the property is not satisfied. Thus, there are respective self-transitions for each state that are sensitive to each input event. In particular, if an event f is received before clock c has reached the value min, the automaton remains in state 'wait' and the property is not violated. Also in state 'fail' the automaton remains receptive but will never leave this state.

To represent also conditional causality, we use the possibility of RSL to express conditions in terms of boolean expressions resulting in the following natural language representation: 'whenever e occurs under $cond_1$, f occurs under $cond_2$ during $[min, max]$'. We restrict to disjunctions and conjunctions of boolean expressions such

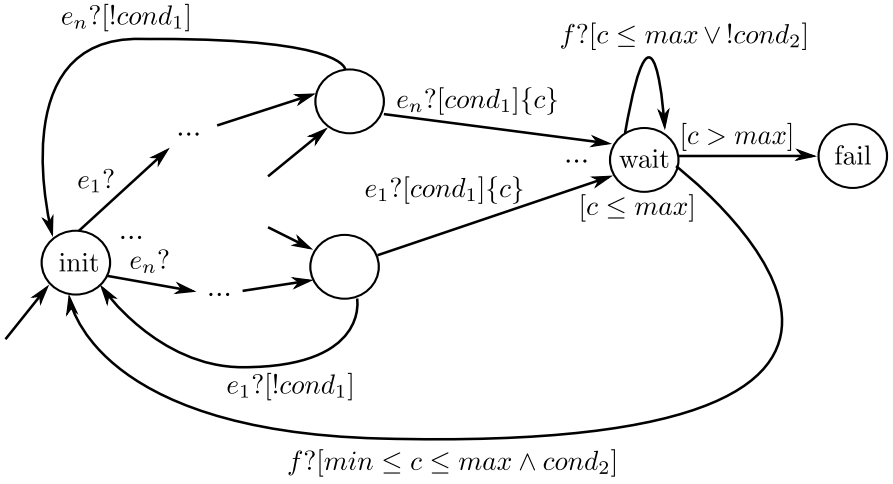


Figure 3.18.: Causality Pattern Automaton

as $'var1 = val1 \vee var2 = val2'$ where $var1$, $var2$ are variables and $val1$, $val2$ are values. As a further enrichment that also exists for the RSL, we allow to reason about sets of events while the order of their occurrence is arbitrary i.e., each order is allowed resulting in the following definition.

Definition 3.3.1 (Causality Pattern) *The causality pattern states the following: 'Whenever $\{e_1, ..., e_n\}$ occurs under $cond_1$, f occurs under $cond_2$ during $[min, max]$' written as*

$$\{e_1, ..., e_n\}[cond_1] \xrightarrow{[min,max]} f[cond_2]$$

where $\Sigma^{in} = \{e_1..., e_n\}$ are input events, f is an output event with $f \notin \Sigma^{in}$, $cond_1$ and $cond_2$ are boolean expressions and $min, max \in \mathbb{R}_0^+$ are delays with $min \leq max$. $\diamond$

Semantics of the causality pattern is defined as the timed automaton depicted in Figure 3.18. The automaton is again defined to be receptive i.e., for each state s and each event $\sigma \in \Sigma^{in} \cup \{f\}$ there additionally exists a transition $s \xrightarrow{\sigma?,true,\{\}} s$ if there does not already exist a transition $s \xrightarrow{\sigma?,true,\{\}} s'$. For readability these transitions are not shown in the figure. The automaton waits for an arbitrary sequence of input events from $\{e_1, ..., e_n\}$ and resets the clock c as soon as the last input event was received and the condition $cond_1$ holds leading to state 'wait'. If the condition $cond_1$ does not hold when the last input event occurs, the automaton returns to state 'init'. In state 'wait', it waits for an event f to occur within a time interval of $[min, max]$. If during this time interval no event f is observed, the automaton reaches the 'fail' state and the pattern is violated. Otherwise, the automaton returns to its initial state. This means that if an event f occurs before min time units, this does not influence the satisfaction of this property.

The intuitive meaning of causality is that the occurrence of all events from the set $\{e_1, ..., e_n\}$ is sufficient for the occurrence of event f within a time interval of $[min, max]$. However, this is not inevitably a necessary condition for f to occur because there may exist another set of input events that also leads to f. In a function node this may happen for example by an OR-activation on different input ports of a function node each leading to the same output event at the same output port.

We introduce the following abbreviations for the causality pattern:

- Empty Condition:

$$\{e_1, ..., e_n\} \xrightarrow{[min,max]} f \iff \{e_1, ..., e_n\}[true] \xrightarrow{[min,max]} f[true]$$

- Input sets with one event:

$$e\ [cond_1] \xrightarrow{[min,max]} f\ [cond_2] \iff \{e\}[cond_1] \xrightarrow{[min,max]} f\ [cond_2]$$

- Set of output events:

$$\begin{aligned} & \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min,max]} \{f_1, ..., f_m\}[cond_2] \\ \iff\ & \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min,max]} f_1[cond_2]\ \wedge ... \wedge \\ & \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min,max]} f_m[cond_2] \end{aligned}$$

Condition Pattern

Another functional pattern, we borrow from the RSL, is the 'F2' pattern, which we call *condition pattern*. It describes that a condition holds for a specific time interval while the bounds are defined by the occurrence of events. It is defined as follows:

Definition 3.3.2 (Condition Pattern) *The condition pattern states the following: 'Whenever e occurs, cond holds during $[e, f]$' written as*

$$[cond]\ \textit{holds during}\ [e, f]$$

where e,f are events and cond is a boolean expression. ◇

The semantics of this pattern is defined in Figure 3.19 in terms of a timed automaton [65]. As for the causality pattern, this automaton is also receptive to the events e and f in each state by adding the respective self-transitions. The automaton looks quite similar to the one in Figure 3.17 but needs no clock because the interval is defined by the occurrence of the events e and f. Thus, the automaton changes the state from 'init' to 'check' if an event e is received. As soon as an event f is received and the condition *cond* holds, it returns to the state 'init'. Whenever the condition *cond* does not hold between the occurrence of e and f, the automaton reaches the state 'fail' and the pattern is violated.

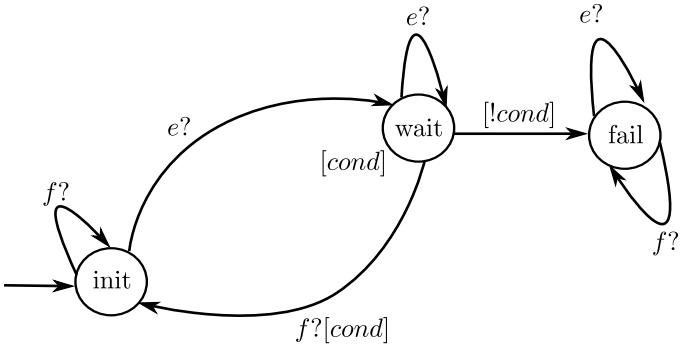


Figure 3.19.: Condition Pattern Automaton (RSL pattern 'F2' [65])

Waiting Time Pattern

In some cases it is desirable to make statements about event occurrences and the delay between them without assuming a causal dependency, which we refer to as *waiting time*. For function networks, this is in particular the case for input ports of function nodes where a number of input streams is synchronized. This means, we have to wait for the i^{th} occurrence of an event on each stream until we can observe a synchronization event. If we now focus on the i^{th} occurrence of an event on a single stream, the waiting time pattern may be used to capture the time we have to wait until we also see the i^{th} event on any other stream. The waiting time pattern is defined as follows:

Definition 3.3.3 (Waiting Time Pattern) *Let $L(A)$ be a timed language over an alphabet A and $L(B)$ be a timed language over an alphabet B. We define the lower and upper waiting time for two words $\omega_a = (a_i, t_i)_{i\in\mathbb{N}^+} \in L(A), \omega_b = (b_i, s_i)_{i\in\mathbb{N}^+} \in L(B)$ as follows:*

$$\Delta^-(\omega_a, \omega_b) = \max(0, \inf_i\{t_i - s_i\})$$
$$\Delta^+(\omega_a, \omega_b) = \sup(0, \sup_i\{t_i - s_i\}))$$

The lower and upper waiting time of two languages $L(A)$ and $L(B)$ are defined as follows:

$$\Delta^-(L(A), L(B)) = \inf(\Delta^-(\omega_a, \omega_b)|\omega_a \in L(A), \omega_b \in L(B))$$
$$\Delta^+(L(A), L(B)) = \sup(\Delta^+(\omega_a, \omega_b)|\omega_a \in L(A), \omega_b \in L(B))$$
$$\Delta(L(A), L(B)) = [\Delta^-(L(A), L(B)), \Delta^+(L(A), L(B))]$$

We further define the waiting time between the i^{th} event of a word $\omega_a \in L(A)$ and its k^{th} successor event with $k > 0$ as follows:

$$\Delta_k^-(\omega_a) = \inf_i\{t_{i+k} - t_i\}$$
$$\Delta_k^+(\omega_a) = \sup_i\{t_{i+k} - t_i\}$$
$$\Delta_k(\omega_a) = [\Delta_k^-(\omega_a), \Delta_k^+(\omega_a)]$$

The waiting time for a language $L(A)$ is defined as follows:

$$\Delta_k^-(L(A)) = \inf(\Delta_k^-(\omega_a)|\omega_a \in L(A))$$
$$\Delta_k^+(L(A)) = \sup(\Delta_k^+(\omega_a)|\omega_a \in L(A))$$
$$\Delta_k(L(A)) = [\Delta_k^-(L(A)), \Delta_k^+(L(A))]$$

For $k = 1$ we write short

$$\Delta^-(L(A)) = \Delta_1^-(L(A))$$
$$\Delta^-(L(A)) = \Delta_1^-(L(A))$$
$$\Delta(L(A)) = \Delta_1(L(A))$$

$\diamond$

Pattern Properties and Operations

We show properties and operations for the previously defined patterns that we need for several proofs about semantic preservation. As a first property, we consider transitivity which is important to reason about different components of function networks that communicate via shared events.

Lemma 3.3.1 (Transitivity of Causality)

$$\begin{aligned} &(1)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\}[cond_2] \ \wedge \\ &(2)\ \{f_1, ..., f_m\}[cond_2] \xrightarrow{[min_2, max_2]} \{g_1, ..., g_r\}[cond_3] \\ \Longrightarrow\ &(3)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1+min_2, max_1+max_2]} \{g_1, ..., g_r\}[cond_3] \end{aligned}$$

Proof: see Lemma Lemma A.2.1 in the appendix on page 210. $\square$

In some situations transitivity cannot be applied directly because the intermediate condition $cond_2$ cannot be immediately derived from the first causality pattern. But if we know that the condition holds, expressed in terms of a condition pattern, we can nevertheless apply transitivity by considering this external condition.

Lemma 3.3.2 (Transitivity of Causality with External Conditions)

$$
\begin{aligned}
&(1)\ \{e_1,...,e_n\}[cond_1] \xrightarrow{[min_1,max_1]} \{f_1,...,f_m\} \ \wedge \\
&(2)\ \{f_1,...,f_m\}[cond_2] \xrightarrow{[min_2,max_2]} \{g_1,...,g_r\} \ \wedge \\
&(3)\ \forall i \in \{1,...,n\}, j \in \{1,...,m\} :\ [cond_2]\ \textit{holds during}\ [e_i, f_j] \\
\Longrightarrow\ &(4)\ \{e_1,...,e_n\}[cond_1] \xrightarrow{[min_1+min_2,max_1+max_2]} \{g_1,...,g_r\}
\end{aligned}
$$

Proof: see Lemma A.2.2 in the appendix on page 211. □

If we have the situation that a causal dependency holds for different conditions, this can be expressed in a single causality pattern where the condition is the disjunction of all single conditions. For example, this is the case if a function node shows the same causal behavior in different states.

Lemma 3.3.3 (Combination of Conditions in Causality Patterns)

$$
\begin{aligned}
&(1)\ \{e_1,...,e_n\}[cond_1] \xrightarrow{[min_1,max_1]} \{f_1,...,f_m\} \ \wedge\ ... \ \wedge \\
&\quad \{e_1,...,e_n\}[cond_k] \xrightarrow{[max_k,min_k]} \{f_1,...,f_m\} \\
\Longrightarrow\ &(2)\ \{e_1,...,e_n\}[cond_1 \vee ... \vee cond_k] \xrightarrow{[min',max']} \{f_1,...,f_m\} \\
&\quad \textit{where}\ min' = \min(min_1,...,min_k),\ max' = \max(max_1,...,max_k)
\end{aligned}
$$

Proof: see Lemma A.2.3 in the appendix on page 211. □

This concludes the preparation for the semantics definition of function networks. We have now defined all patterns that we need to show the properties of function network components to be able to reason about semantics preservation and boundedness.

3.3.2. Basic Function Network Components

We define semantics of basic function networks in terms of timed automata. Timed automata are a well-established formalism to describe real-time systems and there exist sophisticated analysis tools such as UPPAAL [5] to show certain properties for these systems. How these tools may be applied to function networks has already been shown in [16]. For each element of a basic function network, we will define a timed automaton representation that synchronizes with the automata of the other components via synchronization events using the $e?$ and $e!$ notation. Furthermore, we show semantic properties for each component by using the previously defined patterns and establish the basis to reason about boundedness of function networks. To achieve this, we will first derive causal dependencies for each single timed automaton component and then use the property of transitivity to connect them. The overall goal is to express the causal dependencies between input and output events of channels and function nodes including delays and state dependencies.

Let $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ be a basic function network. The semantics of bfn is defined as a composition of timed automata resulting from its source nodes, function nodes and channels. Ports are defined implicitly because each port is assigned to exactly one of the previously listed components. Each input port of a function node contains a synchronization buffer to store incoming events. Each buffer has a capacity that determines how many events may be stored at a time. Whether this capacity is *sufficient* to avoid a buffer overflow depends on the incoming event streams, which depends on the composition of function nodes with each other via channels. If there exists a finite capacity that is sufficient for each buffer, the function network is bounded. Otherwise the capacity must be infinite to avoid a buffer overflow and the function network is unbounded. To cover infinite capacities, we define $\mathbb{N}^\infty$ as the set of positive natural numbers $\mathbb{N}^+$ extended by an additional symbol ∞ that is greater than any element from $\mathbb{N}^+$ i.e., $\mathbb{N}^\infty = \mathbb{N}^+ \cup \{\infty\}$ such that $m < \infty$ for all $m \in \mathbb{N}^+$. Hence, for a finite capacity it holds that $c \in \mathbb{N}^+$, and for an infinite capacity we set c to the special symbol ∞.

We define semantics of a function network as a network of timed automata with a buffer capacity c for each buffer of the network. We will later show for a specific class of function networks how boundedness can be decided and how a finite and sufficient buffer capacity can be determined for bounded buffers.

$$\mathcal{T}(bfn, c) = \mathcal{T}(TA(\mathcal{C}) \parallel TA(\Phi) \parallel TA(\mathcal{F}, c))$$

where $TA(\mathcal{C}) = \prod_{c \in \mathcal{C}} TA(c)$, $TA(\Phi) = \prod_{\phi \in \Phi} TA(\phi)$, $TA(\mathcal{F}, c) = \prod_{f \in \mathcal{F}} TA(f, c)$.

Please note, that the set of events that is used in the automata definitions is not equivalent to the function network event set Σ. This has two reasons: First, function network events may occur at different ports e.g. a function node may receive an event e and may also send an event e. Thus, events become first unique if we also consider the ports. Accordingly, the events that occur in the timed automata definitions are comprised of an event $\sigma \in \Sigma$ and a port $p \in \mathcal{P}$ with $\sigma \in \Sigma(p)$ denoted as $p.\sigma$. Second, sometimes intermediate events are needed to synchronize with other timed automata, which further extends the set of events used in the semantics definitions.

As an abbreviation for a certain sequence of transitions where events are received and sent, we will use a similar notion as it known for finite state machines, which we call *complex transition*. The abbreviation is defined in Figure 3.20. On the left, a complex transition is shown. It is split by a '/' symbol, where the left part is a usual transition with a receiver event $e?$, a guard φ and a set of clock resets ϱ leading from state s to s'. The right part of the complex transition defines a set of urgent sender events $f_1!, .., f_n!$ produced by this transition. The meaning of this abbreviation is defined on the right of Figure 3.20 in terms of a a set of usual transitions and some intermediate states between s and s'. We denote such a complex transition as $s \xrightarrow{e?,\varphi,\varrho/f_1!,..,f_n!} s'$. Please note that we assume that the receivers of the urgent events $f_1, ..., f_n$ do not impose a (different) sequence on the same urgent events.

In the following, we define semantics of basic function network components and show their properties starting with the event source.

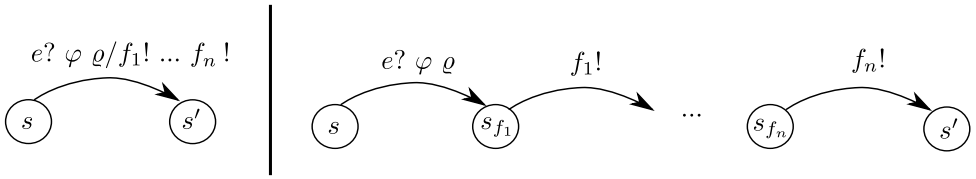


Figure 3.20.: Complex Transitions for Timed Automata

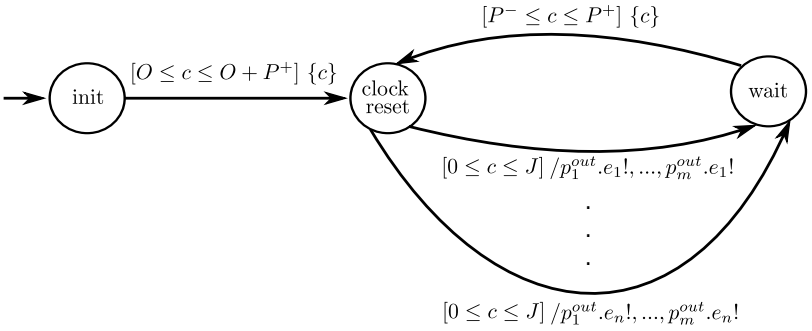


Figure 3.21.: Event Source Automaton $TA(\phi)$

Event Source

An event source $\phi = (EP, \mathcal{P}^{out}) \in \Phi$ is defined as a tuple of an event pattern $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ and a set of output ports $\mathcal{P}^{out}$. Its semantics is defined similar to the input interface automaton defined for task networks in [24] and is depicted in Figure 3.21. A difference to the referred interface automaton is that an event source may produce several output events simultaneously at different output ports while on each output port always the same event e_i is sent. This output event is chosen non-deterministically from the set Σ^{EP}. This is motivated by the fact that an event source models a part of the interface to the environment. Thus, such events may for example be sensor values of a distance sensor and are identical for each output port.

The automaton starts in the state 'init' and after a time interval of $[O, O + P^+]$ it takes a transition to the state 'clock reset' and resets the clock. After a further time interval of $[0, J]$, the next transition to the state 'wait' is taken, which fires an event $e_i \in \Sigma^{EP}$ at each output port $p_j^{out} \in \mathcal{P}^{out}$. Such an event is denoted as $p_j^{out}.e_i$. The next transition from state 'wait' to state 'clock reset' is taken in between P^- and P^+ time units after the previous clock reset. Thus, the time of the clock reset does not depend on the time where the event e_i has been emitted. When taking this transition, the clock is reset again. From here the automaton shows repetitive behavior by emitting the next event again within a time interval of $[0, J]$. The semantics of an event source is defined as follows:

Definition 3.3.4 (Event Source) *Let $\phi = (EP, \mathcal{P}^{out}) \in \Phi$ be an an event source where $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ is the output event pattern with $\Sigma^{EP} = \{e_1, ..., e_n\}$ and $J < P^-$ and $\mathcal{P}^{out} = \{p_1^{out}, ..., p_m^{out}\}$ is a set of m output ports. The semantics of ϕ is defined as the timed automaton $TA(\phi)$ depicted in Figure 3.21. Based on [24] this leads to the following language definition for each port p_j^{out}:*

$$\forall p_j^{out} \in \mathcal{P}^{out} : \; L(p_j^{out}) = \{ \; (\sigma_1, t_1 + \delta_1)(\sigma_2, t_2 + \delta_2)(\sigma_3, t_3 + \delta_3) \ldots \\ \mid \; \sigma_i \in \{p_j^{out}.e_1, ..., p_j^{out}.e_n\}, \; O \leq t_1 \leq O + P^+, \\ \forall i > 1 : \; t_i + P^- \leq t_{i+1} \leq t_i + P^+, \; 0 \leq \delta_i \leq J \; \}$$

Furthermore, it holds that:

$$\forall j, k \in \{1, ..., m\}, w_j \in L(p_j^{out}), w_k \in L(p_k^{out}) : \; (p_j^{out}.e_q, t) \in w_j \Longrightarrow (p_k^{out}.e_q, t) \in w_k.$$

◇

To be able to reason about input languages of channels and function nodes, we show that the language of the event source component is correctly abstracted by the language of the event pattern with the respective parameters. Please note, that in [24] this has already been done for an event model without offset. The start-up phase where the offset plays a role was excluded from the notion of equivalence referred to as *steady state equivalence*. Due to the possibility to also express an offset in event patterns, we can omit this restriction and also cover the start-up phase. This proof will also build the base for event pattern propagation which starts at the source nodes as we will discuss later.

Lemma 3.3.4 (Event Pattern of Event Source) *Let $\phi = (EP, \mathcal{P}^{out}) \in \Phi$ be an event source where $EP = (\Sigma^{EP}, P^-, P^+, J, O)$. Then the event pattern of each output port $p_j^{out} \in \mathcal{P}^{out}$ is a valid abstraction of the respective language i.e.*

$$\forall p_j^{out} \in \mathcal{P}^{out} : \; L(p_j^{out}) \subseteq L(ren(EP, \{p_j^{out}.\sigma \mid \; \sigma \in \Sigma(p_j^{out})\}))$$

Proof: see Lemma A.2.4 in the appendix on page 212. □

Basic Channel

A basic channel connects an output port of a function node or an event source with an input port of another (or the same) function node. Events are transmitted between the respective ports with a zero delay. Accordingly, semantics of a basic channel is defined as follows:

Definition 3.3.5 (Basic Channel) *A basic channel $c = (p_1, p_2) \in \mathcal{C}$ transmitting the event set $\Sigma(p_1)$=$\{e_1, ..., e_n\}$ is defined as the timed automaton $TA(c)$ depicted in Figure 3.22.* ◇

A basic channel forwards its input events to its output port with a zero delay. Thus, the events that may be observed at the output port are determined by the input port. This is expressed in the following causality pattern.

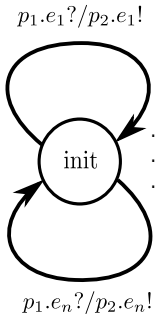


Figure 3.22.: Basic Channel Automaton $TA(c)$

Theorem 3.3.1 (Causality of Basic Channels) *Let $c = (p_1, p_2)$ be a basic channel with $\Sigma(p_1)$=$\{e_1, ..., e_n\}$. Then the following holds:*

$$\forall i \in \{1, ..., n\} : \; p_1.e_i \xrightarrow{[0,0]} p_2.e_i$$

Proof: This follows immediately from the complex transition init $\xrightarrow{p_1.e_i/p_2.e_i}$ init, which exists for each $i \in \{1, ..., n\}$. □

Function Node

The third and most complex element of a basic function network is a function node. The semantics of a function node f is given by a composition of timed automata and is divided into several subcomponents as shown in Figure 3.23.

A function node has a set of n input ports where each may have several incoming channels (but at least one). A function node is activated as soon as on each channel leading to the same input port an event has occurred that was not consumed by an activation yet. This means that for the i^{th} activation of an input port, we need the i^{th} occurrence of any event on each input channel. Due to this, we need a buffer in each input port that is able to store the already received events until all synchronization partners have arrived. The size of this *synchronization buffer* depends on the maximum waiting time between synchronizing events of an input port and may be determined by using event patterns as we will show later. Such a synchronization buffer exists for each input port and is denoted as $sync_i$ for an input port p_i^{in}.

As soon as all needed events for a synchronization have been observed, a synchronization event is produced by each synchronization buffer. The synchronization events of each input port are joined to activate the function node to a stream denoted as I_p. We define the execution semantics of function nodes such that there may be at most one function node execution at a time. This means that each function node is assumed to be executed on a single computation resource, which does not allow concurrent executions. Thus, transitions are always executed sequentially in the order of the arrival

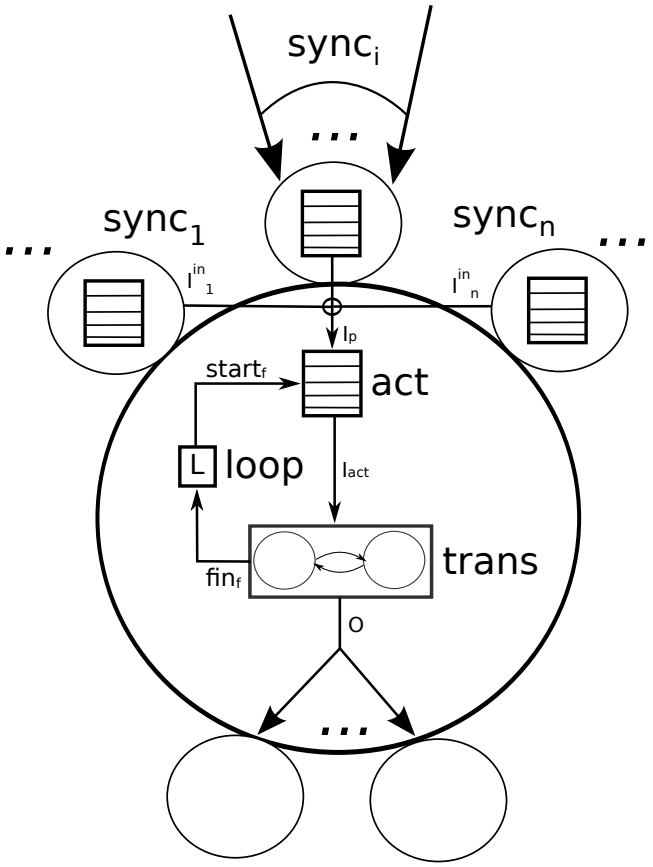


Figure 3.23.: Function Node Semantics - Overview

of their activation events. Hence, there may occur a synchronization event before the previous execution of the function node has finished and we need another buffer to store the activations denoted as *activation buffer act*. This buffer synchronizes activation events with so-called $start_f$ events. Those $start_f$ events are produced by another component named *loop*. A loop component initially produces a $start_f$ event at system start up. Each further $start_f$ event is produced first after the previous execution of the function node has been terminated denoted by an event fin_f.

Whenever a $start_f$ event and an input event are synchronized in the activation buffer (denoted as stream I_{act}), the actual function node execution starts, which is modeled by a transition system component *trans*. Depending on the current state and the input event, the transition system of the function node defines a set of output events that are produced at the output ports. For each output event an individual time delay is defined. Only if all output events of the currently executed transition have

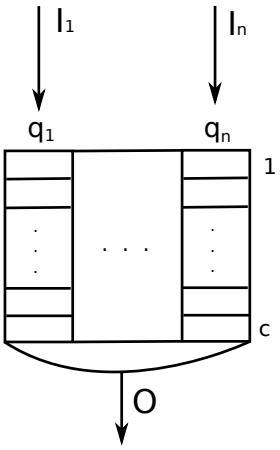


Figure 3.24.: Synchronization Buffer

occurred, the execution of the function node is terminated and a fin_f event is sent to the loop component. In the following, we will define all components of a function node and finally define how a function node is composed. We start with the definition and description of the synchronization buffer component.

Synchronization Buffer A synchronization buffer realizes an AND-activation and synchronizes a set of n input event streams. This means, that it waits on each input stream for the i^{th} occurrence of an event and then produces the i^{th} synchronization event. Each synchronization buffer has a capacity c determining how many events may be received on each stream before a synchronization event has to occur to avoid a buffer overflow. The capacity may also be infinite.

Thus, we can imagine a synchronization buffer as a set of n FIFO-Queues with c places as depicted in Figure 3.24. Each queue q_i receives and stores the events of the input stream I_i by attaching them at the end of the queue. A synchronization event is produced *immediately* when each queue contains at least one event. If this is the case, the first event in_i of each queue q_i is consumed and the synchronization event is produced. This assures that synchronization events are produced in the same order as their input events have been received leading to FIFO semantics. A synchronization event is denoted as the tuple of events that were consumed e.g. $(in_1, ..., in_n)$.

We will use this characterization to model the state space of the timed automaton that realizes the synchronization buffer. Each state is defined as a tuple $(q_1, ..., q_n)$ containing n FIFO queues. Each queue is again a tuple of events it currently stores. An empty queue is denoted as $q_i = <>$. In the initial state each queue is empty leading to $l^0 = (<> ... <>)$. If the automaton receives an input event in_i on stream I_i, this event is added to the queue q_i if there is any place left i.e., if the length of q_i is smaller than the capacity c. If a queue q_i is full, we have a buffer overflow and the automaton

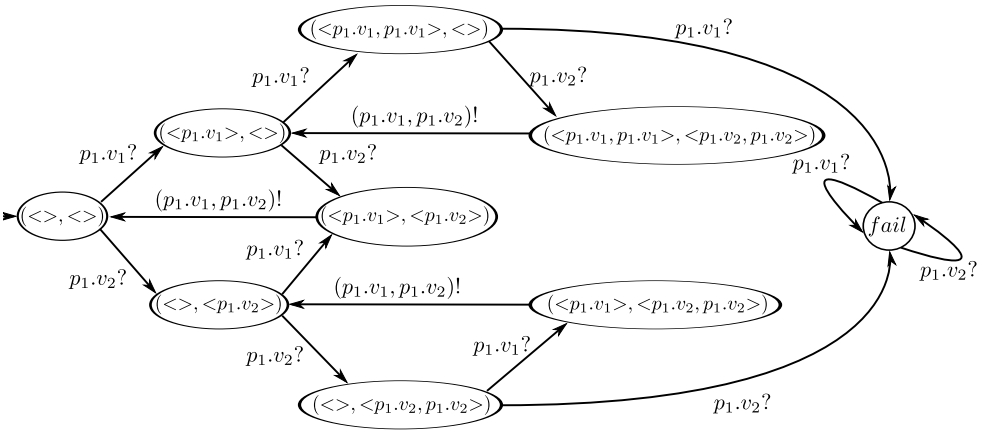


Figure 3.25.: Synchronization Buffer Automaton with capacity $c = 2$

reaches the state 'fail', which it cannot leave anymore. In the state 'fail', the automaton is still receptive to all input events to avoid deadlocks. As soon as there is at least one event contained in each queue, a synchronization event is produced immediately and the first event on each queue is consumed. Because this happens in zero time the automaton is always receptive for any input event.

In Figure 3.25, an example for a synchronization buffer automaton is depicted for the function node f_v from the cruise control example from Figure 3.7. The node f_v has two incoming channels transporting the events v_1 and v_2, which represent sensor values of redundant speed sensors. These events are synchronized at the input port p_1 to aggregate the single sensor values. This leads to the observable events $p_1.v_1$ and $p_1.v_2$. They are stored in two FIFO queues, one for each input channel. In the initial state, both queues are empty denoted as (<><>). We chose here an exemplary buffer capacity of two. This means that the automaton is able to receive up to two events per channel before a synchronization event $(p_1.v_1, p_1.v_2)$ must occur to avoid a buffer overflow, which is indicated by reaching the state 'fail'. We assume for this example that this capacity is sufficient and thus no buffer overflow occurs. Hence, also the waiting time for a synchronization partner is always finite. This is however only assured because the two speed sensors deliver their values with the *same* period. How we can decide boundedness and determine sufficient and finite buffer sizes for bounded buffers is discussed in Section 3.4 in detail.

The synchronization buffer component is defined as follows:

Definition 3.3.6 (Synchronization Buffer) *Let $I = I_1 \cup ... \cup I_n$ be n sets of input events and $c \in \mathbb{N}^\infty$ a capacity. The set of output events is defined as $O = \{(in_1, ..., in_n) \mid in_1 \in I_1, ..., in_n \in I_n\}$. The synchronization buffer component is defined as the timed automaton $Sync(I_1, ..., I_n, c) := (L, L^c, l^0, \Sigma, \Sigma^u, C, Inv, R)$ where*

- $L = \{fail\} \cup \Big\{(q_1, ..., q_n) \mid$

$$\forall i \in \{1, ..., n\} : \; q_i = <\sigma_1, ..., \sigma_k> | \; \sigma_r \in I_i \; \forall r \in \{1, ..., k\},$$
$$k \in \begin{cases} \{1, ..., c\} & , if \; c \in \mathbb{N}^+ \\ \mathbb{N}^+ & , if \; c = \infty \end{cases}$$
$$\vee \; q_i = <> \;\; denoting \; an \; empty \; queue. \qquad \Big\},$$

- $L^c = \emptyset$, $l^0 = (<>, ..., <>)$,
- $\Sigma = I \cup O$, $\Sigma^u = O$, $C = \emptyset$,
- $Inv = \{l \rightarrow true \mid l \in L\}$,
- $R = R_{rec} \cup R_{sync} \cup R_{fail} \cup R_{fail2}$ *where*
  - R_{rec} *is the set of transitions that receive an input event on an input stream and the respective FIFO queue is not full and there is no synchronization event to be produced*

$$R_{rec} = \; \{ \; ((q_1, ..., q_n), in_i?, true, \emptyset, (q_1, ..., q_i', ..., q_n)) \mid$$
$$\exists j : q_j = <>, in_i \in I_i,$$
$$q_i = <\sigma_1, ..., \sigma_k>, 0 \leq k < c,$$
$$q_i' = <\sigma_1, ..., \sigma_k, in_i> \; \},$$

  - R_{sync} *is the set of transitions that produce a synchronization event as soon as on each input stream an event has occurred i.e. no FIFO queue is empty*

$$R_{sync} = \; \{ \; ((q_1, ..., q_n), (in_1, ..., in_n)!, true, \emptyset, (q_1', ..., q_n')) \mid$$
$$\nexists j : q_j = <>,$$
$$\forall i \in \{1, ..., n\} : q_i = <\sigma_1^i = in_i, ..., \sigma_k^i>, q_i' = <\sigma_2^i, ..., \sigma_k^i>\},$$

  - R_{fail} *is the set of transitions that receive an input event on an input stream and the respective FIFO queue is full leading to the 'fail' state*

$$R_{fail} = \; \{ \; ((q_1, ..., q_n), in_i?, true, \emptyset, fail) \mid$$
$$in_i \in I_i, q_i = <\sigma_1, ..., \sigma_k>, k = c \; \},$$

  - R_{fail2} *is the set of self-transitions of the 'fail' state accepting each input symbol* $in \in I$ *i.e.*

$$R_{fail2} = \; \{ \; (fail, in?, true, \emptyset, fail) \mid \; in \in I\}.$$

We denote the set of output events as $O(Sync(I_1, ..., I_n, c)) := O$. ◇

To be able to reason about causal properties between input events and synchronization events, we assume that the synchronization buffer has a finite capacity $c \in \mathbb{N}^+$ and does not overflow i.e., it does not reach the 'fail' state. As soon as this happens, the automaton still receives input events but will never produce a synchronization event. Under which conditions a synchronization buffer does not overflow will be discussed in the part about boundedness of function networks in Section 3.4.

As a first causal property of a synchronization buffer, we will show that after an input event in_i has occurred on each input stream I_i, the respective synchronization event $(in_1, ..., in_n)$ can be immediately observed within a zero time delay.

Lemma 3.3.5 (Synchronization Buffer Causality) *Let us assume a synchronization buffer $Sync(I_1, ..., I_n, c)$ with a finite capacity $c \in \mathbb{N}^+$ that never reaches the 'fail' state. Then the following holds:*

$$\{in_1, ..., in_n\} \xrightarrow{[0,0]} (in_1, ..., in_n)$$
$$\text{where } \forall i \in \{1, ..., n\} : \ in_i \in I_i$$

Proof: The state where the synchronization buffer has received an event in_i on each input stream I_i is characterized by the state $(q_1, ..., q_n)$ where

$$\forall i \in \{1, ..., n\} : q_i = (\sigma_1 = in_i, ..., \sigma_k), \ k > 0.$$

For each such state there exists a transition $((q_1, ..., q_n) \xrightarrow{(in_1,...,in_n)!,true,\emptyset} (q'_1, ..., q'_n))$ $\in R_{sync}$, which produces the synchronization event $(in_1, ..., in_n)$ within a time interval of zero time units. □

In the next lemma, we focus on a single event in_j observed on a stream I_j and show how long it takes until there occur input events on each remaining input stream leading to a synchronization event that contains in_j. With $wait^-_{in}$ and $wait^+_{in}$ we denote the minimum and maximum delay we have to wait for an input event in_i on any stream I_i with $i \in \{1, ..., n\}$. This delay is always bounded if the buffer has a finite capacity and never reaches the 'fail' state.

Lemma 3.3.6 (Synchronization Buffer Causality - Single Events) *Let us assume a synchronization buffer $Sync(I_1, ..., I_n, c)$ with a finite capacity $c \in \mathbb{N}^+$ that never reaches the 'fail' state, $L(I_1), ..., L(I_n)$ be timed languages over the input event sets $I_1, ..., I_n$, and $in_j \in I_j$ be an event where $\exists w_j \in L(I_j) : (in_j, t_k) \in w_j, j \in \{1, ..., n\}$, $k \in \mathbb{N}^+$. Then the following holds:*

$$in_j \xrightarrow{[wait^-_{in}, wait^+_{in}]} (in_1, ..., in_j, ..., in_n),$$
$$\text{where } wait^-_{in} = \max_{i \neq j} \{\Delta^-(L(I_j), L(I_i))\},$$
$$wait^+_{in} = \max_{i \neq j} \{\Delta^+(L(I_j), L(I_i))\}$$

Proof: The minimum and maximum time we have to wait after the occurrence of (in_j, t_k) *until on each other input stream* i *with* $i \in \{1, ..., n\}$, $i \neq j$ *an event* $in_i \in I_i$ *with* $(in_i, s_k) \in w_i$, $w_i \in L(I_i)$ *has occurred is determined by*

$$wait_{in}^{-} = \max_{i \neq j}\{\Delta^{-}(L(I_j), L(I_i))\},\ wait_{in}^{+} = \max_{i \neq j}\{\Delta^{+}(L(I_j), L(I_i))\}.$$

From Lemma 3.3.5 we know that the synchronization event $(in_1, ..., in_j, ..., , in_n)$ *occurs immediately after we have seen such an event on each input stream which leads to the statement to prove.* □

Loop Component The *loop* component is part of a function node and is needed to assure that there will not be an activation of a function node before the previous execution has been finished. Semantics of the loop component is defined in terms of a timed automaton depicted in Figure 3.26. It initially produces an output event o and afterwards reacts on each received input event i by producing another output event o.

Definition 3.3.7 (Loop Component) *The loop component is defined as*

$$Loop(i, o)$$

where i *is an input event and* o *is an output event. Its semantics is defined by the timed automaton on the right of Figure 3.26.* ◇

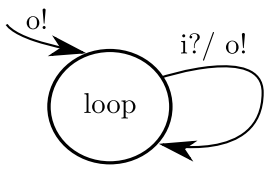


Figure 3.26.: Loop Automaton

The loop automaton accepts any arbitrary sequence of input symbols and is thus receptive. In the following lemma, we show that the loop component produces an output event o whenever it receives an input event i within a zero time delay.

Lemma 3.3.7 *Let* $Loop(i, o)$ *be a loop component. Then the following holds:*

$$i \xrightarrow{[0,0]} o$$

Proof: This immediately follows from the complex transition $loop \xrightarrow{i?/o!} loop$.

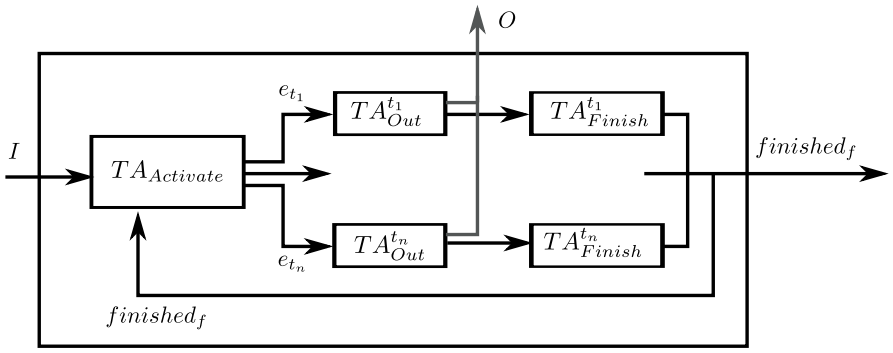


Figure 3.27.: Transition System Components

Transition System The transition system of a function node is modeled as a component that is called $Trans$. Its semantics is defined as a composition of timed automata as depicted in Figure 3.27. It consists of three different parts, which is the *activation automaton* called $TA_{Activate}$ and for each function node transition $t_k \in T$ an *output automata* composition named $TA^{t_k}_{Out}$ and a *finish automaton* named $TA^{t_k}_{Finish}$.

The first part in terms of the activation automaton $TA_{Activate}$ is depicted in Figure 3.28. It represents the execution states of a function node and thus either waits for an activation ('wait') or is executed ('exe'). The states of the transition system are represented by a state variable $state_f$ that initially holds the value s_0 for the initial state. For each function node transition $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$, there exists a transition in the activation automaton leading from the 'wait' state to the 'exe' state.

Due to the previous synchronization of input events at the input ports and a further synchronization with a $start_f$ event, the input events of the automaton are not equal to the events of the transitions in the transition system of the function node. Thus, there exists a mapping function M_{in} that maps the input port p_k and the input event set E_k of a transition t_k to the respective synchronization event in the timed automata representation. This mapping function is part of the interface of this component and is determined when composing a transition system with other components to build a function node. Accordingly, a transition in the automaton is taken if the mapped trigger event $M_{in}(p.E_k)$ of the corresponding function node transition occurs and the state variable $state_f$ has the needed value. At the same time, the $state_f$ variable is updated to the value s'_k. Additionally, an intermediate event e_{t_k} is produced that is received by the respective output automata composition $TA^{t_k}_{Out}$. The activation automaton returns to the initial state as soon as a fin_f event occurs. Furthermore, the automaton has a state 'fail', which is entered either if an input event is received in state 'exe' or a fin_f event occurs in state 'wait'. We will show later under which conditions the 'fail' state is never reached and that these conditions are satisfied when putting all components together to a function node.

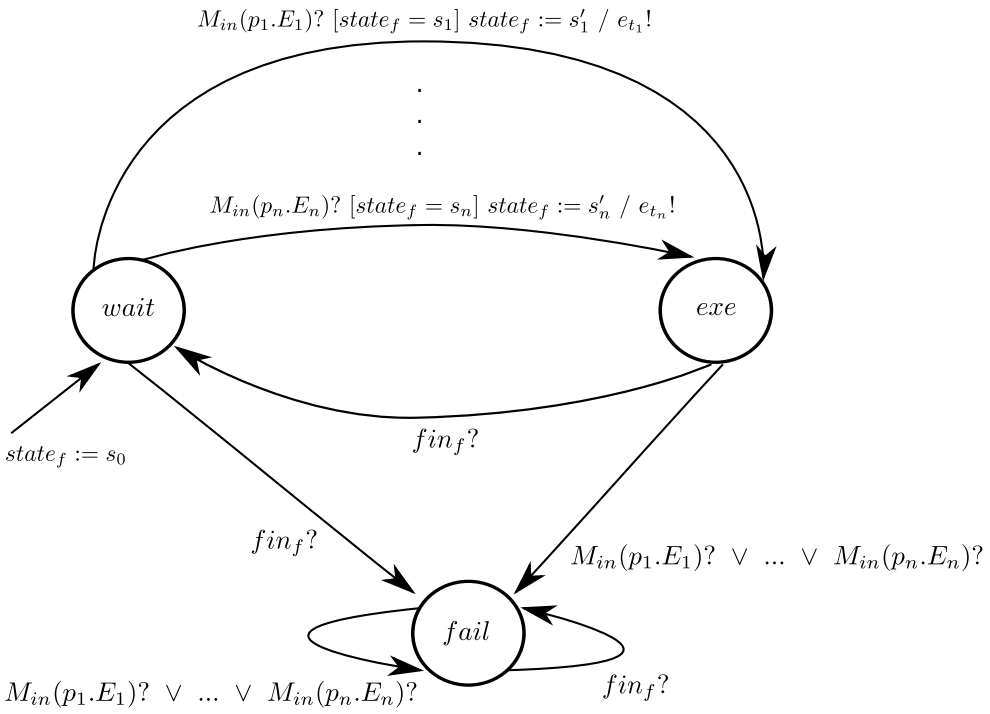


Figure 3.28.: Activation Automaton $TA_{Activate}$

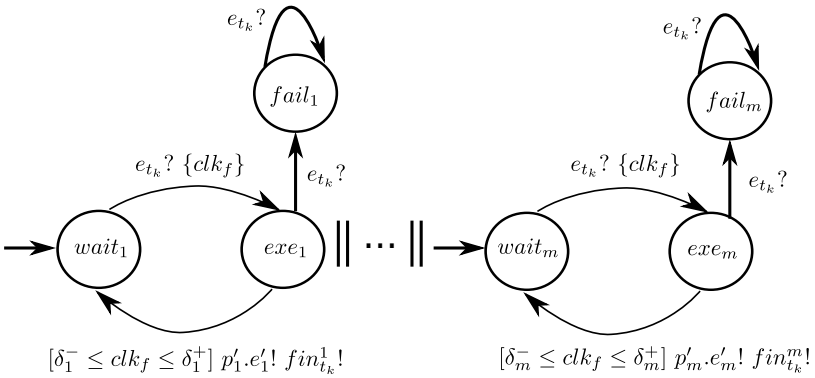


Figure 3.29.: Output Automata Composition $TA^{t_k}_{Out}$ for Transition t_k

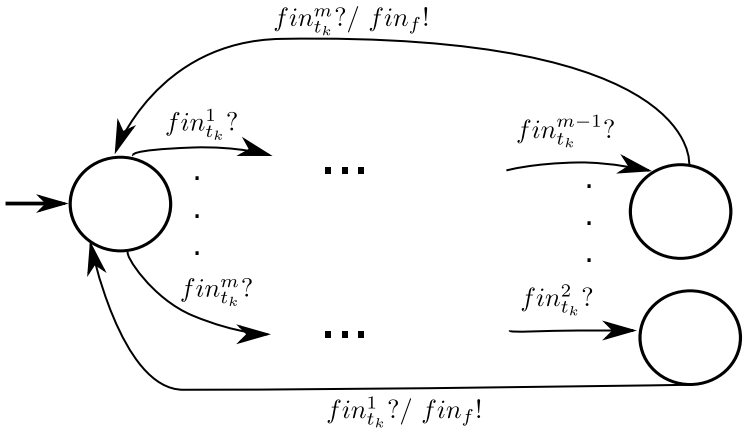


Figure 3.30.: Finish Automaton $TA^{t_k}_{Finish}$ for Transition t_k

The second part of the transition system is the output automata composition $TA^{t_k}_{Out}$ which is created for each transition $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$. Each $TA^{t_k}_{Out}$ consists of a set of m parallel automata as depicted in Figure 3.29. The parameter m is determined by the number of output specifications in Ψ_k. Each single automaton initially waits for the synchronization event e_{t_k} produced by the activation automaton and meaning that transition t_k has been activated. The clock clk_f is reset as soon as this transition is taken. It is sufficient to have one clock for all function node transitions because at a time only one function node transition can be active. Each automaton produces an output event $p'_j.e'_j$ after a delay of $[\delta^-_j, \delta^+_j]$. With the same automaton transition another event is produced named $fin^j_{t_k}$, which is consumed by the respective finish automaton $TA^{t_k}_{Finish}$. All output automata have again a state 'fail', which is shown to be never reached later.

The third part is the finish automaton $TA^{t_k}_{Finish}$, which is again created for each transition $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$. Such a finish automaton is shown in Figure 3.30. It waits for all events $fin^j_{t_k}$ produced by the output automata of the respective function node transition, which may occur in any order. Then it produces the event fin_f and returns to its initial state. The fin_f event is used to lead the activation automaton back to its initial state such that the whole execution process of a function node transition is completed and the system is ready to receive its next activation. This automaton also has a state 'fail', which is not depicted in Figure 3.30 for clarity. It is reached whenever an event $fin^j_{t_k}$ occurs twice before a fin_f event was produced. This is again excluded by the function node composition as we will show later. A transition system component is defined as a composition of these three automata types as follows:

Definition 3.3.8 (Transition System Component) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node with a transition system $\mathcal{A} = (S, s_0, T)$ where $T = \{t_1, ..., t_n\}$ is a set of transitions, I a set of input events and $M_{in} : \bigcup_{p \in \mathcal{P}^{in}} (p \times \Sigma^{act}(p) \to I)$ a function mapping input ports and events of the transition system to the respective input events in I. Let O denote the set of output events defined as $O = \{p'.e' \mid \exists t = (p, E, s \to \Psi, s') \in T : (p', e', \delta) \in \Psi\}$. The semantics of a transition system component is defined by the following composition of timed automata:*

$$Trans(I, f, M_{in}) = TA_{activate} \parallel \prod_{t_k \in T} (TA^{t_k}_{Out} \parallel TA^{t_k}_{Finish})$$

where

1. *$TA_{Activate}$ is the Activation Automaton as defined in Figure 3.28.*
2. *for each $t_k = (p_k, E_k, s_k \to \Psi_k, s'_k) \in T$ with $\Psi = \{(p'_1, e'_1, \delta_1), ..., (p'_m, e'_m, \delta_m)\}$*
   a) *there is a composition of Output Automata $TA^{t_k}_{Out}$ as defined in Figure 3.29,*
   b) *there is one Finish Automaton $TA^{t_k}_{Finish}$ as defined in Figure 3.30. Additionally, there exists a state 'fail' and for each state l without a transition receiving an event $\sigma \in \{fin^1_{t_k}, ..., fin^m_{t_k}\}$, there exists a transition $(l, \sigma, true, \emptyset, fail)$ including the state 'fail' itself.*

◇

As for the synchronization buffer, we also need to show for the transition system that it never reaches a 'fail' state. This is satisfied if no automaton of the transition system component ever reaches its 'fail' state. We will first show that this is true under the assumption that input events only occur when $TA_{Activate}$ is in the 'wait' state. Afterwards, we will prove that this assumption is always satisfied when the transition system is used to construct a function node.

Lemma 3.3.8 (Transition System never reaches Fail State) *A transition system component $Trans(I, f, M_{in})$ never reaches a 'fail' state if input events $M_{in}(p_k.E_k)$ only occur when $TA_{activate}$ is in the 'wait' state.*

Proof: To show this property, we have to show that each automaton of the transition system never reaches its 'fail' state under the given assumption.

1. *The activation automaton $TA_{activate}$ from Figure 3.28 gets into the 'fail' state if either an input event $M_{in}(p_k.E_k)$ occurs when it is in state 'exe' or a fin_f event occurs in state 'wait'. The first case is excluded by the assumption. Concerning the second case, a fin_f event is only produced by one of the $TA^{t_k}_{Finish}$ automata ($k \in \{1, ..., n\}$) after all events $fin^1_{t_k}, ..., fin^m_{t_k}$ of a function node transition have occurred. From $TA^{t_k}_{Out}$ we know that this can only happen after an event e_{t_k} has occurred before. This event again needs an input event $M_{in}(p_k.E_k)$ to occur, which implies that $TA_{activate}$ is in the 'exe' state. Thus, a fin_f event can only occur in the state 'exe' and not in the state 'wait'.*

2. *Each single output automaton from $TA^{t_k}_{Out}$ (see Figure 3.29) gets into its 'fail' state if an event e_{t_k} occurs when it is not in its initial state. From $TA_{activate}$ we know that an event e_{t_k} can only occur together with an input event $M_{in}(p_k.E_k)$. From the assumption we know that this only happens if $TA_{activate}$ is in the state 'wait'. To get into this state, a fin_f event has to occur. From the finish automaton $TA^{t_k}_{Finish}$ we know that this only happens if all events $fin^1_{t_k}, ..., fin^m_{t_k}$ have occurred, which implies that each output automaton has returned into its initial state. Thus, each output automaton never reaches its 'fail' state.*

3. *A finish automaton $TA^{t_k}_{Finish}$ (see Figure 3.30) gets into its 'fail' state if an event from $fin^1_{t_k}, ..., fin^m_{t_k}$ occurs twice before a fin_f event has been produced, which leads back to the initial state where it cannot get into the 'fail' state. We can exclude this again from the assumption: It always needs an input event $M_{in}(p_k.E_k)$ to produce the events $fin^1_{t_k}, ..., fin^m_{t_k}$ and $M_{in}(p_k.E_k)$ only occurs after the previous fin_f event has occurred. Thus, no event $fin^1_{t_k}, ..., fin^m_{t_k}$ can occur twice before a fin_f event has occurred and the finish automaton cannot reach its 'fail' state.*

□

Under the assumption that a transition system component never reaches its 'fail' state, we will show now the causal dependencies induced by the different automata components. It starts with proving for each single automaton which causality patterns can be derived from their transitions. Afterwards, these causality patterns are transitively combined to make statements from input events to output events of the transition system. The activation automaton produces immediately a synchronization event e_{t_k} whenever the transition t_k was triggered by an input event $M_{in}(p_k.E_k)$ and the state variable is set to s_k leading to the following causal dependency.

Lemma 3.3.9 (Activation Automaton Causality) *Let $TA_{Activate}$ be an activation automaton of a transition system $Trans(I, f, M_{in})$ with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$, $\mathcal{A} = (S, s_0, T)$ and $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$. Under the assumption that $TA_{Activate}$ never reaches its 'fail' state it holds*

$$M_{in}(p_k.E_k)[state_f = s_k] \xrightarrow{[0,0]} e_{t_k}[state_f = s'_k]$$

Proof: This causal dependency results immediately from the complex transition

$$wait \xrightarrow{M_{in}(p_k.E_k),[state_f = s_k],state_f := s'_k / e_{t_k}} exe$$

which implies that e_{t_k} is produced with a zero time delay.

□

The finish automaton waits for all events from the set $\{fin^1_{t_k}, ..., fin^m_{t_k}\}$ for a transition $trans_k$ to occur and then produces a fin_f event in a zero time interval.

Lemma 3.3.10 (Finish Automaton Causality) *Let $TA^{t_k}_{Finish}$ be a finish automaton of a transition system $Trans(I, f, M_{in})$ with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$, $\mathcal{A} = (S, s_0, T)$ and $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$ with $\Psi_k = \{(p'_1, e'_1, [\delta^-_1, \delta^+_1]), ..., (p'_m, e'_m, [\delta^-_m, \delta^+_m])\}$. Under the assumption that $TA^{t_k}_{Finish}$ never reaches its 'fail' state it holds*

$$\{fin^1_{t_k}, ..., fin^m_{t_k}\} \xrightarrow{[0,0]} fin_f$$

Proof: The finish automaton from Figure 3.30 is defined almost similar to the observer automaton of this pattern. It first receives all events $\{fin^1_{t_k}, ..., fin^m_{t_k}\}$ in any order and immediately returns to the initial state with the reception of the last event by producing the event fin_f in a zero time delay. □

For the output automata of a transition t_k holds that it produces output events $p'_i.e'_i$ with a delay δ_i after the synchronization event e_{t_k} has occurred. Together with each output event another synchronization event $fin^i_{t_k}$ is produced, which is needed for the finish automaton of that transition. Both is shown in the next lemma.

Lemma 3.3.11 (Output Automata Causality) *Let $TA^{t_k}_{Out}$ be a set of output automata of a transition system $Trans(I, f, M_{in})$ with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$, $\mathcal{A} = (S, s_0, T)$ and $t_k \in T$. Under the assumption that $TA^{t_k}_{Out}$ never reaches its 'fail' state for each $(p'_i, e'_i, \delta_i) \in \Psi$ with $\delta_i = [\delta^-_i, \delta^+_i]$ all the following holds:*

1. $$e_{t_k} \xrightarrow{\delta_i} p'_i.e'_i$$

   Proof: The causality pattern states that after the event e_{t_k} has been observed, the output event $p'_i.e'_i$ must occur between δ^-_i and δ^+_i time units. This is assured by the i^{th} automaton of the output automata (see Figure 3.29) by the succeeding transitions $wait_i \xrightarrow{e_{t_k}?,\{clk_f\}} exe_i$ and $exe_i \xrightarrow{[\delta^-_i \leq clk_f \leq \delta^+_i]/p'_i.e'_i!, fin^i_{t_k}} wait_i$.

2. $$p'_i.e'_i \xrightarrow{[0,0]} fin^i_{t_k}$$

   Proof: Follows immediately from the complex transition

   $$exe_i \xrightarrow{[\delta^-_i \leq c \leq \delta^+_i]/p'_i.e'_i!, fin^i_{t_k}!} wait_i$$

   in the i^{th} output automaton of Figure 3.29. □

Please note, that we know from the automata definitions that there exists no other causal dependency leading to an output event $p'_i.e'_i$ other than the one shown in Lemma 3.3.11. In the next lemma, we transitively combine the causal dependencies of the single automata to capture the causal behavior of a complete transition system. The lemma has two parts. The first part shows that each input event that occurs in a specific state leads to a set of output events according to the respective transition. The second part abstracts from concrete output events but refers to fin_f events, which are produced after the delay resulting from the minimum and maximum delay of all output delays of a transition.

Lemma 3.3.12 (Transition System Causality) *Let $Trans(I, f, M_{in})$ be a transition system with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$, $\mathcal{A} = (S, s_0, T)$ and $t_k = (p_k, E_k, s_k \rightarrow \Psi_k, s'_k) \in T$ with $\Psi_k = \{(p'_1, e'_1, [\delta_1^-, \delta_1^+]), ..., (p'_m, e'_m, [\delta_m^-, \delta_m^+])\}$. Under the assumption that no automaton of the transition system reaches its 'fail' state all the following holds:*

1.

$$\forall (p'_i, e'_i, [\delta_i^-, \delta_i^+]) \in \Psi : \; M_{in}(p_k.E_k)\ [state_f = s_k] \xrightarrow{[\delta_i^-, \delta_i^+]} p'_i.e'_i[state_f = s'_k]$$

Proof:

a) From Lemma 3.3.9 and Lemma 3.3.11 we know that

$$M_{in}(p_k.E_k)[state_f = s_k] \xrightarrow{[0,0]} e_{t_k}[state_f = s'_k] \;\wedge\; e_{t_k} \xrightarrow{\delta_i} p'_i.e'_i.$$

b) From the activation automaton we know that $[state_f = s_k]$ holds during $[e_{t_k}, fin_f]$ because the state may only change when receiving an input event. With $p'_i.e'_i \xrightarrow{[0,0]} fin^i_{t_k}$ (Lemma 3.3.11) and $\{fin^1_{t_k}, ..., fin^m_{t_k}\} \xrightarrow{[0,0]} fin_f$ (Lemma 3.3.10) we can conclude

$$[state_f = s_k] \text{ holds during } [e_{t_k}, p'_i.e'_i].$$

From a) and b) the statement to prove follows by transitivity with external conditions (Lemma 3.3.2).

2.

$$M_{in}(p_k.E_k)\ [state_f = s_k] \xrightarrow{[\delta_k^{min}, \delta_k^{max}]} fin_f$$
$$\text{where } \delta_k^{min} = \max(\delta_1^-, ..., \delta_m^-),\ \delta_k^{max} = \max(\delta_1^+, ..., \delta_m^+)$$

Proof: In Lemma 3.3.10 it has been shown that $\{fin^1_{t_k}, ..., fin^m_{t_k}\} \xrightarrow{[0,0]} fin_f$ and from Lemma 3.3.11 we know that $\forall i \in \{1, ..., m\} : p'_i.e'_i \xrightarrow{[0,0]} fin^i_{t_k}$ holds leading to $\{p'_1.e'_1, ..., p'_1.e'_1\} \xrightarrow{[0,0]} fin_f$. This means that fin_f occurs together with the last output event. The last output event of the transition t_k occurs at the latest at $\delta_k^{max} = \max(\delta_1^+, ..., \delta_m^+)$ and not before $\delta_k^{min} = \max(\delta_1^-, ..., \delta_m^-)$ leading to the statement to prove.

□

Based on the second part of the previous lemma, we will show now that a transition system produces a fin_f event after each occurrence of any input event independently from the state. Thus, we consider here the complete set of transitions and use the property of function nodes that there must exist a transition for each combination of input events and state. We denote the minimum and maximum delays that occur in any output specification of any transition as δ^{min} and δ^{max}.

Lemma 3.3.13 (Transition System Delay Bounds) *Let $Trans(I, f, M_{in})$ be a transition system with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$, $\mathcal{A} = (S, s_0, T)$, $T = \{t_1, ..., t_n\}$ and $\forall k \in \{1, ..., n\} : t_k = (p_k, E_k, s_k \to \Psi_k, s'_k)$ with minimum and maximum output delays δ_k^{min} and δ_k^{max}. Under the assumption that no automaton of the transition system ever reaches its 'fail' state the following holds:*

$$\forall k \in \{1, ..., n\} : M_{in}(p_k.E_k) \xrightarrow{[\delta^{min}, \delta^{max}]} fin_f$$
$$\text{where } \delta^{min} = \min(\delta_1^{min}, ..., \delta_n^{min}), \delta^{max} = \max(\delta_k^{max}, ..., \delta_n^{max})$$

Proof: From Lemma 3.3.12 we know that it holds

$$\forall k \in \{1, ..., n\} : M_{in}(p_k.E_k)\ [state_f = s_k] \xrightarrow{[\delta_k^{min}, \delta_k^{max}]} fin_f.$$

From the basic function network definition in Def. 3.2.1 we know that there exists a transition for each combination of input events and states. Thus, for each input symbol a fin_f event is produced after certain delay. This delay is determined by the minimum and maximum delay of any transition of the transition system leading to

$$\forall k \in \{1, ..., n\} : M_{in}(p_k.E_k) \xrightarrow{[\delta^{min}, \delta^{max}]} fin_f$$
$$\text{where } \delta^{min} = \min(\delta_1^{min}, ..., \delta_n^{min}), \delta^{max} = \max(\delta_k^{max}, ..., \delta_n^{max})$$

□

Function Node Composition A function node is defined as a composition of the previously defined components as shown in Figure 3.23. Each input port contains a synchronization buffer component to synchronize the events of all incoming channels leading to a further synchronization buffer component called activation buffer. Here, the input events are synchronized with $start_f$ events, which are produced by a loop component. The output events of the activation buffer are received by the transition system component, which produces events at output ports and a $finished_f$ event. The mapping function M_{in} is defined such that each set of activation events of a transition is mapped to the respective synchronization event that is produced by the activation buffer. This is determined by considering the causal dependencies induced by the activation buffer leading to the following definition.

Definition 3.3.9 (Function Node Component) *Let $c \in \mathbb{N}^\infty$ be a capacity. A function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ with n input ports $\mathcal{P}^{in} = p_1^{in}, ..., p_n^{in}$ is defined as a composition of timed automata components*

$$TA(f, c) = \prod_{i=1}^{n} sync_i^{in} \parallel act \parallel trans \parallel loop, \text{ where}$$

1. *for each input port $p_i^{in} \in \mathcal{P}^{in}$ with k incoming channels $\{c_1, ..., c_k\}$ with $c_j = (p_j^*, p_i^{in})$ and the corresponding sets of input events $I_1, ..., I_k$ with $I_j = \{p_i^{in}.e \mid e \in \Sigma(c_j)\}$ a synchronization buffer component $sync_i^{in}$ is defined as*

$$sync_i^{in} := Sync(I_1, ..., I_k, c)$$

2. *an activation buffer for the function node f is defined as*

$$act := Sync(I_p, \{start_f\}, c)$$

where $I_p = O(sync_1^{in}) \cup ... \cup O(sync_n^{in})$.

3. *a transition system component*

$$trans := Trans(O(act), f, M_{in})$$

is defined to model the transition system $\mathcal{A}$ of f, where M_{in} is the input mapping function M_{in}, which is defined as follows:

$$\forall p^{in} \in \mathcal{P}^{in}, \{e_1, ..., e_n\} \in \Sigma^{act}(p^{in}) \mid \{p^{in}.e_1, ..., p^{in}.e_n\} \xrightarrow{\delta} (i, start_f) : \\ M_{in}(p^{in}, \{e_1, ..., e_n\}) := (i, start_f), \text{ where } i \in O(act)$$

4. *a loop component loop is defined as*

$$loop := Loop(fin_f, start_f)$$

◇

Before we are able show the causal dependencies resulting from the function node definition, we need to show under which conditions no automaton of the function node composition ever reaches a 'fail' state. For the transition system component we have shown that this only holds under a specific assumption. With the definition how a function node is composed, we can now resolve this assumption by showing that it always holds for function nodes. Thus, a function node never reaches a 'fail' state if all of its synchronization buffers (including the activation buffer) never reach their 'fail' state as shown in the following theorem.

Theorem 3.3.2 (Function Node never reaches Fail State) *Let f be a function node. Then the following holds:*

$$f \text{ never reaches 'fail' state} \iff \forall i : sync_i^{in} \text{ never reaches fail state} \wedge \\ act \text{ never reaches 'fail' state}$$

Proof: Because the loop component cannot get into a 'fail' state by definition, it remains to show that the transition system component $Trans(I, f, M_{in})$ never reaches a 'fail' state. Following Lemma 3.3.8, this is assured if an input event $M_{in}(p_k.E_k)$ only occurs when $TA_{Activate}$ is in the state 'wait'. We will prove that this assumption always holds by induction over the occurrences of $start_f$ events at points in time t_i.

- *Base Case i=1: The first $start_f$ event is determined by the loop component to occur at time $t_1 = 0$. From the activation buffer we know from Lemma 3.3.6 that*

an input event for the transition system can only occur if both a $start_f$ event and a synchronized input event has occurred after a delay of $wait_{in}$ i.e.,

$$start_f \xrightarrow{wait_{in}} (in, start_f)$$

with $(in, start_f) \in I$ and $\exists p_k.E_k : M_{in}(p_k.E_k) = (in, start_f)$. Because $TA_{Activate}$ cannot leave its initial state 'wait' without receiving an input event, it is always in the state 'wait' for the initial input event.

- *Inductive Step: We assume that the statement holds for the i^{th} $start_f$ event at t_i and we show that it will also hold for the succeeding $start_f$ event at t_{i+1}. Thus, $TA_{Activate}$ is in the 'wait' state when receiving the i^{th} input event and we can apply the causal dependency from Lemma 3.3.13 i.e.,*

$$\forall k \in \{1, ..., n\} : M_{in}(p_k.E_k) \xrightarrow{[\delta^{min}, \delta^{max}]} fin_f.$$

From the loop component we know from Lemma 3.3.7 that

$$fin_f \xrightarrow{[0,0]} start_f.$$

Because the loop component is the only component that produces $start_f$ events and only the initial $start_f$ event is produced without any causal dependency, it follows that the next $start_f$ event at t_{i+1} can first occur after the previous fin_f event has occurred. When receiving a fin_f event in state 'exe', $TA_{Activate}$ returns to the state 'wait' and remains there until the $i + 1^{th}$ input event occurs, which concludes the proof. □

With the help of all the previous definitions and proofs, we are now able to show the causal dependencies that hold for function nodes. For all the following considerations on causality, we assume a function node where each buffer has a sufficient finite capacity and never gets into a 'fail' state. In Section 3.4 about boundedness, we will show how we can assure this for a specific class of function networks by propagating event patterns through the function network beginning at the sources. This enables us to iteratively determine sufficient buffer sizes for each function node.

We start with showing the causal dependencies of input events for function nodes i.e., which input events lead to an activation of a function node and which delays may occur before an activation starts. To denote synchronization events at input ports we introduce the following abbreviations: Let p be an input port and let $E = \{in_1, ..., in_k\} \in \Sigma^{act}(p)$. Then we define $p.(in_1, ..., in_k) := (p.in_1, ..., p.in_k)$ and $p.(E) := p.(in_1, ..., in_k)$. Furthermore, we denote the delays where we wait for a $start_f$ event as '$wait_{start}$'. For clarity, we omit the index of $start_f$ as long as it is clear from the context which function node we refer to.

Theorem 3.3.3 (Function Node Input Causality) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node. For each input port $p \in \mathcal{P}^{in}$ with k incoming channels $c_1, ..., c_k$ where $\forall j \in \{1, ..., k\} : \exists c_j = (p'_j, p) \in \mathcal{C}$ the following holds:*

$$\forall j \in \{1, ..., k\}, in_j \in \Sigma(p'_j) :$$

(1) $p.in_j \xrightarrow{[wait^-_{in}, wait^+_{in}]} p.(in_1, ..., in_j ..., in_k)$,

(2) $p.(in_1, ..., in_j ..., in_k) \xrightarrow{[wait^-_{start}, wait^+_{start}]} (p.(in_1, ..., in_j ..., in_k), start_f)$

(3) $p.in_j \xrightarrow{[wait^-_{in} + wait^-_{start}, wait^+_{in} + wait^+_{start}]} (p.(in_1, ..., in_j ..., in_k), start_f)$

(4) $p'_j.in_j \xrightarrow{[wait^-_{in} + wait^-_{start}, wait^+_{in} + wait^+_{start}]} (p.(in_1, ..., in_j ..., in_k), start_f)$

(5) $\{p.in_1, ..., p.in_k\} \xrightarrow{0} p.(in_1, ..., in_k)$

Proof:

1. *(1) follows directly from the synchronization buffer of input port p and Lemma 3.3.6 and the abbreviation $p.(in_1, ..., in_k) = (p.in_1, ..., p.in_k)$.*

2. *(2) follows directly from the activation buffer act and Lemma 3.3.6.*

3. *(3) follows directly from (1) and (2) by transitivity.*

4. *From input channels we know by Theorem 3.3.1:*

$$\forall j \in \{1, ..., k\} : \; p'_j.in_j \xrightarrow{[0,0]} p.in_j.$$

   Together with (3), (4) follows by transitivity.

5. *(5) follows from Lemma 3.3.5 and the abbreviation*

$$p.(in_1, ..., in_k) = (p.in_1, ..., p.in_k).$$

□

The next theorem shows how the output behavior of a function node causally depends on the input and $start_f$ events and the state of the internal transition system. The loop component produces a $start_f$ event at system start-up to enable the first execution of a function node. When the execution is terminated, a fin_f event is sent and the loop component produces another $start_f$ event to release the next activation as soon as a synchronized input event has arrived. The $start_f$ event has an important role for the causality of a function node because it is needed to determine the state of a function node when an execution starts, which influences the output behavior. If $start_f$ events are not considered in a causality pattern, we cannot determine the output behavior in the general case. Nonetheless, there are special cases where this is possible, as we will see later when considering state-independent function nodes.

Theorem 3.3.4 (Function Node Output Causality) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node. For each input port $p \in \mathcal{P}^{in}$ and each transition $t_k = (p_k, E_k, s_k \to \Psi_k, s'_k) \in T$ with $(p'_j, e'_j, [\delta_j^-, \delta_j^+]) \in \Psi$ the following holds:*

$$(1)\ (p_k.(E_k), start_f)[state_f = s_k] \xrightarrow{[\delta_j^-, \delta_j^+]} p'_j.e'_j[state_f = s'_k]$$

$$(2)\ (p_k.(E_k), start_f)[state_f = s_k] \xrightarrow{[\delta_k^{min}, \delta_k^{max}]} fin_f$$

Proof:

1. *From the transition system we know from Lemma 3.3.12:*

$$M_{in}(p.E)\ [state_f = s] \xrightarrow{[\delta_j^-, \delta_j^+]} p'_j.e'_j[state_f = s'].$$

   From the definition of M_{in} in Def. 3.3.9 with $M_{in}(p.E) = (p_k.(E_k), start_f))$ we can conclude that statement (1) holds.

2. *From Lemma 3.3.13 we know that*

$$M_{in}(p_k.E_k) \xrightarrow{[\delta_k^{min}, \delta_k^{max}]} fin_f.$$

   From the definition of M_{in} in Def. 3.3.9 with $M_{in}(p.E) = (p_k.(E_k), start_f))$ we can conclude that statement (2) holds.

□

As the last part of causal dependencies for general function nodes, we consider the dependency of fin_f events from $start_f$ events and the time delay that may pass between them. This becomes important in the later sections when we show boundedness and it is of interest how long we may have to wait for the next start event to occur. This is essential to determine the needed capacity of the activation buffer. The time we have to wait for an input event from any input port to arrive at the activation buffer is denoted as $wait_{I_p}$, where I_p denotes the input stream of the activation buffer (according to Def. 3.3.9 and Figure 3.23).

Theorem 3.3.5 (Function Node Finish Causality) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node with $\mathcal{A} = (S, s_0, T)$ with $T = \{t_1, ..., t_n\}$ and I_p the set of input events for the activation buffer act as defined in Def. 3.3.9. Then it holds:*

$$start_f \xrightarrow{[wait_{I_p}^- + \delta^{min}, wait_{I_p}^+ + \delta^{max}]} fin_f$$

Proof: For the activation buffer act we know from Lemma 3.3.6 that it holds

$$start_f \xrightarrow{[wait_{I_p}^-, wait_{I_p}^+]} (i, start_f).$$

where $[wait^-_{I_p}, wait^+_{I_p}]$ *denotes the delay interval we have to wait for an input event on* I_p*. From Lemma 3.3.13 we know that for any* $M_{in}(p_k.E_k)$ *it holds*

$$M_{in}(p_k.E_k) \xrightarrow{[\delta^{min},\delta^{max}]} fin_f.$$

From Def. 3.3.9 we know that there exists a $p_k.E_k$ *with* $M_{in}(p_k.E_k) = (i, start_f)$ *leading with transitivity to*

$$start_f \xrightarrow{[wait^-_{I_p}+\delta^{min},wait^+_{I_p}+\delta^{max}]} fin_f.$$

□

From the statements about causality for general function nodes, we will now derive causal dependencies for function nodes with only one state. Here, we can omit the state dependency and thus also make statements without considering $start_f$ events.

Corollary 3.3.1 (Function Node Causality with One State) *Let* $f = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out})$ *be a function node where the set of states contains exactly one element* $S = \{s\}$*. For each transition* $t = (p, E, s \to \Psi, s') \in T$ *with* $s = s'$ *and for each* $(p', e', \delta) \in \Psi$ *with* $\delta = [\delta^-, \delta^+]$ *the following holds:*

$$(1)\ (p.(E), start_f) \xrightarrow{[\delta^-,\delta^+]} p'.e' \ \wedge$$
$$(2)\ p.(E) \xrightarrow{[wait^-_{start}+\delta^-,wait^+_{start}+\delta^+]} p'.e'$$

Proof: From $S = \{s_k\}$ *follows that* $[state_f = s_k]$ *holds during* $[p.(E), p'.e']$*.*

$$\overset{Theorem\ 3.3.4}{\Longrightarrow} (p.(E), start_f)[state_f = s_k] \xrightarrow{[\delta^-,\delta^+]} p'.e'[state_f = s_k]$$
$$\overset{S=\{s_k\}}{\Longrightarrow} (1)\ (p.(E), start_f) \xrightarrow{[\delta^-,\delta^+]} p'.e'$$
$$\overset{Theorem\ 3.3.3}{\Longrightarrow} (2)\ p.(E) \xrightarrow{[wait^-_{start}+\delta^-,wait^+_{start}+\delta^+]} p'.e'$$

□

The next theorem shows the causal dependencies for state-independent function nodes, which is essential to be able to apply event pattern propagation for this class of function nodes. Their causal behavior is similar to function nodes with only one state except that the delay between input and output events may vary.

Theorem 3.3.6 (Output Causality of State-Independent Nodes) *Let* f *be a function node with* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$*, where* $S = \{s_1, ..., s_m\}$*,* $p^{out} \in \mathcal{P}^{out}$*. Then the following holds:*

$$\forall s_j \in S : \exists t_j = (p^{in}, E, s_j \to \Psi_j, s'_j)$$
$$\textit{where } \exists \psi_j = (p^{out}, e', [\delta^-_j, \delta^+_j]) \in \Psi_j$$
$$\Longrightarrow\ (p^{in}.(E), start_f) \xrightarrow{[\min(\delta^-_1,...,\delta^-_m),\max(\delta^+_1,...,\delta^+_m)]} p^{out}.e'$$

Proof:

$$
\begin{aligned}
& \forall s_j \in S : \exists t_j = (p^{in}, E, s_j \to \Psi_j, s'_j) \\
& \quad \textit{where } \exists \psi_j = (p^{out}, e', [\delta_j^-, \delta_j^+]) \in \Psi_j \\
\overset{Theorem\ 3.3.4}{\Longrightarrow} \; & \forall s_j \in S : (p^{in}.(E), start_f)[state_f = s_j] \xrightarrow{[\delta_j^-, \delta_j^+]} p^{out}.e' \\
\Longleftrightarrow \; & (p^{in}.(E), start_f)[state_f = s_1] \xrightarrow{[\delta_1^-, \delta_1^+]} p^{out}.e' \wedge \ ... \wedge \\
& (p^{in}.(E), start_f)[state_f = s_m] \xrightarrow{[\delta_m^-, \delta_m^+]} p^{out}.e' \\
\overset{Lemma\ 3.3.3}{\Longrightarrow} \; & (p^{in}.(E), start_f)[state_f = s_1 \vee ... \vee state_f = s_m] \\
& \xrightarrow{[\min(\delta_1^-, ..., \delta_m^-), \max(\delta_1^+, ..., \delta_m^+)]} p^{out}.e'
\end{aligned}
$$

From $S = \{s_1, ..., s_m\}$ *follows* $(state_f = s_1 \vee ... \vee state_f = s_m) =$ *true and thus*

$$
(p^{in}.(E), start_f) \xrightarrow{[\min(\delta_1^-, ..., \delta_m^-), \max(\delta_1^+, ..., \delta_m^+)]} p^{out}.e'
$$

□

This concludes the definition of semantics of basic function network components and the causal dependencies they induce.

3.3.3. Extended Function Network Components

Based on the causal dependencies for components of basic function networks, we will now apply this knowledge to extended function network components. This is needed because task creation is defined for extended function networks and thus for showing semantics preservation, we also need to consider causality of these components. The causal dependencies can be derived from the translation of extended function networks to basic function networks as defined in Def. 3.2.3.

A signal data node is modeled as a simple function node with one state and one transition for each input event that may occur. We restrict to show the behavior when an input event and a $start_f$ event is available. The delays induced by waiting for a $start_f$ event can be determined in the same way as for general function nodes.

Corollary 3.3.2 (Causality of Signal Data Node) *For a signal data node* $d = (\mathcal{P}^{in}, \delta, \mathcal{P}^{out}) \in \mathcal{D}_{signal}$ *the following causal dependency holds:*

$$
\forall p^{in} \in \mathcal{P}^{in}, e \in \Sigma(p^{in}) : \ (p^{in}.e, start_{f_d}) \xrightarrow{\delta} p^{out}.e
$$

Proof: Follows immediately from Def. 3.2.3 and Corollary 3.3.1. □

Accordingly, we can show the causality that holds for a shared data node, which is also modeled as a function node. We are mainly interested in the causal dependencies

that arise when reading from the data node. Here, we will also consider the delay $wait_{start}$ that is needed to wait for a $start_f$ event to point out that also a reading process may be temporary blocked by a previous write or read access.

Corollary 3.3.3 (Causality of Shared Data Node) *According to Def. 3.2.3, a shared data node* $d = (\{p^{in}\}, \delta, \sigma_0, \{p^{out}\}) \in \mathcal{D}_{shared}$ *is translated into a function node* $f_d = (\{p^{in}, p_d^r\}, (S, s_0, T), \{p^{out}, p_d^{\perp}\}) \in \mathcal{F}_b$ *leading to the following causal dependency:*

$$\forall r \in \Sigma(p_d^r): \; p_d^r.r \xrightarrow{\delta + wait_{start}} p^{out}.\sigma, \; \sigma \in \Sigma(p^{in}),$$
$$where \; wait_{start} = [wait_{start}^-, wait_{start}^+]$$

Proof: From Def. 3.2.3 and Theorem 3.3.4 follows that

$$\forall s_\sigma \in S: \; (p_d^r.r, start_{f_d})[state_{f_d} = s_\sigma] \xrightarrow{\delta} p^{out}.\sigma[state_{f_d} = s_\sigma]$$

Together with Theorem 3.3.3 this leads to the statement to prove. □

For the FIFO data node the situation is quite similar as for the shared data node. The overall time we have to wait for reading an event from the FIFO is determined by the execution delay δ of the data node and the waiting time for a $start_f$ event, which is denoted as $wait_{start}$.

Corollary 3.3.4 (Causality of FIFO Data Node) *According to Def. 3.2.3 a FIFO data node* $d = (\{p^{in}\}, \delta, c, \{p^{out}\}) \in \mathcal{D}_{fifo}$ *is translated into a function node* $f_d = (\{p^{in}, p_d^r\}, (S, s_0, T), \{p^{out}, p_d^{\perp}\}) \in \mathcal{F}_b$ *leading to the following causal dependency:*

$$\forall r \in \Sigma(p_d^r): p_d^r.r \xrightarrow{\delta + wait_{start}} p^{out}.\sigma, \; \sigma \in \Sigma(p^{out}),$$
$$where \; wait_{start} = [wait_{start}^-, wait_{start}^+]$$

Proof: From Def. 3.2.3 and Theorem 3.3.4 the following holds:

(1) $\forall s_{\sigma_1,\sigma_2,\ldots,\sigma_k} \neq empty \in S:$

$(p_d^r.r, start_{f_d})[s_{f_d} = state_{\sigma_1,\sigma_2,\ldots,\sigma_k}] \xrightarrow{\delta} p^{out}.\sigma_1[state_{f_d} = s_{\sigma_2,\ldots,\sigma_k}],$

(2) $(p_d^r.r, start_{f_d})[state_{f_d} = empty] \xrightarrow{\delta} p^{out}.z[state_{f_d} = empty]$

Thus, for each state an event σ *is produced, if a read event occurs, which leads together with Theorem 3.3.3 to the statement to prove.* □

The last data node is the finite source data node, where we show that the intended causal behavior can only be assured under a specific condition stating that the input events arrive within a maximum time bound of $P^- - J$ time units.

Corollary 3.3.5 (Causality of Finite Source Data Node) *According to the extended function network definition from Def. 3.2.3, a finite source data node* $d = (\{p^{in}\}, \delta, EP, \{p^{out}\}) \in \mathcal{D}_{fsource}$ *with* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *is translated into a function node* $f_d = (\{p^{in}, p_d^{tr}\}, (S, s_0, T), \{p^{out}, p_d^{\perp}\})$. *This leads to the following causal*

dependency if the assumption holds that the delay between an output and input event is bounded by $P^- - J$:

$$\Delta(L(\Sigma^{EP}), L(\Sigma(p^{in}))) \leq P^- - J$$
$$\implies \forall e \in \Sigma^{EP} : \; p_d^{tr}.e \xrightarrow{\delta} p^{out}.e$$

Proof: see Corollary A.2.1 in the appendix on page 213. □

A further element of extended function networks are activation channels that are either translated to simple function nodes or to basic channels, which leads to the following causal dependencies.

Corollary 3.3.6 (Causality of Activation Channels) *An activation channel $c = (p^{out}, \delta, p^{in}) \in \mathcal{C}^A$ with $\delta = [\delta^-, \delta^+]$ is either translated to a function node f_c or a basic channel leading to the following causal dependency:*

$$\forall e \in \Sigma(p^{out}) : \begin{cases} p^{out}.e \xrightarrow{[0,0]} p^{in}.e & , \text{if } \delta^- = \delta^+ = 0 \\ p^{out}.e \xrightarrow{wait_{start}+\delta} p^{in}.e & , else \end{cases}$$

Proof: As defined in Def. 3.2.3, if $\delta^- = \delta^+ = 0$, c is translated into a basic channel $c_b = (p^{out}, p^{in}) \in \mathcal{C}_b$ leading with Theorem 3.3.1 to the statement of the first case. Otherwise c is translated into a function node $f_c = (\{p\}, (\{s_0\}, s_0, T), \{p'\}) \in \mathcal{F}_b$ and two basic channels $c_{out} = (p^{out}, p) \in \mathcal{C}_b$ and $c_{in} = (p', p^{in}) \in \mathcal{C}_b$ leading with Corollary 3.3.1 and Theorem 3.3.1 to the statement of the second case. □

This concludes the section about semantics of basic and extended function networks and provides us with all properties that are relevant to reason about the question of boundedness of function networks in the next section.

3.4. Boundedness and Event Pattern Propagation

First, the question arises why we need to deal with boundedness and its decidability at all. For this work, the reason can be found in the fact that we aim at implementing a function network on a hardware architecture. This always means that we have a limited set of resources e.g. in terms of memory. Thus, it has to be ensured that the network can be modeled with a finite set of states in order to be implementable. For function networks this means that there must exist a sufficient and finite capacity for each synchronization buffer.

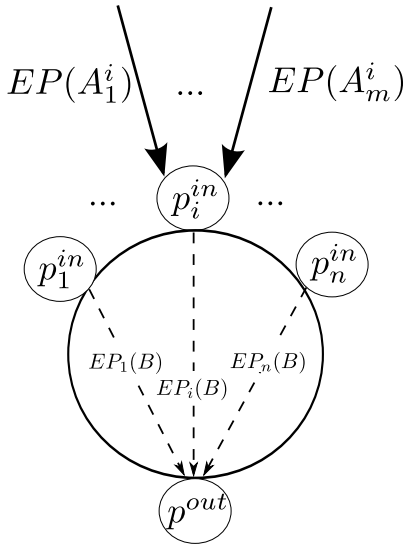
In Section 3.3, we have defined function network semantics depending on a capacity c for all synchronization buffers where the question remains how to choose a proper capacity. First, an infinite capacity $c = \infty$ would always assure that the 'fail' state is never reached for each buffer. But, such a function network might be unbounded and thus not implementable. To decide boundedness, we need to determine whether there also exists a finite capacity $c \in \mathbb{N}^+$ that is sufficient for each buffer to never reach its 'fail' state.

In the following, we will show that boundedness is decidable for a specific class of function networks, which we call *periodic state-independent* function networks. This class of function networks is based on two assumptions. First, we assume that each source node produces events with a periodic event pattern i.e., $P^- = P^+$. This is sufficient for the context of this work, because the specification models of Simulink only allow periodic activations. The second assumption is that all function nodes are state-independent. This property means that the causal output behavior of a function node does not depend on the state but only on the input events of the node. This property is also satisfied for function networks that arise from the translation of a Simulink specification model because the internal behavior of Stateflow blocks is not modeled explicitly. Furthermore, there is an exception in terms of a special function node called *period multiplier*, which is necessary to realize a valid translation from Simulink models but not state-independent. For this kind of node, we will show in Chapter 4 how output event patterns can be derived and how boundedness can be decided for those networks as well. Furthermore, we will show in Chapter 5 about task creation that the property of state-independence is preserved by the operations to merge function nodes.

Based on the proposed assumptions, we will define an algorithm that decides boundedness for periodic state-independent function networks. This algorithm is based on propagating event patterns through the network starting at the sources whose output event patterns are known initially. We will show how event patterns are propagated for basic function network components in terms of channels and state-independent function nodes. The translation of extended function networks to basic function networks enables us to apply these results also for extended function networks if the components are state-independent. Thus, we show in a second step under which condition the additional elements are state-independent and thus are part of the considered class of function networks.

3.4.1. Event Pattern Propagation

Event pattern propagation is based on the causal dependencies of function nodes and channels that have been shown in Section 3.3. Let us first assume a function node as depicted in Figure 3.31 with an output port p^{out}. For this output port we want to determine the event pattern of all events that may occur. We further assume that there exists a set of n input ports with causal dependencies to p^{out} and that all these causal dependencies are unconditional i.e., each occurrence of input events leads to an output event of a common event set. These assumptions hold for the class of state-independent function nodes as we will see later. We further assume that each input port p_i^{in} has m incoming channels leading to a set of event sets $A_1^i, ..., A_m^i$ that are synchronized at that port. The following lemma shows that the event patterns of all events that may occur at p^{out} (which we refer to as the set B) can be derived by considering all the event patterns of input ports with causal dependencies to p^{out}.



$$EP(B) = super(EP_1(B), ..., EP_n(B))$$

Figure 3.31.: Event Pattern Propagation using Causality Pattern

Lemma 3.4.1 (Event Pattern Propagation with Causality Pattern) *Let f be a function node with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ where $p_{out} \in \mathcal{P}^{out}$ is an output port with a set of output events $B := \Sigma(p^{out})$. Let further*

- *$\{p_1^{in}, ..., p_n^{in}\} \subseteq \mathcal{P}^{in}$ be the set of all input ports that have a causal dependency to events of the output port p^{out},*
- *for each input port p_i^{in} ($i \in \{1, ..., n\}$) with m incoming channels and a set of events arriving on these channels $A_1^i, ..., A_m^i$ hold*

$$\forall a_1^i \in A_1^i, ..., a_m^i \in A_m^i : \{p_i^{in}.a_1^i, ..., p_i^{in}.a_m^i\} \xrightarrow{[min_i, max_i]} p^{out}.b \text{ where } b \in B,$$

- *there exist period-equivalent event patterns for all these channels and their sets of input events i.e., $EP(A_1^i) \overset{P}{=} ... \overset{P}{=} EP(A_m^i)$,*

Then the event pattern $EP(B)$ for the output port p^{out} is determined as follows:

$$EP(B) = super(EP_1(B), ..., EP_n(B))$$
$$\text{where } EP_i(B) = ren(delay(sync(EP(A_1^i), ..., EP(A_m^i)), [min_i, max_i]), B)$$
$$\text{with } i \in \{1, ..., n\}$$

Proof: We know the event pattern for each incoming channel of each input port p_i^{in}, which are period-equivalent i.e., $EP(A_1^i) \overset{P}{=} ... \overset{P}{=} EP(A_m^i)$. A valid abstraction for the

occurrence of an event from each incoming channel $\{p_i^{in}.a_1^i, ..., p_i^{in}.a_m^i\}$ *is the synchronization of all the event patterns as shown in Lemma 3.1.7 leading to*

$$EP(\{p_i^{in}.a_1^i, ..., p_i^{in}.a_m^i\}) = sync(EP(A_1^i), ..., EP(A_m^i)).$$

From $\{p_i^{in}.a_1^i, ..., p_i^{in}.a_m^i\} \xrightarrow{[min_i, max_i]} p^{out}.b$ *we know that as soon as on each input channel an event has occurred, there occurs an event* $b \in B$ *within a time interval of* $[min_i, max_i]$*. From Lemma 3.1.9 we know that this language can be abstracted by the delay function for event patterns, which leads with the renaming function to*

$$EP_i(B) = ren(delay(sync(EP(A_1^i), ..., EP(A_m^i)), [min_i, max_i]), B).$$

The occurrence of an event $b \in B$ *can be abstracted by the superposition function as shown in Lemma 3.1.8 leading to the statement to prove i.e.,*

$$EP(B) = super(EP_1(B), ..., EP_n(B)).$$

□

For function nodes where the events of an output port p^{out} exclusively depend on a single input port $p^{in} \in \mathcal{P}^{in}$ with a set of events $A_1, ..., A_m$, the event pattern propagation can be simplified to a pure synchronization. This means, that we can omit the superposition of event patterns, which leads to

$$EP(B) = ren(delay(sync(EP(A_1), ..., EP(A_m)), [\min, \max]), B).$$

It becomes even more simpler if there exist only causal dependencies from a single input port $p^{in} \in \mathcal{P}^{in}$ with one incoming channel with a set of events A. In this case, also the synchronization can be omitted leading to

$$EP(B) = ren(delay(EP(A), [\min, \max]), B).$$

To be able to apply event pattern propagation to state-independent function nodes, it remains to show that a function node with a state-independent output port exactly leads to the assumptions of Lemma 3.4.1. Thus, event pattern propagation can be applied for all function node output ports that satisfy the property of state-independence.

Lemma 3.4.2 (State-Independent Event Pattern Propagation) *Let* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ *be a function node with* $\mathcal{A} = (S, s_0, T)$ *with an output port* $p^{out} \in \mathcal{P}^{out}$ *with* $B = \Sigma(p^{out})$ *that is state-independent. Let us furthermore assume that all event patterns of channels to the same input port of* f *are period-equivalent. Then we can apply Lemma 3.4.1 to determine the output event pattern* $EP(B)$*.*

Proof: According to the definition of state-independence in Def. 3.2.7, for each input port $p_i^{in} \in \mathcal{P}^{in}$ *with a transition to* p^{out} *each combination of input events leads to an output event* b *at output port* p^{out} *i.e.*

$$\forall s_j \in S,\ \forall a_1^i \in A_1^i, ..., a_m^i \in A_m^i :$$
$$\exists t = (p_i^{in}, (a_1^i, ..., a_m^i), s_j \rightarrow \{...(p^{out}, b, \delta_j)...\}, s_j') \in T \text{ with } b \in B$$

From Theorem 3.3.6 follows that this leads to the following causal dependency:

$$\forall a_1^i \in A_1^i, ..., a_m^i \in A_m^i : \{p^{in}.a_1^i, ..., p^{in}.a_m^i\} \xrightarrow{[min', max']} b.$$

Together with the knowledge that all input event patterns are period-equivalent, we have satisfied all assumptions of Lemma 3.4.1 and can use it to determine $EP(\Sigma(B))$. □

As basic channels are also part of a basic function network, we show now how event propagation is performed for these channels by considering again the causal dependencies we have shown previously for basic channels.

Lemma 3.4.3 (Event Pattern Propagation for Basic Channels) *Let c be a basic channel with* $c = (p_1, p_2)$, $\Sigma_1 = \Sigma(p_1) = \{p_1.e_1, ..., p_1.e_n\}$ *and* $\Sigma_2 = \Sigma(p_2) = \{p_2.e_1, ..., p_2.e_n\}$. *Then the following holds:*

$$EP(\Sigma_2)) = ren(EP(\Sigma_1), \Sigma_2).$$

Proof: This proof follows from the event pattern propagation in Lemma 3.4.1 with

$$\begin{aligned} &\forall p_1.e \in \Sigma_1 : \ p_2.e \in \Sigma_2 : \ p_1.e \xrightarrow{[0,0]} p_2.e \\ \Longrightarrow \ &EP(\Sigma_2) = \ ren(delay(EP(\Sigma_1), [0,0]), \Sigma_2) \\ &= \ ren(EP(\Sigma_1), \Sigma_2) \end{aligned}$$

□

This concludes our journey into event pattern propagation and gives us all the statements we need to propagate event patterns in periodic state-independent function networks. Nevertheless, we want to point out how we are able to derive output event patterns also for function nodes that are not state-independent.

Deriving General Output Event Patterns

To derive output event patterns for arbitrary function nodes, we make use of the timed automaton representation that underlies the function network formalism. Under the assumption that the event patterns of all input ports of a function node are known, we are able to construct a function node as a composition of timed automata as we have defined in Section 3.3. By taking this composition, we can apply the approach of Thiele et al. [47] to determine event patterns for all output ports. In that approach, observer automata are constructed that count events and measure the distance between events of an arbitrary timed automaton. In an iterative algorithm upper and lower curves are determined represented as *Real-Time Calculus (RTC)* arrival curves with a period interval and initial start values for both curves.

The RTC arrival curves defined in Def. 2.3.3 can be translated into event patterns as a valid abstraction, which is shown in the following corollary.

Corollary 3.4.1 (Translating from RTC to Event Pattern) *A RTC function*

$$\alpha^u(\Delta) := N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor$$
$$\alpha^l(\Delta) := N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor$$

can be represented by an event pattern $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *where*

- $J = (N^u - 1) \cdot P^-$
- $O = -N^l \cdot P^+$

Proof: See Corollary A.3.1 in the appendix on page 214. □

3.4.2. Boundedness

We will show how boundedness can be decided for periodic state-independent function networks, where all source event patterns are periodic with $P^- = P^+$ and all function nodes are state-independent.

In function networks, we define buffers as *synchronization buffers*, which are used in input ports of function nodes to synchronize incoming event streams, and as activation buffers for function nodes to assure that there is only one process execution active at a time. Because both buffers are realized by the same basic component, we start with defining boundedness for synchronization buffers as follows:

Definition 3.4.1 (Boundedness of Synchronization Buffers) *A synchronization buffer* $sync = Sync(I_1, ..., I_n, c)$ *is bounded iff there exists a finite capacity* $c \in \mathbb{N}^+$ *such that the synchronization buffer automaton never reaches its 'fail' state i.e.*

$$sync\,\text{is bounded} \iff \exists c \in \mathbb{N}^+ \mid sync \text{ never reaches the 'fail' state}$$

◇

The definition of boundedness for a basic function network immediately follows from the previous definition by claiming that all buffers of a function network are bounded.

Definition 3.4.2 (Boundedness of Basic Function Networks) *Let bfn be a basic function network and SYNC bet the set of all synchronization buffer components in bfn. bfn is bounded if each synchronization buffer is bounded*

$$bfn\,\text{is bounded} \iff \forall sync \in SYNC: \; sync \text{ is bounded}$$

◇

Deciding Boundedness by Event Pattern Propagation Based on event pattern propagation, we will show in the following how boundedness can be decided for basic function networks only by using the definition of function networks and without applying any other techniques such as model-checking. Thus, we call this approach *static* decidability of buffer boundedness.

To characterize the conditions under which a finite buffer is bounded, meaning that it will not reach its 'fail' state, we need to reason about the maximum time we have to wait to see an input event on each stream. This is captured by the maximum time distance where on any two input streams I_i and I_j the same number of events has occurred. For event patterns this maximum can be approximated by using the δ_t function from Def. 3.1.7 leading to

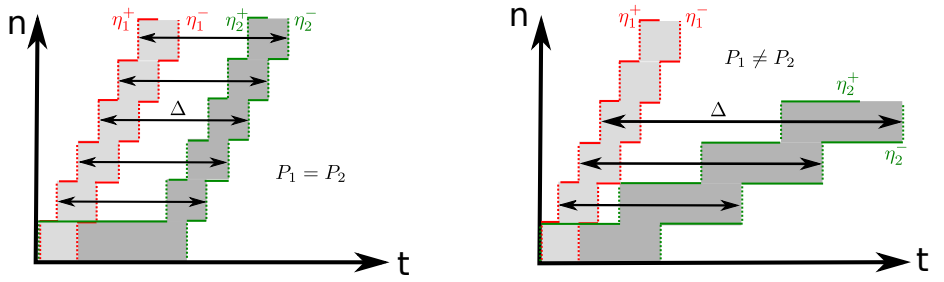
$$max := \sup_{i,j \in \{1,\ldots,n\}, i \neq j} (\delta_t(EP(I_i), EP(I_j))).$$

This is also the maximum time it takes to see a synchronization event. max may also be infinite if there is no upper bound for this waiting time which may lead to the need for an infinite buffer capacity.

If $max < \infty$ holds, this is a sufficient condition for a buffer to be bounded because the maximum waiting time is bounded and hence the number of events that need to be stored is bounded as well. But it is not a necessary condition for event patterns in general because $max = \infty$ must not always lead to an unbounded buffer. The reason for this may be correlations between the synchronized streams that are abstracted by the η functions and thus also the δ_t function leading a finite distance although $max = \infty$. However, this can only happen if streams have different lower and upper period bounds i.e. $P^- \neq P^+$. In this case, there may be some streams where the distance is finite and some where it is not finite. And if stream correlations prevent any two streams with an infinite distance to occur simultaneously, there exists a finite buffer capacity although $max = \infty$.

However, if we restrict to the subclass of periodic event patterns (i.e. $P^- = P^+$), as we do for deciding boundedness, $max < \infty$ is also a necessary condition for a buffer to be bounded. This is because all streams described by a *periodic* event pattern of a specific port have the same period and thus the time distance for two periodic event patterns will either be bounded for *all* streams or unbounded for all streams. Hence, correlations between streams can never lead to a finite distance between two streams if $max = \infty$ because the periods are fixed for all streams.

Thus, we can show for periodic event patterns that a buffer never reaches its 'fail' state if $max < \infty$ holds, and that this is only the case if all input event patterns have the same period. If this is given we can always find a finite capacity that is sufficient to avoid a buffer overflow. This is illustrated in Figure 3.32 where in Figure 3.32a the Eta-functions of two periodic event patterns EP_1 and EP_2 with the *same* period i.e. $P_1 = P_2$ are depicted. The maximum time distance is shown for a number of succeeding steps of the Eta-functions and denoted as Δ. It is obvious that there exists a maximum bound for Δ because the Eta-functions of EP_1 and EP_2 rise with the same period leading to the same slope. In Figure 3.32b the situation is shown for two periodic event patterns EP_1 and EP_2 with *different* periods i.e. $P_1 \neq P_2$. Here, the



(a) Periodic event patterns with equal periods lead to finite distance bound

(b) Periodic event patterns with different periods lead to infinite distance bound

Figure 3.32.: Deciding boundedness for periodic event patterns

distance Δ grows with every periodic step as it is shown for a number of succeeding steps. Hence, there is no finite bound for Δ leading to $max = \infty$.

We will show decidability of buffer boundedness in two steps: First, we show in Lemma 3.4.4 that a buffer overflow occurs for periodic streams if and only if $max < \infty$ holds. Based on this statement we show in a second step in Lemma 3.4.5 that the question whether a synchronization buffer is bounded is decidable if all its input event patterns are periodic.

Lemma 3.4.4 (Synchronization Buffer Overflow) *Let sync be a synchronization buffer with* $sync = Sync(I_1, ..., I_n, c)$ *with the input streams* $I_1, ..., I_n$ *where the event pattern* $EP(I_i)$ *of each input stream is periodic i.e.* $P_i^- = P_i^+$. *Then the following holds:*

$$\begin{aligned} & \exists c \in \mathbb{N}^+ \mid \textit{sync never reaches the 'fail' state} \\ \iff & max := \sup_{i,j \in \{1,...,n\}, i \neq j} (\delta_t(EP(I_i), EP(I_j))) < \infty \end{aligned}$$

Proof: The synchronization buffer automaton only reaches the 'fail' state if there occur $c+1$ *input events on the same input stream* I_i *before an event was consumed from the queue by a synchronization. A lower bound for the minimum distance between* $c+1$ *events on stream* I_i *can be determined by* $\delta^-_{EP(I_I)}(c+1)$ *(Def. 3.1.2). For periodic streams the maximum time until we see a synchronization event can be determined by considering the maximum time until we have seen an event on each two streams* I_i *and* I_j *approximated by* $max := \sup_{i,j \in \{1,...,n\}, i \neq j} (\delta_t(EP(I_i), EP(I_j)))$.

1. $\Longrightarrow$*: If we assume that there exists a finite capacity* $c \in \mathbb{N}^+$ *such that sync never reaches the 'fail' state, it must hold that there always occurs a synchronization event before* $c+1$ *events have been observed on a single input stream i.e.*

$$\forall i \in \{1, ..., n\} : \delta^-_{EP(I_I)}(c+1) \geq max.$$

This is only possible if $max < \infty$ *holds leading to the statement to prove.*

2. $\Longleftarrow$: *If we assume that* $max < \infty$ *holds, there always exists a finite capacity* $c \in \mathbb{N}^+$ *such that there never occur* $c+1$ *events on a single input stream before a synchronization event occurs. This is because periods of event patterns are always greater than zero and hence for a finite max we can always find a capacity c such that* $\forall i \in \{1, ..., n\} : \delta^-_{EP(I_I)}(c+1) \geq max$. *It follows that sync never reaches the 'fail state' for this c, which concludes the proof.*

□

Lemma 3.4.5 (Static Decidability of Buffer Boundedness) *Let sync be a synchronization buffer with* $sync = Sync(I_1, ..., I_n, c)$ *with the input streams* $I_1, ..., I_n$ *where the event pattern* $EP(I_i)$ *of each input stream is periodic i.e.* $P_i^- = P_i^+$. *Then boundedness of sync can be decided as follows:*

$$sync \text{ is bounded} \iff \forall i, j \in \{1, ..., n\} : \ EP(I_i) \overset{P}{=} EP(I_j)$$

Proof: We can apply Def. 3.4.1 and Lemma 3.4.4 and get the following statement to prove:

$$max := \sup_{i,j \in \{1,...,n\}, i \neq j} (\delta_t(EP(I_i), EP(I_j))) < \infty$$
$$\iff \forall i, j \in \{1, ..., n\} : \ EP(I_i) \overset{P}{=} EP(I_j)$$

1. $\Longrightarrow$ *(Proof by contraposition): If for any i,j the event streams of* I_i *and* I_j *are not period-equivalent i.e.* $EP(I_i) \overset{P}{\neq} EP(I_j)$, *then we know from Lemma 3.1.6 that*
$$\delta_t(EP(I_i), EP(I_j)) = \sup(\delta_t(\eta_i^+, \eta_j^-), \delta_t(\eta_i^+, \eta_j^-)) = \infty$$
because $\delta_t(\eta_i^+, \eta_j^-) = \infty$ *and* $\delta_t(\eta_i^+, \eta_j^-) = \infty$ *leading to* $max = \infty$.

2. $\Longleftarrow$ *: Assuming that* $EP(I_i) \overset{P}{=} EP(I_j)$ *holds for all* $i \in \{1, ..., n\}$, *we can determine an upper bound for max with the help of Lemma 3.1.5:*
$$\begin{aligned} max &= \sup_{i,j \in \{1,...,n\}, i \neq j} (\delta_t(EP(I_i), EP(I_j))) \\ &\leq \max_{i,j \in \{1,...,n\}, i \neq j} \max(O_i, O_j) + 2 \cdot P + J_i + J_j) \\ &\leq \max_{i,j \in \{1,...,n\}, i \neq j} (O_i) + 2 \cdot P + 2 \cdot \max_i(J_i) < \infty \end{aligned}$$
With Def. 3.1.2, we can determine a sufficient capacity c_i *for each i as follows:*
$$\begin{aligned} & (\delta^-_{EP(I_I)}(c_i + 1) \geq max \\ \iff & (c_i + 1 - 1) \cdot P - J_i \geq \max_i(O_i) + 2 \cdot P + 2 \cdot \max_i(J_i) \\ \iff & c_i \geq \frac{\max_i(O_i) + 2 \cdot \max_i(J_i) + J_i}{P} + \frac{2 \cdot P}{P} \\ \Longrightarrow & c_i = \left\lceil \frac{\max_i(O_i) + 2 \cdot \max_i(J_i) + J_i}{P} \right\rceil + 2 \end{aligned}$$

The capacity c is then determined by taking the maximum of all c_i:

$$c = \max_i(c_i) = \left\lceil \frac{\max_i(O_i) + 3 \cdot \max_i(J_i)}{P} \right\rceil + 2$$

Thus, the buffer is bounded by c.

□

It is trivial to see that a buffer with a single periodic input stream is always bounded because each periodic event stream is period-equivalent to itself.

Up to now, we have covered the general case of synchronization buffers as they are used in input ports. A special case of synchronization buffers is their use as activation buffers of function nodes. Here, we do not have an arbitrary number of streams that are synchronized but exactly two input streams. The first one contains the output events of the synchronization buffers of all input ports of the function node denoted as I_p. The second stream contains only $start_f$ events, which are needed to activate a function node. The first $start_f$ event is produced by the *loop* component at system start-up. Afterwards, another $start_f$ event occurs first after the previous execution of the respective function node has terminated indicated by a fin_f event. Thus, the language of $start_f$ events depends not only on the language over all input events I_p. Furthermore, it is determined by the execution delays of transitions of the function node which determines when a fin_f event is produced. In particular, it depends on the maximum transition delay, which we denote as δ^{max}. This is the maximum time a fin_f event may be delayed after the function node execution has started.

Thus, we first show how the language of $start_f$ events is determined before showing how to decide boundedness for activation buffers.

Lemma 3.4.6 (Language of Start Events) *Let the output language of the activation buffer $L(EP(I_p))$ be defined as follows:*

$$\begin{aligned} L(EP(I_p)) = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in I_p, \\ & (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\ \}\ & \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

Then the language of start events is determined as follows:

$$\begin{aligned} L(start_f) = \{\ & (start_f, u_1)...(start_f, u_i)...(start_f, u_{i+n})...\} \\ \mid\ & u_1 = 0, \\ & u_{i+1} = \max(u_i, t_i) + [\delta^{min}, \delta^{max}] \end{aligned}$$

where $\delta^{min}, \delta^{max}$ are defined as in Def. 3.3.13.

Proof: From the definition of the loop component in Def. 3.3.7 we know that $u_1 = 0$. From Lemma 3.3.13, we know that for the transition system it holds

$$(\sigma_i, start_f) \xrightarrow{[\delta^{min}, \delta^{max}]} fin_f$$

with $\sigma_i \in I_p$. From the loop component we know that $fin_f \xrightarrow{[0,0]} start_f$ holds. All together this leads to

$$u_{i+1} = \max(u_i, t_i) + [\delta^{min}, \delta^{max}].$$

□

Based on this language, we will now show that an activation buffer is bounded if each cycle of length k of the transition system has a length smaller than $k \cdot P$ where P is the period of input events. Otherwise, the period of fin_f events would become greater than P leading to unboundedness. If the length of a cycle is smaller, the period of fin_f events is still P because it needs still an input event to see another $start_f$ event.

Lemma 3.4.7 (Deciding Boundedness of Activation Buffers) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node with $\mathcal{A} = (S, s_0, T)$. Let further be*

- *R be the set of all partial runs of $\mathcal{A}$ where $r = (s_1 \xrightarrow{t_1} s_2 ... \xrightarrow{t_k} s_{k+1}) \in R$ is a partial run of the transition system of length k with $s_i \in S$, $t_i = (p_i, E_i, s_i \to \Psi_i, s_{i+1}) \in T$,*
- *$\delta^{max}(r) = \sum_{i=1}^{k} (\delta_i^{max})$ denote the maximum delay of a run r where δ_i^{max} denotes the maximum delay of a transition t_i as introduced in Lemma 3.3.12,*
- *$act = Sync(I_p, \{start_f\}, c)$ be the activation buffer of f,*
- *$EP(I_p) = (\Sigma^{EP}, P, P, J, O)$ be the periodic event pattern over I_p.*

Then it holds that act is bounded if each cyclic partial run of length k has a length smaller than $k \cdot P$ i.e.:

$$act \text{ is bounded} \iff \forall r = (s_1 \xrightarrow{t_1} s_2 ... \xrightarrow{t_k} s_1) \in R \ : \ \delta(r) \leq k \cdot P$$

Proof: see Lemma A.3.1 in the appendix on page 215. □

When deciding boundedness, cycles play a significant role because they may lead to unbounded systems. But a cycle is not a sufficient condition for unboundedness because their definition is based on paths without respecting any state-dependencies. To decide boundedness, we need a stronger notion of a cycle taking into account the internal states of a function node, which we will denote as *cyclic causal dependency*. Such a dependency exists if there is a cycle that cannot be left under all circumstances. Thus, we claim that there must exist mutual causal dependencies between two events leading to the following definition.

Definition 3.4.3 (Cyclic Causal Dependency) *We define a cyclic causal dependency between two ports p_1 and p_2 as a mutual causal dependency between events of these ports i.e., there exists a cyclic causal dependency if*

$$\exists e_1 \in \Sigma(p_1),\ e_2 \in \Sigma(p_2) : \ p_1.e_1 \xrightarrow{[min,max]} p_2.e_2 \ \wedge \ p_2.e_2 \xrightarrow{[min',max']} p_1.e_1$$

◇

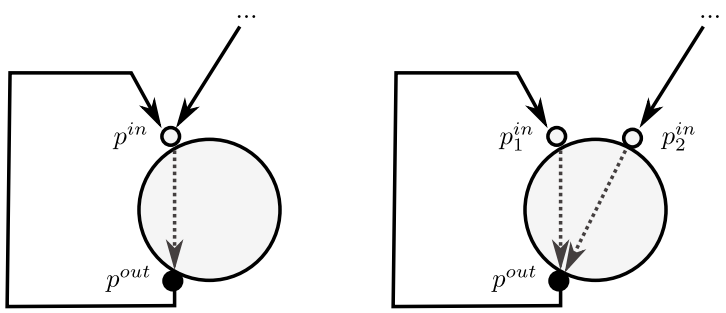


Figure 3.33.: Examples for Cyclic Causal Dependencies

The next lemma deals with the question how event patterns evolve when we apply event pattern propagation on ports with cyclic causal dependencies. In this case, event pattern propagation will never converge to a stable event pattern because the cycle cannot be left. With each propagation pass, an event pattern is superposed with itself, which halves the period and thus the period becomes infinitely small.

Lemma 3.4.8 (Self-Superposition halves Period) *Let* $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ *be two period-equivalent event patterns i.e.* $EP_1 \overset{P}{=} EP_2$ *and* $EP_s = (\Sigma_s^{EP}, P_s^-, P_s^+, J_s, O_s) = super(EP_1, EP_2)$ *be their superposition. Then it holds that*

$$P_s^+ = \frac{1}{2} \cdot P^+ \ \wedge \ P_s^- = \frac{1}{2} \cdot P^-$$

Proof:

$$P_s^+ = \frac{1}{\frac{1}{P^+} + \frac{1}{P^+}} = \frac{1}{\frac{2}{P^+}} = \frac{1}{2} \cdot P^+$$

$$P_s^- = \frac{1}{\frac{1}{P^-} + \frac{1}{P^-}} = \frac{1}{\frac{2}{P^-}} = \frac{1}{2} \cdot P^-$$

□

A cyclic causal dependency always leads to unboundedness because it can never be left. We differentiate between two different kinds of cyclic dependencies depending on whether the loop is an AND or an OR loop.

An example of an AND loop with a cyclic causal dependency is shown on the left of Figure 3.33, where a function node is depicted with an input port p^{in} with two incoming channels and an output port p^{out}. It has one transition, indicated as dotted arrow, which produces an event at p^{out} whenever there occurs an event on each input channel of p^{in}. But due to the fact that the left channel of p^{in} originates from p^{out}, there exists a cyclic causal dependency between these ports and there will never occur an event on this channel. Thus, it is not possible to determine an event pattern for p^{in}

and thus also not for p^{out}. Hence, the synchronization buffer of p^{in} would have to store all the events from the right channel without ever receiving a synchronization partner from the other channel. For a finite synchronization buffer this will lead to an overflow and its fail state will be reached because there will never occur a synchronization event.

On the right, an example of an OR loop is depicted, where a function node has a cyclic causal dependency from its output port p^{out} to its input port p_1^{in}. The transitions of the function node are again indicated as dotted arrows in the node. Assuming that the event pattern of the input port p_2^{in} is known, still the event pattern of p^{out} cannot be determined because it also depends on p^{in}. If we nevertheless take this incomplete event pattern of p^{out} and propagate it to p^{in}, we would get into an infinite propagation loop because the event pattern for the output port becomes never stable.

We will show in the next lemma for both types of cyclic dependencies that they always lead to unboundedness for state-independent function networks.

Lemma 3.4.9 (Cyclic Causal Dependency leads to Unboundedness) *Let bfn be a reachable function network with* $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ *and* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ *be a state-independent function node with an input port* $p^{in} \in \mathcal{P}^{in}$ *and a output port* $p^{out} \in \mathcal{P}^{out}$ *and a cyclic causal dependency between* p^{in} *and* p^{out}*. Let further be sync the synchronization buffer of* p^{in} *and act the activation buffer of* f*. Then it holds that at least sync is unbounded or act is unbounded i.e.*

$$\begin{aligned} & \exists e_{in} \in \Sigma(p_{in}),\ e_{out} \in \Sigma(p_{out}) : \\ & p^{in}.e_{in} \xrightarrow{[min,max]} p^{out}.e_{out} \ \wedge\ p^{out}.e_{out} \xrightarrow{[min',max']} p^{in}.e_{in} \\ \implies & \textit{act is unbounded} \ \vee\ \textit{sync is unbounded} \end{aligned}$$

Proof: see Lemma A.3.2 in the appendix on page 216. □

Now we put everything together and define an algorithm to decide boundedness for periodic state-independent function networks. The algorithm is based on event pattern propagation and collects all event patterns in a set denoted as $\mathcal{EP}$. It starts with adding all event patterns from all source nodes to $\mathcal{EP}$ because they are known by the definition of function networks. Then it checks iteratively for each input port of a function node whether for each incoming channel an event pattern is available. Initially, this is given for all channels that start at a source node. If an event pattern for an input port can be determined, it is added to $\mathcal{EP}$. In the next step, it is checked for each output port of a function node if $\mathcal{EP}$ contains an event pattern for each input port with a transition (and thus also a causal dependency) to that output port. If this is the case, also the event pattern of the output port can be derived and added to $\mathcal{EP}$. This process is repeated until either a synchronization or an activation buffer becomes unbounded or no further event patterns can be derived. In the latter case, boundedness can be decided as follows: If there exists an event pattern for each function node in $\mathcal{EP}$, the function network is bounded. Otherwise, there must exist a cyclic causal dependency leading to an unbounded function network.

Definition 3.4.4 (Algorithm to decide Boundedness) *Let $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ be a periodic state-independent basic function network and $\mathcal{EP}$ be a set of event patterns, which is initially empty i.e., $\mathcal{EP} = \emptyset$. The algorithm to decide boundedness is defined as follows:*

1. *For each output port of a source node we know the event pattern by definition and add it to $\mathcal{EP}$ i.e. $\forall \phi = (EP, \mathcal{P}^{out}) \in \Phi,\ \forall p^{out} \in \mathcal{P}^{out} : EP(p^{out}) \in \mathcal{EP}$*

2. *For each function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$:*

   a) *For each input port $p^{in} \in \mathcal{P}^{in}$, where for each input channel $c_i = (p_i, p^{in})$ $(i \in \{1, ..., n\})$ exists an event pattern $EP(p_i) \in \mathcal{EP}$:*

      i. *If $\forall i, j \in \{1, ..., n\} : EP(p_i), EP(p_j) \in \mathcal{EP} \ \wedge \ EP(p_i) \overset{P}{=} EP(p_j)$, then determine $EP(p^{in}) \in \mathcal{EP}$ as shown in Lemma 3.4.1.*

      ii. *If otherwise $\exists i, j \in \{1, ..., n\} : EP(p_i), EP(p_j) \in \mathcal{EP} \ \wedge \ EP(p_i) \overset{P}{\neq} EP(p_j)$, then the synchronization buffer of this port is unbounded and thus bfn is* unbounded.

   b) *If there does not already exists an $EP(f) \in \mathcal{EP}$ and for each input port p_i^{in} (with $i \in \{1, ..., n\}$) exists an event pattern $EP(p_i^{in})$, then decide boundedness of the activation buffer as shown in Lemma 3.4.7. If the activation buffer is unbounded, also bfn is* unbounded.

      Otherwise, add the event pattern of f to $\mathcal{EP}$, which is determined as

      $$EP(f) = super(EP(p_1^{in}), ..., EP(p_n^{in})) \in \mathcal{EP}.$$

   c) *For each output port $p^{out} \in \mathcal{P}^{out}$, where $\nexists EP(p^{out}) \in \mathcal{EP}$: If there exists an event pattern $EP(p^{in}) \in \mathcal{EP}$ for each input port p_I^{in} with $i \in \{1, ..., n\}$ with a transition to p^{out}, then determine $EP(p^{out})$ with Lemma 3.4.2 as follows:*

      $$EP(B) = super(EP_1(B), ..., EP_n(B)) \in \mathcal{EP}$$

      where $B = \Sigma(p^{out})$.

3. *Repeat the steps 2(a), 2(b) and 2(c) until either*

   a) *for each function node $f \in \mathcal{F}$ exists an event pattern $EP(f) \in \mathcal{EP}$, and thus bfn is* bounded*, or*

   b) *the size of $\mathcal{EP}$ does not increase anymore. Then bfn is* unbounded.

$\diamond$

As the final theorem, we show that boundedness is statically decidable if all source nodes deliver periodic event patterns, each function node is state-independent and the function network is reachable. This statement is proved with the help of the previously defined algorithm by showing its correctness.

Theorem 3.4.1 (Deciding Function Network Boundedness) *Boundedness for a basic function network bfn is decidable if the following assumptions hold:*

- *Assumption A1: For each source node the output event pattern EP is periodic i.e.,* $\forall \phi = ((\Sigma^{EP}, P^-, P^+, J, O), \mathcal{P}^{out}) \in \Phi : P^- = P^+$.
- *Assumption A2: Each function node is state-independent.*
- *Assumption A3: bfn is reachable.*

Proof: We prove this lemma with the help of the algorithm defined in Def. 3.4.4 by showing that it decides boundedness for periodic state-independent function networks. A function network is bounded if all its synchronization buffers are bounded and unbounded as soon as at least one buffer is unbounded (see Def. 3.4.2). Synchronization buffers can be found at two places in a function network: 1. as synchronization buffers in input ports and 2. as activation buffers of function nodes. A special case is the presence of causality loops leading to unboundedness, which is covered in 3.

1. *Boundedness of synchronization buffers in input ports is checked in step 2 (a) of the algorithm. As soon as the event patterns of all incoming streams are known, it can be decided if this buffer is bounded (Lemma 3.4.5). If the buffer is bounded, the event pattern of the input port can be determined as done in step 2(a) i. If any synchronization buffer is unbounded, also the function network is unbounded as stated in step 2(a) ii. of the algorithm.*

2. *Boundedness of activation buffers can only be decided if all event patterns of all input ports are available meaning that their synchronization buffers are bounded. This question was covered by 1. of this proof. Assuming the availability of the event patterns for all input ports, their superposition is also periodic by Def. 3.1.9. Then, boundedness of the activation buffer can be decided as shown in Lemma 3.4.7. This is done in step 2(b) of the algorithm.*

   Thus, if boundedness of all input port synchronization buffers could be shown, boundedness of each activation buffer can be decided. And only if the activation buffer of a function node is bounded, an event pattern for the function node is determined and added to $\mathcal{EP}$*. This means, that if for each function node an event pattern could be determined, each activation buffer and each synchronization buffer must be bounded and thus the whole function network is bounded. This is covered by step 3 (a) of the algorithm.*

3. *A third case is that an event pattern of an incoming stream of an input port is not known. One reason for this may be that the event pattern propagation has not reached the previous function node yet. This is covered by step 3 of the algorithm where the steps 2(a),2(b) and 2(c) are repeated as long as new event patterns can be derived.*

   If we cannot derive any new event patterns but there are still event patterns of function nodes missing, the only reason for this can be cyclic causal dependencies. This is because without cyclic causal dependencies, we would always be able to

derive new event patterns or decide that the function network is unbounded. This is assured by the assumptions A2 and A3 stating that bfn is state-independent and each port of bfn is reachable. In the case of a cyclic causal dependency, we cannot determine the event pattern of an output port p^{out} because it depends on a port p^{in} whose event pattern we cannot determine as well. Lemma 3.4.9 shows that such a cyclic dependency always leads to unboundedness. In this case, the algorithm would decide that bfn is unbounded in step 3b) because the set of event patterns $\mathcal{EP}$ is incomplete while its size does not grow anymore. □

This concludes the proof that boundedness is decidable for reachable *basic* function networks if they are periodic and state-independent. Because the Simulink translation and the task creation process are based on extended function networks, it is furthermore interesting if boundedness is also decidable for extended function networks.

Deciding boundedness for extended function networks

To extend the notion of state-independence to extended function networks, we prove under which conditions this property holds also for its additional elements by considering their translation to function nodes. We start with showing that FIFO, shared and signal data nodes are state-independent by definition.

Corollary 3.4.2 (State-Independence of FIFO, Shared and Signal) *The data nodes FIFO, shared and signal are state-independent.*

Proof:

1. *FIFO: As defined in Def. 3.2.3, a FIFO data node is state-independent because for its only output port p^{out} it holds that all transitions to p^{out} are triggered by the event r and there exists no other transition from another port leading to p^{out}.*
2. *Shared: The proof works similar as for FIFO data nodes.*
3. *Signal: As defined in Def. 3.2.3, a signal data node has only one state.* □

For a finite source data node, the property of state-independence is only given under the condition that the minimum distance between input events is less than the minimum distance between events of its triggering source node, which is $P^- - J$.

Corollary 3.4.3 (State-Independence of Finite Source) *Let $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ be the event pattern of a data node $d_{fsrc} = (\{p^{in}\}, \delta, EP, \{p^{out}\})$. d_{fsrc} is state-independent if the following holds:*

$$\Delta(\Sigma(p^{out}), \Sigma(p^{in})) \leq P^- - J$$

Proof: A finite source data node behaves only state-independent if it is assured that each time an event $e \in \Sigma^{EP}$ from its event source arrives, it is in the 'ready' state such that an event $e \in \Sigma(p^{out})$ is produced at p^{out}. We know from Corollary 3.3.5 that under this assumption, the causal dependency $p^{out}_{tr}.e \xrightarrow{\delta} p^{out}.b$ holds leading to state-independence. □

Another additional element of extended function networks are read channels. Due to Def. 3.2.3, an input port p with read channels of a function node f is translated to a function node f_p. f_p is state-independent because it has only one state. Thus, also read channels are part of the class of periodic state-independent function networks. This enables us to decide boundedness also for extended function networks as long as the mentioned assumptions for finite source data nodes are satisfied and all input event patterns are periodic.

3.5. Summary and Related Work

We introduced the formalism of function networks as an extension of classic task networks. We started with a motivation why this is useful and needed within the context of this work and also beyond. When defining function networks, we distinguish between a basic function network offering a minimum set of elements for semantics definition and proofs, and an extended function network. The latter is intended to give different roles to function nodes as for example the role of a signal to model time-consuming communication in a distributed system. To describe the occurrence of events at observation points in the network, we defined event patterns that are an extension of common event stream models. Semantics of function networks were defined by a set of atomic components that are each defined as networks of timed automata. These components are composed to build the elements of a basic function network while semantics of an extended function network is implicitly defined by a formal translation into a basic function network.

We defined a set of patterns to capture certain causal and timing properties of function networks components. This enables us to describe which properties we need to preserve when translating Simulink models and creating tasks. For each basic function network component we showed causal dependencies from input to output events by using these patterns. The respective properties of extended function network components were derived with the help of the translation from basic function networks.

Another important property for a task model that should be implemented is the question of boundedness. We proved for a specific class of function networks, how boundedness can be decided by propagating event patterns through the network starting at the sources. For a bounded function network we can determine sufficient finite buffer sizes for all synchronization buffers and thus are able to implement a bounded function network as a composition of timed automata with a finite set of states. The assumptions we did for this part of the thesis are sufficient but not necessary conditions to decide boundedness. This means, that there may exist further classes of function networks where boundedness is decidable.

In summary, with function networks we defined a modeling formalism that allows the application of analysis techniques on different abstraction levels. On the one hand, the classic entities for scheduling analysis such as tasks and signals can be identified building a bridge to real-time scheduling theory. On the other hand, semantics is given in terms of timed automata allowing the application of the manifold analysis techniques that are available for this formalism. In [16], we showed how we can combine testing

and model-checking for function networks while using the model-checker UPPAAL as back-end. In [14, 15], we showed how we can apply these analysis techniques also to a System-C based modeling language named OSSS by giving a formal translation into function networks. Thus, function networks serve as intermediate language between high level specification languages and those formalisms used for analysis and verification such as timed automata.

Related Work As motivated in the introduction of this chapter, the background of this work is mainly real-time scheduling theory meaning in particular design and analysis of real-time software tasks allocated on a distributed heterogeneous multiprocessor hardware architecture. The formalism that is naturally considered within this field are task networks. Theory about task networks has largely evolved over decades and has started with simple independent tasks with periodic behavior [51]. Later, communication was introduced by defining task networks of periodic tasks that exchange messages at the end of their execution, as for example in [79]. Nowadays, we are dealing with complex task activation models regarding OR and AND activations of tasks, which can be found e.g. in [36] and [69]. In [68] a notion of hierarchical event streams is introduced which give a more exact representation of multiple AND or OR-connected input streams than the more conservative AND and OR-operations from the work of Jersak [40]. To be able to analyze such complex models, new methods and techniques were developed. A popular approach is the compositional performance analysis [40], which is based on event models as well but extends the classic task model by considering multiple inputs and outputs for tasks. For this, AND and OR operations on multiple input streams are defined while focusing on periodic streams with jitter. Additionally, data rates are considered, which state how many tokens are consumed or produced. Another approach is the modular performance analysis (MPA) [83, 78, 47] which is also based on a formalism with many similarities to event streams named Real-Time Calculus. Here, so-called arrival functions are used to model the computation that is requested by a process, and service functions are used to model the amount of computation that can be delivered by a resource. In [84] the model of real-time calculus is extended to support different types of events on the same stream that each lead to a different workload and thus to more exact analysis results. Also in [37] arrival functions are used that are quite similar to Real-Time Calculus. They describe periodic streams with an initial burst value and a minimum period.

Nonetheless, task network models are typically not able to model functional behavior, which is often inevitable in dependable system design. Thus, much work has been done to combine them with other formalisms and techniques. For example, in [47], the MPA approach has been combined with timed automata while offering methods that allow to transform the model of one formalism to another. This means that timed automata and arrival curves may be derived from each other.

Another work [39] extended the MPA approach to be able to regard, on the one hand, correlations of streams that are first split and later joined to one stream again and, on the other hand, the blocking read semantics of Kahn Process Networks. The approach of [73] extended the compositional performance analysis to also consider path forking

and merging as well as functional cycles in a task network by bounding the maximum number of tokens in a cycle. Another example is [31] where timed automata are used to model activation patterns for tasks that are more complex than the periodic model with jitter or the sporadic event model allow. Furthermore, resource constraints are considered as an access to a shared variable using semaphores.

Beside task networks, there are also other stream-oriented formalisms that are suitable to capture control and signal processing applications like Kahn Process Networks (KPN) [42], Synchronous Data Flow Graphs (SDFG) [34], and Petri Nets [29] and its timed extensions as e.g. Time Petri Nets or Timed Arc Petri nets [75].

We will now compare function networks exemplary with some of these formalisms starting with Petri nets and synchronous data flow graphs, which are a subclass of Petri nets and known as weighted marked graphs in Petri net theory [34]. The major difference to function networks is a missing notion of time and that processes are lazy. This means that they are not necessarily executed as soon as they are ready to execute. In Petri nets, this is modeled by tokens that are placed initially at places and then travel through the network. As soon as the needed number of tokens is available, a process may be executed but it does not have to. Thus, movement of tokens cannot be related to concrete time instances as it is the case for function networks, where we use event patterns to describe the occurrences of events. Additionally, function networks allow to model internal states of processes, that influence their behavior, which is not possible in Petri nets. Some timing aspects of function networks may be covered by using for example *timed* Petri nets, which consider execution delays of processes.

A model with some similarities to Petri nets are Kahn Process Networks [62], which are untimed as well. Processes are also lazy and defined in terms of a program with 'put' and 'get' operations on incoming and outgoing FIFO streams. A process is always either executing or waiting for a single input token on one stream. The execution delay of processes is not determined explicitly in KPNs. Instead, it is defined that a process produces its output after an unknown but finite amount of time.

Another very common and established formalism to model real-time systems are timed automata (TA) [3]. Timed automata are often used for modeling real-time systems because there exist a lot of verification tools, such as UPPAAL [5]. Furthermore, its syntax and semantics is quite intuitive and easy to understand and TAs are able to model also systems of industrial size. But TAs are not very well suited for the system design itself due to the lack of high level building blocks and a more abstract or concise representation of e.g. tasks or processes [75]. This is the reason why we do not use TAs for system modeling itself but based semantics of function networks on TAs to benefit from the manifold available tools and methodologies.

For modeling real-time systems, there also exist algebraic approaches like CCS and CSP [67], timed CSP [38] and combination of data, process and time in CSP-OZ-DC (COD) [38]. These languages originate from the area of formal verification and are thus not intended to model task networks and to perform a scheduling analysis. They are typically more expressive than classic task networks and allow a more detailed modeling of functional behavior in combination with timing properties.

Concerning high level system modeling, there exist a number of frameworks for heterogeneous modeling such as Metropolis [21] and Ptolemy [9]. Metropolis [21] is a

platform-based design methodology with formal modeling techniques and abstraction levels. It allows to model functional (software) parts in terms of processes and so-called media used for inter-process communication as well as service (hardware) parts in terms of quantity managers (e.g. arbiter). Both parts are organized in netlists, where the software part is called *scheduled netlist* and the hardware part *scheduling netlist*. It was later enhanced to METRO II to allow heterogeneous IP import, separation of performance and behavior properties, and design space exploration. METRO II is based on events, offers building blocks such as components, ports and connections and also declarative specifications in terms of constraints and assertions. With adapters different models of computation (MOCs) can be hierarchically combined. The modeling framework Ptolemy II [9], which was developed in the Ptolemy project, also supports a number of different MOCs as e.g. discrete time, continuous time, synchronous data flow or process networks and allows to experiment with them. MOCs may be combined hierarchically for system modeling.

In contrast to these frameworks, the formalism of function networks is intended as intermediate language for real-time analysis and not as a high-level modeling language to compose a system. Thus, it is typically derived from a specification - as we will show for Simulink in the next chapter - to use it as formal model for further design and analysis steps. Nevertheless, function networks offer an abstraction level where relevant entities for analyzing timing properties of systems as tasks, shared variables, and FIFO buffers can still be identified. This allows, for example, to represent intermediate results of the design space exploration in terms of function networks.

4. Translating Simulink Models to Function Networks

As sketched in the introduction in Chapter 1, our approach addresses a scenario where a new feature given as a Simulink specification model should be allocated to an existing system model of a car. The goal is to find a cost-minimal extension of the target hardware to allocate the new feature while meeting all timing constraints. To be able to do this, we need an allocation of the tasks that model the new feature to hardware resources. The first step to reach this goal is to translate the specification model into a function network, which enables the application of timing analysis techniques.

In this translation step, it is essential to preserve the semantics of the original model to get correct results. Execution semantics of Simulink models is defined in terms of a partial order of block executions. To separate the translation from the task creation step and to keep the translation as simple as possible, we want to preserve the flattened block structure of the Simulink model as well. Hierarchy in terms of subsystems will not be translated because it does not add semantic expressiveness i.e., each hierarchical Simulink model may be represented by a flat model with the same semantics [52]. Nevertheless, hierarchy may be considered in task creation in terms of partitioning constraints as we will explain in Chapter 5.

For the translation, we need to find reasonable representations for blocks and signals in the function network formalism. A block can be considered as a process that realizes a specific function and is, in the end, represented as a piece of program code. Thus, its intuitive representation in a function network is a function node. Accordingly, a signal between two blocks is modeled as a channel in the function network to maintain the connectivity of blocks. While the translation of the structure is quite intuitive, the correct representation of timing semantics of the Simulink model needs some deeper investigations. First, for this work, we are not aiming at representing the complete functional behavior of a Simulink model in function networks. This means in particular that we do not want to model the concrete numerical values of signals. What we are interested in, is the timing behavior of the model in terms of the order in which blocks are executed, and the time a complete simulation step needs to be finished. While for the simulation of a model in Simulink a total order on blocks named *Block Sorted Order* is created, we will refer to the partial order induced by the connectivity of blocks and signals that underlies this total order. This is sufficient because the partial order of blocks guarantees that each total order that satisfies the partial order leads to the same functional simulation results [22, 52].

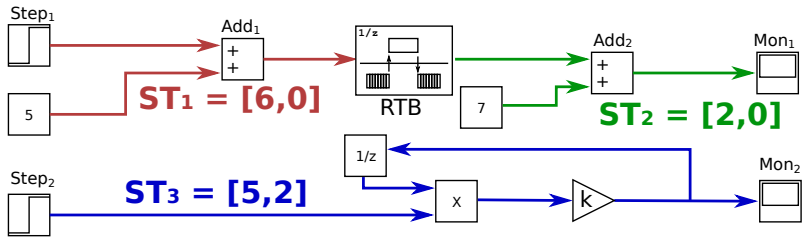
This leads to the main challenge of this step: the translation of a synchronous specification into a task model, where activations of tasks are triggered by events. An execution of a Simulink model can be described as a sequence of updates on signals

that occur due to block executions. This means that at each execution a block reads the current values from its input signals and updates its output signals. Whether a block is executed in a specific simulation step depends on the block sample time given by a period and an initial phase offset. A set of connected blocks with the same sample times is called a *synchronous set* [22]. Blocks of different synchronous sets may be connected by *rate transition blocks*, which guarantee a deterministic data-transfer and compliance with the partial order when the blocks of both synchronous sets are executed in the same simulation step. It is possible to connect blocks of different synchronous sets only if one period is an integral multiple of the other. In contrast to Simulink block diagrams, function networks combine process executions and the reading of input values by the concept of events. A function node is executed whenever on each channel of an input port of the node an event was received. Events may also serve as an abstract representation of values.

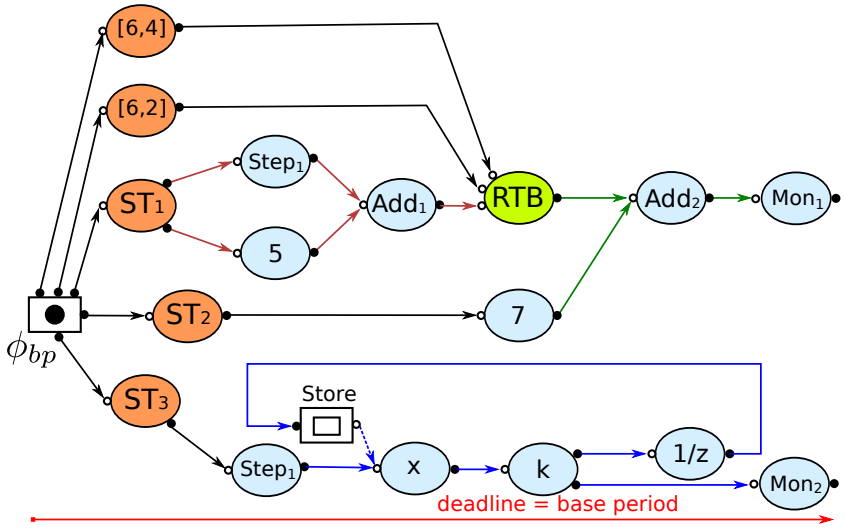
In Figure 4.1a, an example of a Simulink block diagram is depicted consisting of three different synchronous sets with sample times ST_1, ST_2 and ST_3. Sample times are given by a period *per* and an initial phase offset *init* written as $ST_i = [per_i, init_i]$. The initial phase offset determines the first execution of a block. Thus, a sample time of $[5, 2]$ with $per = 5$ and $init = 2$ leads to block executions in the simulation steps 2,7,12,17... . The synchronous sets with sample times $ST_1 = [6, 0]$ and $ST_2 = [2, 0]$ are connected by a rate transition block RTB. Here, the source block of the rate transition runs with a slower rate (greater period) than the target block. Thus, the blocks with sample time ST_2 are executed more often but always need the value from the output signal of the last block with ST_1, which is the block Add_1. In this case, the rate transition block stores the latest value of this signal and provides it to the succeeding block. Whenever both synchronous sets are executed within the same simulation step, blocks with ST_1 are executed first due to the partial order and the rate transition block stores the value for later execution instances.

In Figure 4.1b, the corresponding function network representation is depicted. It starts on the left with a source node ϕ_{br} producing events with the *base period* (*bp*) of the Simulink model. The base period is the greatest common divisor of all periods that occur in the model and is here determined to $bp = 1$. Function nodes that have incoming channels from ϕ_{bp} are special function nodes to convert the base period into the sample time of each existing synchronous set. Because the period of each sample time is a multiple of the base period, we call these nodes *period multiplier* nodes. For a period $per = k \cdot bp$ the respective period multiplier node has a transition system that produces for each k^{th} input event an output event. Furthermore, output events may be delayed by the initial phase offset *init* of the sample time.

In order to be able to reason about semantics preservation of the specification model, we first define formal semantics for the execution of Simulink models based on updates of signals due to block executions. In a second step, we relate those signal updates to events in the function network translation. This means that each time a signal is updated by a block in Simulink, there must occur a corresponding event in the function network representation. For function nodes that originate from blocks within the same synchronous set, a synchronization of all input channels at one input port of



(a) Simple Example of a Simulink Block Diagram



(b) Translation of Block Diagram to Function Network

Figure 4.1.: Example for a Simulink Translation

each node leads to the intended behavior. This means that the execution only starts if all needed input signals have been updated.

For connected blocks of different synchronous sets, the concept of rate transitions is transfered to function networks by creating a dedicated function node that translates from the sample time of the source block to the sample time of the target block. If the target sample time is faster in terms of a smaller period, we need to produce the missing output events in those simulation steps, where only the target block is executed. This is done by adding activation channels from further period multiplier nodes, which create the needed events. For the other case, where the period of the target block is higher, the rate transition is modeled by a period multiplier node. Thus, if we translate from a period per to a period $k \cdot per$, only each k^{th} input event leads to an output event and an activation of of the successor node.

In Figure 4.1, the rate transition block RTB connects blocks with ST_1 and ST_2 where $per_1 = 6 > per_2 = 2$. This rate transition block is represented by a function

node RTB with three input ports in the function network translation. One input port receives all output events from the direct predecessor block Add_1 with sample time ST_1. These events are forwarded to the direct successor block Add_2 of the synchronous set with sample time ST_2. Thus, each time when both synchronous sets run together, the partial order is obtained because the blocks with sample time ST_1 are executed first. For those steps where only the blocks with ST_2 run, two additional period multiplier nodes are created to produce all missing events to model ST_2. This is done by taking the period of ST_1 and any initial phase offset that is a multiple of the period of ST_2, except the one of ST_1. In this example, this leads to a period of 6 and offsets of 2 and 4 because the offset 0 is already covered by ST_1. We denote such a set of sample times as $ST_1 \setminus ST_2 = \{[6, 2], [6, 4]\}$.

All remaining 'ordinary' Simulink blocks are translated to function nodes with one input port where all signals of preceding nodes are synchronized. An example is block Add_1, which waits for executions of its predecessor nodes $Step_1$ and 5. Sink blocks are represented as function nodes without outgoing channels because they do not produce any output signals. In our example, there are two sink blocks Mon_1 and Mon_2 representing monitor blocks in Simulink to observe output values of the model.

For signals that close a non-algebraic loop, which is a loop containing a delay block, a shared data node is created. This shared data node stores the value of the previous simulation step to provide it in the succeeding step. Thus, the respective start node of the loop has a read channel to read from the shared data node when activated. In our example, we have a loop starting at the block X and ending at the unit delay block $1/z$. In the function network translation, we have a shared data node $Store$, which stores the latest value produced by the node $1/z$ and the node X has a read channel to read from $Store$. Here, it is important to note that we need to assure that in each simulation step each sequence of blocks is executed before the next simulation step starts. Otherwise, the value in the shared data node $Store$ might not have been updated by $1/z$ and the next execution of X would read the old value leading to wrong results. This leads to the need for defining respective end-to-end deadlines when considering implementations of Simulink models.

To enable the implementation of the model on a target hardware architecture, we employ existing code generators, such as *TargetLink* and *Embedded Coder*, to generate code for each single block. Worst case execution times (WCETs) are calculated for the resulting code for each block by using tools like aiT [30] and used to assign weights to the respective function nodes in the task creation step. Here, we take the minimum WCET of all available processors and hence assume each block to run on its optimal processor. This has two reasons: First, we can safely state that certain deadlines are violated if already an optimal deployment would exceed the deadline, and second also the design space exploration is based on this assumption which enables to guarantee lower bounds for hardware costs. Please note, that the WCET for standard blocks from the Simulink library only needs to be calculated once for each processor type and then can be used for each model that should be translated. Accordingly, for this work, we consider *executable* Simulink specification models that can be used for code generation and meet the TargetLink modeling guidelines [26, 54].

This leads to the second important issue when reasoning about semantic preservation: the compliance of timing constraints. Simulink models are inherently untimed where block execution and communication is instantaneous. Obviously, this does not hold for any implementation of a Simulink model that runs on real hardware. Thus, Simulink implicitly assumes that the execution of all connected blocks is finished before the next simulation step starts. Accordingly, as the final step to capture Simulink semantics correctly, we have to ensure that the execution of all blocks that may be executed in the same simulation step is finished before the next simulation step begins. Otherwise, we might get overlapping simulation steps and signals that Simulink assumes to be updated may not be updated in time in the function network translation. In the example from Figure 4.1, this is the case for the non-algebraic loop from block X to block $1/z$. To capture such constraints, we employ the concept of *end-to-end deadlines* as introduced in Section 3.2.3. In order to preserve the semantics of a Simulink model, we thus define end-to-end deadlines for each maximal chain of partially ordered nodes that may be executed in the same simulation step. The length of the deadline is determined by the base period bp of the Simulink model. Accordingly, in the example of Figure 4.1, we define a deadline from the source node to each sink node of the function network with a length of $bp = 1$ time units.

Please note that it is important for the task creation process to retain the different synchronous sets because otherwise no valid deadlines could be defined. Accordingly, task creation will be constrained in Chapter 5 such that the merging of any two function nodes of different synchronous sets into the same task is forbidden.

Chapter Outline In Section 4.1, we define a formal execution semantics for Simulink models based on the formalism of *Timed Synchronous Block Diagrams*. With the help of this semantics, we define in Section 4.2 a translation scheme from Simulink to function networks and prove in Section 4.3 that semantics is preserved in terms of partial order and timing. Section 4.4 summarizes this chapter, discusses related work and gives an outlook to possible extensions of the the presented approach.

4.1. Formal Semantics for Simulink Models

Due to a missing 'official' formal semantics, we rely on the most common approach found in literature, where discrete Simulink models are defined as *Synchronous Block Diagrams* (SBDs) and *Timed Synchronous Block Diagrams* (TBDs), respectively [53, 52, 60]. A synchronous block diagram is a directed graph where nodes are blocks that are connected by edges called signals. Timed synchronous block diagrams are an extension where special triggers are introduced to model different sample times.

To be able to implement the specification model on target processors, we generate code for each Simulink block using TargetLink. Thus, we assume a discrete Simulink model that meets the TargetLink modeling guidelines [26, 54]. This means, among other things, that only a subset of block types is available. For example, no blocks from the *continuous* library are allowed and only a few blocks from the *source* library may be used. Furthermore, block priorities are ignored and algebraic loops must not

be used. The latter assumption is also necessary to be able to represent a Simulink model as synchronous block diagram. The simulation parameters must be chosen such that a fixed-step solver with a single-task model is used. This also means that different synchronous sets may be only connected by rate transition blocks.

4.1.1. Timed Synchronous Block Diagrams

According to [53, 52], a *Synchronous Block Diagram* (SBD) consists of a set of *blocks* having input and output *ports*. Blocks are either *atomic* or *composite* blocks, where the latter ones contain sub-blocks. Atomic blocks are classified as either *combinational* blocks, which are state-less, or *sequential* blocks, which contain internal states. Sequential blocks are called *Moore-sequential* if their output "only depends on the state, but not on the inputs" [52]. A block diagram is created by connecting output ports with input ports of blocks via *signals*. Output ports can be connected to more than one input port while input ports can only be connected to one output port. If a block has a boolean input signal working as *trigger*, it only computes new output values if the value of this trigger signal is true. For each output of a triggered block, the user has to specify initial values that are valid for the starting phase before the trigger was true for the first time. An SBD is called *flat* "if it only contains atomic blocks" [52].

Semantics of synchronous block diagrams is defined in [53] by defining semantics of signals by determining at which points in time a signal is updated by a block. In this definition, a signal gets assigned a value for each simulation step. This means that there is a value available for each signal at each time t_i. The decision if and how a value is changed depends on the block that produces the signal, and its trigger. In the following, we define signal semantics based on [53] while we explicitly represent the input signals of a block producing a signal.

Definition 4.1.1 (Signal Semantics) *A signal s is a total function $s : \mathbb{N}^+ \to V$ where V is a non-finite set of values and $s(t_i)$ denotes the value of s at time t_i. Let b be the block that produces s with the input signals $in_1, ..., in_n$, and v_s the initial value of s. Let further denote $b(\{in_1, ..., in_n\}, s, t_i)$ the result of the execution of b for signal s at time t_i. If b is triggered by a signal tr, $s(t_i)$ is determined as follows:*

$$s(t_i) = \begin{cases} v_s & , if\ tr(t_i) = false \wedge t_i = 0 \\ s(t_{i-1}) & , if\ tr(t_i) = false \wedge t_i > 0 \\ b(\{in_1, ..., in_n\}, s, t_i) & , if\ tr(t_i) = true \end{cases}$$

If b_s has no trigger, s is determined as if the trigger was always true. ◇

A *Timed Synchronous Block Diagram* (TBD) is an SBD where each non-triggered block has a special trigger called *firing time specification* (FTS) [52]. The semantics are equivalent to SBDs, because firing time specifications are a special case of triggers. In Simulink, an FTS is called *sample time*. Based on [52], we define a firing time specification as follows:

Definition 4.1.2 (Firing Time Specification) *A Firing Time Specification (FTS) is a pair* $fts = (per, init)$ *where* per *is the period and* $init$ *is the initial phase offset with* $init \in \mathbb{N}_0$, $per \in \mathbb{N}^+$ *and* $init < per$. ◇

As a next step, we formalize the definition of a timed block diagram to a graph that consists of blocks, signals and edges connecting blocks via signals. Because the hierarchy of a TBD in terms of composite blocks - which are subsystems in Simulink - does not add expressiveness but is only intended to structure the model, we restrict to flat TBDs. A procedure to flatten a TBD is described in the work of Lublinerman [52]. Furthermore, as in [52], we assume TBDs to be acyclic in terms of algebraic loops. This means that all cycles must contain at least one Moore-sequential block such as a *Unit Delay* block in Simulink. This assures a valid and cycle-free partial order of block executions, which guarantees that values of signals are updated correctly. We define a timed synchronous block diagram as follows:

Definition 4.1.3 (TBD) *A timed synchronous block diagram (TBD) is a graph defined as a tuple* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *where*

- B *is a set of blocks where each block* $b \in B$ *has a type,*

$$type(b) \in \{\text{'combinational', 'sequential', 'Moore-sequential', 'rate transition', 'data store memory', 'data store write', 'data store read'}\},$$

- S *is a set of signals,*
- $E \subseteq B \times S \times B$ *is a set of edges where each edge* $e = (b, s, b') \in E$ *leads from block* b *to* b' *via signal* s *and there is exactly one producer block* b_s *for each signal* s *i.e.* $\forall s \in S : \exists!\ b_s \in B \mid (b_s, s, b') \in E$,
- $\mathcal{FTS}$ *is a a set of firing time specifications,*
- $tr : B \rightarrow \mathcal{FTS}$ *is a surjective function that assigns a block to a firing time specification i.e. each block has a firing time specification and blocks may share the same firing time specifications while for each firing time specification there exists at least one block.* ◇

Due to the target link modeling guidelines we assume the use of a fixed-step solver in Simulink. Thus, each model has a base period, which determines the time distance between two simulation steps. The base period is determined as the greatest common divisor of all firing time specifications of a TBD while we allow a global offset shifting represented as a constant x. Please note, that sample times in Simulink are relative values that have no specified time unit. Thus, we assume that the 'real' execution period of the model is determined by the base period and some factor set by the user. This factor then leads to the time unit in terms of e.g. milliseconds or microseconds, which is also assumed as time unit for the worst case execution times of blocks. We abstract from this factor by assuming that it is chosen sufficiently large to be able to represent all execution times of blocks. For some blocks, we assume an execution time of ϵ time units, which is assumed as a negligible small value greater than zero.

Definition 4.1.4 (Base Period) *Let* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *be a timed block diagram with a set of sample times* $\mathcal{FTS} = \{fts_1, ..., fts_n\}$ *where* $\forall i \in \{1, ..., n\}$: $fts_i = (per_i, init_i)$. *Then its base period is defined as*

$$bp = \max(k \mid \forall i \in \{1, ..., n\}\ \exists x \in \mathbb{N}_0 : (per_i \bmod\ k = 0)\ \wedge\ ((init_i - x) \bmod\ k = 0)$$

◇

4.1.2. Execution Semantics for Simulink Models

We introduce execution semantics for Simulink models based on timed synchronous block diagrams by defining a transition system for each simulation step. This semantics is based on updates of signals which occur due to the execution of blocks.

In general, a signal update does not necessarily mean that its value is changed but that a new value was written and is available for other blocks. Thus, we could claim that a signal update occurs at each simulation step for each signal. However, this would lead to a number of signal updates where we actually know that the value cannot change. This is the case for those time steps where a block is never executed due to its sample time. For signal semantics, this means that the trigger in terms of a firing time specification is not true, and thus the new value is identical to the value of the previous time step. This would lead to a number of block executions and thus signal updates where in fact nothing is computed.

The same holds for other kind of triggers in terms of signals from other blocks, which are determined dynamically at runtime. Those *dynamic triggers* cannot be regarded a-priori when defining signal updates. To achieve this, one would need a complete functional model of each block that produces a trigger signal to determine if this signal is true, which is not in the scope of this work. For this work, we regard dynamic triggers as ordinary signals that are evaluated by the block itself during its execution. Depending on the trigger value either new output values are computed or the old values are written again while both cases result in a signal update. Thus, the trigger signals, which are referenced in Def. 4.1.1, are always firing time specifications and no dynamic triggers.

However, this work could be extended to explicitly regard the internal behavior of blocks (e.g. Stateflow blocks), which would also allow to support dynamic triggers. To achieve this, one would need to perform a kind of abstract interpretation to identify the conditions where dynamic triggers are true. This would allow to avoid the overapproximations on signal updates that result from the assumptions we do for this work. How such an approach may work for Stateflow blocks is discussed in the summary of this chapter in Section 4.4.

To determine the static points in time where a block is ready to execute, we need to consider the period and initial phase offset given by its firing time specification. This leads to the following definition.

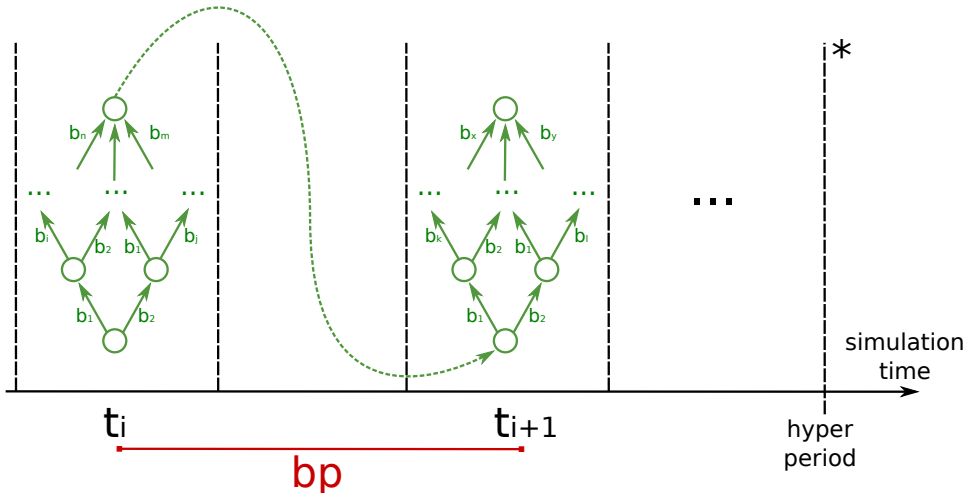


Figure 4.2.: Simulink Execution Semantics

Definition 4.1.5 (Block Ready) *Let b be a block with $tr(b) = (per, init)$. b is ready at simulation step t if its firing time specification states that this block should be executed at t i.e.*

$$rdy(b,t) = \begin{cases} true, & if\, \exists k \in \mathbb{N}_0 : init + k * per = t \\ false, & otherwise \end{cases}$$

◇

Based on this definition, we are able to define a partial order on block executions that is determined by signals between blocks and their firing time specifications. Two blocks are partially ordered at a simulation step t if there exists a signal connecting them and both blocks are ready in simulation step t. Furthermore, the source block of the signal must not be a Moore-sequential block because these blocks are delay blocks that store the value for the next simulation step.

Definition 4.1.6 (Partial Order on Block Executions) *Let $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ be a TBD. The partial order on block executions $PO_B(tbd, t)$ for simulation step t is defined as follows:*

$$\begin{aligned}(b,b') \in PO_B(tbd,t) \iff\ & b,b' \in B \ \wedge\ (b,s,b') \in E \ \wedge\ rdy(b,t) \ \wedge\ rdy(b',t) \wedge \\ & type(b) \neq \text{'Moore-sequential'} \\ & \vee\ \exists b_1,...,b_n \in B, n \in \mathbb{N}^+ : (b,b_1),...,(b_n,b') \in PO_B(tbd,t)\end{aligned}$$

We write $b_1 <_t b_2$, if $(b_1, b_2) \in PO_B(tbd, t)$. ◇

To describe execution semantics of a TBD at a specific simulation step, we define a transition system where transitions represent block executions and states are abstract

representations of the current signal values. This means, that a state indicates which signals have already been updated in the current time step.

In Figure 4.2, a time line is depicted representing the simulation time steps of Simulink. For each time t_i, we define a transition system with one initial state. This initial state is determined for the first point in time $i = 0$ as a tuple of the initial values of all signals. For $i > 0$, the state is determined as the end state of the transition system of the previous time step t_{i-1}. This is indicated by a dotted arrow connecting the respective states of both transition systems. Each transition system covers all valid orderings of block executions with respect to the partial order of the considered time step. Due to the fact that states are characterized by means of signal values and each execution order that respects the partial order leads to the same signal updates, each transition system has exactly one end state as we will show later. And because each block is executed periodically, there always exists a hyper period *hp* where the execution behavior is repeated. This means that the transition system of time step t_{i+hp} is the same as the one of time step t_i because the same set of blocks is executed. The distance between simulation steps is equivalent to the base period *bp* of the model.

We will now give a general definition of a TBD transition system and define afterwards how such a transition system is derived from a given TBD. A TBD transition system is defined as follows:

Definition 4.1.7 (TBD Transition System) *Let V be a non-finite set of signal values and B be a finite set of blocks. A TBD transition system is defined as a tuple (Q, q_0, Γ), where*

- *$Q \subseteq 2^V$ is a set of states where each state is a tuple of signal values*
- *$q_0 \in Q$ is an initial state*
- *$\Gamma : Q \times B \to Q$ is a set of transitions which lead from a source state to a target state and represent a block execution*

We further denote $Q_e \subseteq Q$ as the set of end states $Q_e = \{q_e \mid \nexists \gamma = (q_e, b, q'_e) \in \Gamma\}$.

◇

A run of a TBD transition system is defined as follows:

Definition 4.1.8 (Run of TBD Transition System) *Let $Trans = (Q, q_0, \Gamma)$ be a TBD transition system. A run of $Trans$ is a sequence of transitions starting in q_0 and ending in an end state $q_n \in Q_e$ written as*

$$run =< \gamma_1, ..., \gamma_n >$$

where $\gamma_1 = (q_0, b_1, q_1)$ and $\forall i \in \{1, ..., n-1\} : \gamma_i = (q_i, b_{i+1}, q_{i+1})$ with $q_n \in Q_e$ and $q_{i+1} = \Gamma(q_i, b_{i+1})$. The set of all runs of $Trans$ is denoted as $Runs(Trans)$. Furthermore, we write $\gamma_i \in run$ if $run =< ..., \gamma_i, ... >$.

◇

We further define the position of a transition in a run as follows:

Definition 4.1.9 (Position in Run) *Let* $run = < \gamma_1, ..., \gamma_n >$ *be a run. The position of a transition* γ_i *in run is defined as:*

$$pos(\gamma, run) = \begin{cases} i & , if\ \exists\ i \in \{1, ..., n\} :\ \gamma = \gamma_i \\ 0 & , else \end{cases}$$

◇

Based on these general definitions, we now define how we can derive a concrete TBD transition system for a given TBD and a simulation step t_i. The initial state of the transition system is determined by the initial values of each signal if $i = 0$, or by the end state of the previous simulation step t_{i-1}. We define a set of transitions for each block b that is contained in the partial order of t_i except for sink blocks. This is because sink blocks do not update any signals and their execution has no semantic relevance. Due to the block partial order, a block b may only execute if each preceding block $b_{in} <_{t_i} b$ was executed. Thus, for each block with outgoing signals, we define a transition executing b in each state q where all input signals have been computed. According to Def. 4.1.1, an update of a signal s produced by a block b with the input signals S^b_{in} is denoted as $b(S^b_{in}, s, t_i)$. For preceding blocks that are not in the partial order of t_i, the value is not updated because this block will not execute in the considered time step. This is the case for rate transition blocks having inputs from blocks with a greater period. In those situations, rate transition blocks store the values of input signals and thus do not need an update on these signals.

Furthermore, a block is never executed twice within the same simulation step. Thus, we only define a transition for those states where all output signals of the block have *not* been updated in the current simulation step. Hence, the source state of a transition must satisfy the condition that for each output signal b the value has not been updated in the currently considered time step. The target state q' of a transition is determined by adopting the signal values of the source state q for all signals that are no output signals of block b. For each output signal s_{out} of b, the value is determined to $b(S^b_{in}, s_{out}, t_i)$ where S^b_{in} represents the set of input signals of b. The set of states of the transition system is determined recursively as the set of all target states of a transition and the initial state. This leads to the following definition of a transition system for a given TBD.

Definition 4.1.10 (Transition System for TBD) *Let* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *be a TBD with* $S = \{s_1, ..., s_n\}$ *and* t_i *be a simulation step with* $i \in \mathbb{N}_0$. *Let* v_{s_j} *be the initial value of signal* s_j *with* $j \in \{1, ..., n\}$. *The transition system* $Trans(tbd, t_i) = (Q, q_0, \Gamma)$ *is defined as follows where* $q(j)$ *denotes the value of the* j^{th} *place of a state* $q \in Q$*:*

- *the initial state is inductively defined as being either the end state of the previous point in time* t_{i-1} *or the state containing the initial values if* $t_i = t_0$*, i.e.* $q_0 = (s_1(t_i), ..., s_n(t_i))$ *where*

$$s_j(t_i) = init_{s_j} := \begin{cases} v_{s_j} & , if\ t_i = t_0 \\ s_j(t_{i-1}) & , else \end{cases}$$

- *we define a transition* (q, b, q') *if* b *is in the partial order of time step* t_i*, all signals needed for the execution of* b *have already been updated in* q*, all signals that are produced by* b *have* not *been updated in* q*, and there is at least one signal that is updated by* b *at all (i.e.* b *is no sink block).* q' *is determined by updating all signals produced by* b *i.e.*

$$
\begin{aligned}
\Gamma = \{\ & (q, b, q') \mid \\
& \exists\ (b, s, b_{out}) \in E, b <_{t_i} b_{out}, \\
& \forall (b_{in}, s_k, b) \in E \text{ with } k \in \{1, ..., n\}, b_{in} <_{t_i} b : \ q(k) \neq init_{s_k}, \\
& \forall (b, s_l, b_{out}) \in E \text{ with } l \in \{1, ..., n\} : \ q(l) = init_{s_l}, \\
& \forall j \in \{1, ..., n\} : \\
& q'(j) = \begin{cases} b(S^b_{in}, s_j, t_i) & , if\ (b, s_j, b_{out}) \in E \\ & where\ S^b_{in} = \{s \in S \mid \exists (b_{in}, s, b) \in E\} \\ q(j) & , else \end{cases} \\
\}&
\end{aligned}
$$

- *the set of states is recursively defined as* $Q = \{q' \mid \exists\ (q, b \to q') \in \Gamma\} \cup \{q_0\}$ $\diamond$

To show that the TBD transition system is a valid representation of the execution semantics of a TBD, we need to prove that each transition system respects the partial order of blocks induced by the TBD and that its end state is unique.

For the compliance to the partial order, we need to show two statements. First, we need to show that for each block b_1 with $b_1 <_t b_2$, there exists exactly one transition $\gamma_{b_1} = (q_1, b_1, q_1')$ in each run. This assures that each block within the partial order that updates any signals is executed in each run exactly one time. Second, we need to show that for the case where b_2 is not a sink block (which means at the same time that there exists a transition $\gamma_{b_2} = (q_2, b_2, q_2') \in \Gamma$), b_1 is always executed before b_2. This means that its position in a run must respect the partial order leading to $pos(\gamma_{b_1}, run) < pos(\gamma_{b_2}, run)$.

Theorem 4.1.1 (Preserving Partial Order) *Let* $Trans(tbd, t) = (Q, q_0, \Gamma)$ *be a transition system for a* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *and a simulation step* t*. Then all the following holds:*

$$\forall\ b_1 <_t b_2, \forall\ run \in Runs(Trans(tbd, t)) :$$

1.

$$\exists!\ \gamma_{b_1} = (q_1, b_1, q_1') \in run$$

Proof: According to Def. 4.1.10, for each block b_1 *with* $b_1 <_t b_2$ *a transition* $(q_1, b_1, q_1') \in \Gamma$ *is created for each state* q_1 *where all input signals but no output signal of* b_1 *have been updated. Any run must start in* q_0 *and end in an end state* $q_e \in Q_e$*. An end state is a state without any outgoing transitions i.e., there is*

no block anymore that can be executed. This may be either the case because each block of the partial order was already executed (and thus also b_1), or the needed input signals have not been updated. If b_1 is a source block i.e., $\nexists b : \ b <_t b_1$, then it has no restrictions on input signal updates and must have been executed before reaching an end state. Inductively it follows that if all blocks b with $b <_t b_1$ have been executed γ_{b_1} can be taken. Whenever a transition that executes block b_1 was taken, no other transition executing the same block can be taken, because its output signals have been updated. It follows

$$\exists!\ \gamma_{b_1} = (q_1, b_1, q_1') \in run.$$

□

2.

$$\gamma_{b_2} = (q_2, b_2, q_2') \in run \implies pos(\gamma_{b_1}, run) < pos(\gamma_{b_2}, run)$$

Proof: If there exists such a γ_{b_2}, then this transition can only start in a state where all output signals of b_1 have been updated because $b_1 <_t b_2$. Thus, transition γ_{b_1} must have been executed before and hence its position in the run must precede that of γ_{b_2}.

□

From Theorem 4.1.1 follows that a TBD transition system is cycle-free because an irreflexive partial order has no cycles by definition. This also means that each run of a TBD transition system is finite and thus has a well-defined end state.

As a second condition for a transition system to be valid, we show that the end states of all runs are identical leading to a unique end state. Because states in a TBD transition system are characterized by the values of signals, two states are only identical if all their signal values are identical. Thus, if the end state is unique this means that independently from the total order how blocks are executed the resulting signal values are identical because all signals were updated in the correct partial order.

Theorem 4.1.2 (Unique End State) *Let $Trans(tbd, t) = (Q, q_0, \Gamma)$ be a transition system for a $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ with $S = \{s_1, ..., s_k\}$ at simulation step t. Then for any two runs*

- *$run_1 \in Runs(Trans(tbd, t))$ with the end state $q_{e_1} = (v_1^1, ..., v_1^k)$,*
- *$run_2 \in Runs(Trans(tbd, t))$ with the end state $q_{e_2} = (v_2^1, ..., v_2^k)$*

holds that their end states are identical i.e.

$$q_{e_1} = q_{e_2}$$

Proof: For the end states to be identical it must hold that all values are identical i.e.

$$\forall i \in \{1, ..., k\} : v_1^i = v_2^i.$$

We distinguish the following cases where we denote the block that produces s_i as b_{s_i}:

1. b_{s_i} *is not executed in time step t:*

   In this case it holds that $\nexists(b_{s_i}, b') \in PO_B(tbd, t)$ *and* $\nexists(b, b_{s_i}) \in PO_B(tbd, t)$. *Thus,* b_{s_i} *is never executed neither in* run_1 *nor in* run_2 *and the value of* s_i *remains the initial value of this time step for both runs leading to identical values*

   $$v_1^i = v_2^i = init_{s_i}.$$

2. b_{s_i} *is executed in time step t but has no incoming signals:*

   In this case it holds that $\exists(b_{s_i}, b') \in PO_B(tbd, t)$ *but* $\nexists(b, b_{s_i}) \in PO_B(tbd, t)$. *Together with Theorem 4.1.1 follows that there is exactly one transition executing* b_{s_i} *in each run i.e.* $\exists(q_1, b_{s_i}, q_1') \in run_1$ *and* $\exists(q_2, b_{s_i}, q_2') \in run_2$. *In this case the signal value must be the same for both runs because it does not depend on any input values leading to*

   $$v_1^i = v_2^i = b_{s_i}(\emptyset, s_i, t).$$

3. b_{s_i} *is executed in time step t and has incoming signals* $in_1, ..., in_x$ $(x > 0)$*:*

   In this case the value of signal s_i *is determined for the two considered runs to*

   $$v_1^i = b_{s_i}(\{v_1^{in_1}, ..., v_1^{in_x}\}, s_i, t),$$
   $$v_2^i = b_{s_i}(\{v_2^{in_1}, ..., v_2^{in_x}\}, s_i, t).$$

   Hence, $v_1^i = v_2^i$ *holds if and only if* $v_1^{in_j} = v_2^{in_j}$ *holds for all* $j \in \{1, ..., x\}$.

   In the case considered here it holds that $\exists(b_{s_i}, b') \in PO_B(tbd, t)$*, and together with Theorem 4.1.1 follows that there is exactly one transition executing* b_{s_i} *in each run i.e.* $\exists(q_1, b_{s_i}, q_1') \in run_1$ *and* $\exists(q_2, b_{s_i}, q_2') \in run_2$. *According to Theorem 4.1.1 the TBD transition system respects the partial order and hence transitions may only start at states where all needed input signals have been updated and that signals are never updated twice. Because each input signal* in_j $(j \in \{1, ..., x\})$ *has again a single producer block, denoted as* b_{in_j}*, the value of each* in_j *is either determined to*

   a) $v_1^{in_j} = v_2^{in_j} = init_{in_j}$*, if* b_{in_j} *is not executed in time step t (case 1.),*

   b) $v_1^{in_j} = v_2^{in_j} = b_{in_j}(\emptyset, s_i, t)$*, if* b_{in_j} *has no incoming signals (case 2.), or*

   c) $v_1^{in_j} = b_{in_j}(\{v_1^{in_1^j}, ..., v_1^{in_y^j}\}, s_i, t)$ *and* $v_2^{in_j} = b_{in_j}(\{v_2^{in_1^j}, ..., v_2^{in_y^j}\}, s_i, t)$*, if* b_{in_j} *has incoming signals* $in_1^j, ..., in_y^j$ *(case 3.).*

      In this case we recursively apply case 3. until case 1. or case 2. holds, leading to identical signal values. We know that we will finally reach case 1. or case 2. because the partial order is cycle-free and thus we will finally find either a block without any incoming signals or a block that is not executed in time step t.

□

This theorem provides us with the important property that the end state is unique for each simulation step resulting in identical signal values for each run of a TBD transition system. Hence, concatenation of TBD transition systems is well-defined, resulting in also well-defined execution semantics for consecutive simulation steps.

4.2. Translating Simulink

Based on the formalization of a Simulink model as a timed synchronous block diagram and its semantics as a TBD transition system, we will now define the translation of such a model into a function network. Beside the translation of the structure in terms of blocks and signals, we also need to consider the timing properties of the original model that are given in terms of firing time specifications. Thus, the question arises how and when events shall be produced in the function network translation to correctly represent the Simulink timing behavior. For blocks with incoming signals, the respective function nodes are activated as soon as all events of preceding nodes have arrived. Please note that this also covers trigger signals, which are modeled as incoming signals as well. Function nodes of blocks without any incoming signals need to be activated by an external trigger event with the period and offset of the blocks firing time specification. In function networks, events sources are used for producing events with a certain event pattern. Thus, a first step is to define how a firing time specification is translated into an event pattern.

Definition 4.2.1 (FTS Translation) *Let $fts = (per, init)$ be a firing time specification and e be a function network event. The respective event pattern for e is defined as $EP(fts, e) = (\{e\}, per, per, 0, init)$.*

◇

An event source with an event pattern $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ produces the first output event non-deterministically between 0 and $O + P^+ + J$ time units. If we create an event source for each firing time specification of the model, these event sources would not be synchronized. If we assume for example two firing time specifications $fts_1 = (2, 0)$ and $fts_2 = (4, 0)$, which are both active at time $t = 0$, the first event of the event source implementing fts_1 may occur at $t = 0$ and for fts_2 at $t = 4$. Obviously, this is not consistent with Simulink semantics where a synchronization is implicitly assumed and each block execution is assumed to be finished within bp time units. To solve this, we define for the function network translation a single event source ϕ_{bp}, which is running with the base period bp as period. To model the different firing time specifications, special function nodes called *period multiplier* are defined, which are triggered by events from ϕ_{bp} and convert the base period to the period of the respective firing time specification. This assures the synchronization of the initial events of all synchronous sets. A period multiplier is defined as a function node with one input port p^{in}_{fts} and m output ports. An additional output port $p^{\perp}$ is used to represent executions where no output event for other nodes is produced. This port is not part of the interface to other nodes and will not be connected to any channel in the translation. Its transition system is defined such that it multiplies the period P at

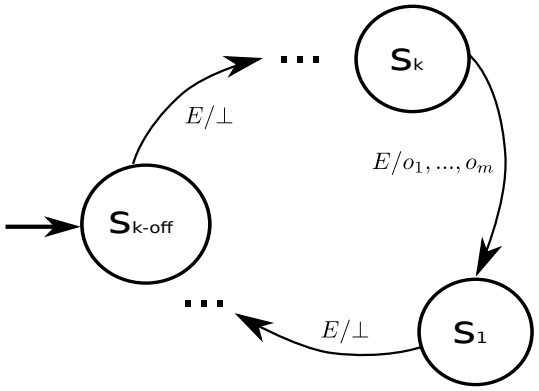


Figure 4.3.: Period Multiplier Transition System

its input port with a factor k while adding an offset off. This offset determines the number of periods until the first event is emitted. Additionally, a delay $\delta = [\delta^-, \delta^+]$ is defined for the execution of transitions.

Definition 4.2.2 (Period Multiplier) *Let $k \in \mathbb{N}^+$ be a factor with $k > 1$, $off \in \mathbb{N}_0$ an offset with $off < k$, $\delta = [\delta^-, \delta^+] \in \mathbb{N}^+ \times \mathbb{N}^+$ a delay and $\{o_1, ..., o_m\}$ a set of output events with $m > 0$. A period multiplier function node is defined as*

$$f^{Mult}(k, off, \{o_1, ..., o_m\}, \delta) := (\{p^{in}_{fts}\}, \mathcal{A}, \{p_{o_1}, ..., p_{o_m}, p^{\perp}\})$$

where $\mathcal{A} = (S, s_0, T)$ with $S = \{s_1, ..., s_k\}$, $s_0 = s_{k-off}$ and

$$\begin{aligned} T = \{\ & (p^{in}_{fts}, E, s_k \rightarrow \{(p_{o_1}, o_1, \delta), ..., (p_{o_m}, o_m, \delta)\}, s_1) \ \mid \ E \in \Sigma^{act}(p^{in}_{fts})\ \} \cup \\ \{\ & (p^{in}_{fts}, E, s_i \rightarrow (p^{\perp}, \perp, \delta), s_{i+1}) \in T \ \mid E \in \Sigma^{act}(p^{in}_{fts}), 1 \leq i < k\ \} \end{aligned}$$

$\diamond$

In Figure 4.3, the transition system of the period multiplier node is depicted, where we restrict for clarity to denote transitions only by their events and omit ports and delays. From the initial state s_{k-off} it needs off input events E until the state s_k is reached while no output event o_i is emitted. Thus, the transitions consume an input event and proceed to the next state while producing a $\perp$ event, which will not be consumed by any other node. Output events o_i are only produced by the transition from s_k to s_1. Thus, the first output event is produced after off input events have occurred. Afterwards, each k^{th} input event produces an output event leading to a period of $k \cdot P$ and an offset of $off \cdot P$ for all output events $o_1, ..., o_m$.

Accordingly, we show in the next lemma how the output language of the output ports of a period multiplier node is derived by referring to the language of input events determined by the input event pattern.

Lemma 4.2.1 (Period Multiplier - Output Port Language) *Let* $f^{Mult}(k, off, \{o_1, ..., o_m\}, [\delta^-, \delta^+]) = (\{p^{in}\}, \mathcal{A}, \{p_1^{out}, ..., p_m^{out}, p^\perp\})$ *be a period multiplier function node and* $EP(\Sigma(p^{in})) = (\Sigma(p^{in}), P, P, J, O)$ *be the event pattern of the input port leading with Lemma 3.1.2 to the timed language*

$$\begin{aligned} L(EP(\Sigma(p^{in}))) = \{\ & (p^{in}.\sigma_1, t_1)...(p^{in}.\sigma_i, t_i)...(p^{in}.\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma(p^{in}), \\ & (1)\ t_i \in [\max(0, (i-1)\cdot P - J), O + (i+1)\cdot P + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m\cdot P - J), O + (m+2)\cdot P + J) \\ \}\ & \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

Then the language for each output port p_j^{out} *with* $j \in \{1, ..., m\}$ *is defined as follows:*

$$L(p_{o_j}) = \{(p_{o_j}.o_j, r_1 + [\delta^-, \delta^+])...(p_{o_j}.o_j, r_i + [\delta^-, \delta^+])...\}$$
$$\textit{where } r_i = t_{1+off+(i-1)\cdot k}$$

Proof: see Lemma B.1.1 in the appendix on page 217. □

Based on the output language, we are able to show that the period multiplier node really produces the event pattern it is supposed to. This means that its output language can be abstracted by an event pattern with a period $k \cdot P$ and an offset of $off \cdot P$.

Lemma 4.2.2 (Period Multiplier - Output Event Pattern) *Let* $f^{Mult}(k, off, \{o_1, ..., o_m\}, [\delta^-, \delta^+]) = (\{p^{in}\}, \mathcal{A}, \{p_{o_1}, ..., p_{o_m}, p^\perp\})$ *be a period multiplier function node and* $EP = (\Sigma^{EP}, P, P, J, O)$ *be the event pattern of the input port. Then the following holds:*

$$\forall j \in \{1, ..., m\} : L(p_{o_j}) \subseteq L(delay((\{p_{o_j}.o_j\}, P\cdot k, P\cdot k, J, O + off\cdot P),\ [\delta^-, \delta^+]))$$

Proof: see Lemma B.1.2 in the appendix on page 217. □

Period multiplier nodes will also be used to model rate transition blocks that lead from a block with a smaller period to a block with a higher period. For rate transition blocks from a higher period to a smaller period, we need external triggers to activate the node when the successor node is not executed. These external triggers are modeled again in terms of period multiplier nodes. To determine the correct event patterns for these additional triggers, we define some operations on firing time specifications. First, we need something we call *FTS Extension* where an fts is extended to a given period. This means, that we represent the same fts as a set of fts's with a different period comparable to expanding a fraction in mathematics.

Definition 4.2.3 (FTS Extension) *Let* $fts = (per, init)$ *be a firing time specification and* $per_{ex} = n{*}per$ $(n \in \mathbb{N}^+)$ *be the period it should be extended to. FTS extension is defined as the following set of firing time specifications:*

$$ex(fts, per_{ex}) = \{(per_{ex}, (init + i\cdot per) \bmod per_{ex}) \mid 0 \le i \le n\}$$

◇

Second, we define the difference between two fts's and the element-relation between fts's as follows:

Definition 4.2.4 (FTS Difference) *Let $fts_a = (per_a, init_a)$ and $fts_b = (per_b, init_b)$ be firing time specifications.*

$$fts_a \setminus fts_b \Longleftrightarrow ex(fts_a, per_b) \setminus \{fts_b\}$$

$\diamond$

Definition 4.2.5 (FTS Element) *Let $fts_a = (per_a, init_a)$ and $fts_b = (per_b, init_b)$ be firing time specifications with $per_a = n * per_b$ $(n \in \mathbb{N}^+)$.*

$$fts_a \in fts_b \Longleftrightarrow fts_a \in ex(fts_b, per_a)$$

$\diamond$

For the Simulink translation, we denote a function node that results from the translation of a block b as f_b. The translation of blocks within the same synchronous set is quite straight forward. Each *combinational*, *sequential* or *Moore-sequential* block is translated to a function node f_b with one input port, and all incoming signals of a block are translated to channels that are synchronized at this input port. Thus, the function node can be first executed if the data of all input signals is available i.e., on each input channel an event has been received. The transition function consists of a single state and one transition that is fired as soon as all input events have arrived. This transition fires all output events while the delay is the worst case execution time of this block. For signals that are produced by Moore-sequential blocks, shared data nodes are created to avoid cyclic causal dependencies.

Each *rate transition* block rt with an input signal in from block a and an output signal out to block b is translated to a function node f_{rt} that converts the firing time specification $fts_a = tr(a) = (per_a, init_a)$ of block a to the firing time specification $fts_b = tr(b) = (per_b, init_b)$ of block b. We assume that per_a and per_b are integer multiples i.e., either $per_a = k * per_b$ or $per_b = k * per_a$ $(k \in \mathbb{N}^+)$ and $init_a = init_b$. This is assured by selecting the Simulink check box "Ensure deterministic data transfer" of the rate transition block.

If $per_a > per_b$, block b requires its input data more often than it is produced by block a. Thus, the rate transition block stores the latest value of the output signal of block a and offers it to block b whenever it is needed. In the function network, this is realized by adding a set of period multiplier nodes triggering f_{rt} whenever it is not triggered by the preceding function node f_a but an activation is required by the successor f_b due to its firing time specification. The set of period multiplier nodes produces events with an event pattern implementing $fts_a \setminus fts_b$, which results in the set of firing time specifications that is needed to compensate the mismatch between fts_a and fts_b. This means, that for each firing time specification $fts \in fts_a \setminus fts_b$ a period multiplier node is created. Coming back to the example from the introduction in Figure 4.1, the block RTB represents a rate transition that leads from sample time $ST_1 = [6, 0]$ to sample time $ST_2 = [2, 0]$. In the function network representation, this

is realized by the function node RTB with input channels from two period multiplier nodes, one for each sample time in the set $ST_2 \setminus ST_1 = \{[6,2],[6,4]\}$.

If otherwise $per_a < per_b$, b must only be activated every k^{th} time where $k = \frac{per_b}{per_a}$. This is realized in the function network by creating a period multiplier node transforming $period_a$ to $period_b$ by applying the factor k. Please note, that for the translation, we assume that rate transition blocks always have exactly one input and one output signal, and that data store memory blocks have each only one data store write block.

Furthermore, we assume that for each block $b \in B$ an execution time is given by a function $wcet : B \to \mathbb{N}^+$. For a block b that is a *data store memory*, *data store read* or *data store write* block, we assume $wcet(b) = \epsilon$ because those blocks represent a local memory where read and write delays are already included in the WCETs of the blocks accessing the memory. Furthermore, we define a function $M : S \to \mathcal{P} \times \Sigma$ mapping signals of *tbd* to combinations of function network ports and events.

Definition 4.2.6 (Translate TBD to Function Network) *Let $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ be a timed block diagram and $wcet : B \to \mathbb{N}^+$ be a function delivering the worst case execution time for each block $b \in B$. Furthermore, we define a function $M : S \to \mathcal{P} \times \Sigma$ mapping signals to combinations of function network ports and events. tbd is translated into a function network $fn_{tbd} = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ as follows:*

1. *Translating firing time specifications: Let bp be the base period with $fts_{bp} = (bp, 0)$ and $\mathcal{FTS} = \{fts_1, ..., fts_n\}$ be the set of all firing time specifications of tbd. We define a source node*

$$\phi_{bp} = (EP(fts_{bp}, tr), \{p^{\phi_{bp}}_{fts_1}, ..., p^{\phi_{bp}}_{fts_n}\}) \in \Phi.$$

The set of global trigger events produced by ϕ_{bp} is defined as

$$TR_{bp} = \{p^{\phi_{bp}}_{fts_1}.tr, ..., p^{\phi_{bp}}_{fts_n}.tr\}.$$

For each firing time specification $fts \in \mathcal{FTS}$, a period multiplier node f_{fts} is defined that creates the respective firing time specification. Let $b_1, ..., b_m$ be all blocks without any input signals (except from Moore-sequential blocks) that have the firing time specification fts i.e.

$$\forall j \in \{1, ..., m\} : \; tr(b_j) = fts \; \wedge \\ \nexists (b', s, b_j) \in E \mid type(b') \neq \text{'Moore-sequential'}$$

Then f_{fts} is defined as follows:

$$\forall fts = (per, init) \in \mathcal{FTS} : \\ f_{fts} = f^{Mult}\left(\frac{per}{bp}, \frac{init}{bp}, \{tr_{b_1}, ..., tr_{b_m}\}, [\epsilon, \epsilon]\right) \in \mathcal{F}.$$

Each period multiplier node f_{fts} is connected by a channel $(p^{\phi_{bp}}_{fts}, p^{in}_{fts}) \in \mathcal{C}^A$ from an output port of ϕ_{bp} to the input port of f_{fts}. Furthermore, we define a channel to each triggered block i.e., $\forall j \in \{1, ..., m\} : (p_{tr_{b_j}}, p^{in}_{b_j}) \in \mathcal{C}^A$.

2. *Translating blocks:*

a) *For each block b with $type(b) \in$ {'combinational','sequential','moore-sequential'} with $m > 0$ output signals $out_1, ..., out_m$ a function node f_b is defined with*

$$f_b = (\{p_b^{in}\}, (\{s_0\}, s_0, T), \{p_{out_1}, ..., p_{out_m}\}) \in \mathcal{F} \text{ where}$$
$$T = \{(p_b^{in}, E, s_0 \rightarrow \{(p_{out_1}, out_1, \delta), ..., (p_{out_m}, out_m, \delta)\}, s_0) \mid E \in \Sigma^{act}(p_b^{in})\}$$

The mapping function for all output signals is defined as follows:

$$\forall j \in \{1, ..., m\} : M(out_j) := p_{out_j}.out_j.$$

Please note, that $\Sigma^{act}(p_b^{in}) = \{E\}$ has exactly one element, because each channel transports exactly one event.

b) *Let $rt \in B$ be a block with $type(b)$ = 'rate-transition', an input edge $(a, in, rt) \in E$ from block a via signal in with $fts_a = tr(a) = (per_a, init_a)$ and an output edge $(rt, out, b) \in E$ via signal out to block b with $fts_b = tr(b) = (per_b, init_b)$. Let furthermore $per_a = n * per_b$ or $per_b = n * per_a$ $(n \in \mathbb{N}^+)$ and $init_a = init_b$. Then rt is translated to a function node $f_{rt} \in \mathcal{F}$ as follows:*

i. *If $per_a > per_b$, for each $fts_i = (per_i, init_i) \in fts_a \setminus fts_b = \{fts_1, ..., fts_n\}$ f_{rt} has an input port $p_{rt_i}^{in}$ that is triggered by an activation channel from a period multiplier*

$$f_i^{Mult} = f^{Mult}\left(\frac{per_i}{bp}, \frac{init_i}{bp}, \{tr_{rt}\}, [\epsilon, \epsilon]\right) \in \mathcal{F}$$

where bp is the base period. Each f_i^{Mult} has an incoming activation channel from ϕ_{bp}. The transition system contains a transition leading from each $p_{rt_i}^{in}$ to p_{rt}^{out} and from p_{rt}^{in} to p_{rt}^{out} i.e.

$$f_{rt} = (\{p_{rt}^{in}, p_{rt_1}^{in}, ..., p_{rt_n}^{in}\}, (\{s_0\}, s_0, T), \{p_{rt}^{out}\})$$

where

$$\begin{aligned} T \;=\; & \{\; (p_{rt_1}^{in}, tr_{rt_1}, s_0 \rightarrow \{(p_{rt}^{out}, out, \delta)\}, s_0), ... \\ & \quad (p_{rt_n}^{in}, tr_{rt_n}, s_0 \rightarrow \{(p_{rt}^{out}, out, \delta)\}, s_0) \;\} \cup \\ & \{\; (p_{rt}^{in}, E, s_0 \rightarrow \{(p_{rt}^{out}, out, \delta)\}, s_0) \mid E \in \; \Sigma^{act}(p_{rt}^{in})\} \end{aligned}$$

with $\delta = [wcet(rt), wcet(rt)]$. We define $M(out) := p_{rt}^{out}.out$.

ii. *If $per_a < per_b$, f_{rt} is defined as a period multiplier*

$$f_{rt} := f^{Mult}(k, off, \{out\}, [wcet(rt), wcet(rt)])$$

with $k = \frac{per_b}{per_a}$ and $off = 0$. We define $M(out) := p_{out}.out$.

c) *For each block* $b \in B$ *with* $type(b) =$ *'data store memory', a shared data node* $d_b = (\mathcal{P}^{in}, \delta, \sigma_0, \mathcal{P}^{out}) \in \mathcal{D}_{shared}$ *is defined where* σ_0 *is the initial value of* b *and* $\delta = [\epsilon, \epsilon]$. *For each block* b_w *writing to* b *with* $type(b_w) =$ *'data store write' and an edge* $(b', s_w, b_w) \in E$, *we define an input port* $p_w^{in} \in \mathcal{P}^{in}$ *with an incoming channel* $(p_w, [0,0], p_w^{in}) \in \mathcal{C}^A$. *For each block* b_r *reading from* b *with* $type(b_r) =$ *'data store read' and an edge* $(b_r, s_r, b'') \in E$, *we define an output port* $p_r^{out} \in \mathcal{P}^{out}$ *with a read channel* $(p_r^{out}, [0,0], p_r) \in \mathcal{C}^R$.

d) *For each* sink *block* b, *which is a block with* $n > 0$ *inputs and* $m = 0$ *outputs, a function node* f_b *is defined as*

$$f_b = (\{p_b^{in}\}, (\{s_0\}, s_0, T), \{p_b^{\perp}\}) \in \mathcal{F} \text{ where}$$
$$T = \{(p_b^{in}, E, s_0 \to \{(p_b^{\perp}, \perp, [wcet(b), wcet(b)])\}, s_0) \mid E \in \Sigma^{act}(p_b^{in})\}$$

3. *Translating edges and signals: Let* $e = (b_1, s, b_2) \in E$ *be an edge with* $M(s) = p_s.s$ *and* $f_{b_1} = (\mathcal{P}_{b_1}^{in}, \mathcal{A}_{b_1}, \mathcal{P}_{b_1}^{out})$, $f_{b_2} = (\mathcal{P}_{b_2}^{in}, \mathcal{A}_{b_2}, \mathcal{P}_{b_2}^{out})$ *be the function nodes the blocks* b_1 *and* b_2 *are translated to with* $p_s \in \mathcal{P}_{b_1}^{out}$ *and* $p_{b_2}^{in} \in \mathcal{P}_{b_2}^{in}$.

a) *If* b_1 *is a Moore-sequential block,* e *is translated to a shared data node* $d = (\{p_d^{in}\}, [\epsilon, \epsilon], d_{shared}, \{p_d^{out}\})$ *with an incoming activation channel* $c_a = (p_s, [0,0], p_d^{in}) \in \mathcal{C}^A$ *from* f_{b_1} *and an outgoing read channel* $c_r = (p_d^{out}, [0,0], p_{b_2}^{in}) \in \mathcal{C}^R$ *to* f_{b_2}.

b) *Otherwise,* e *is translated to an activation channel* $c = (p_s, [0,0], p_{b_2}^{in})$ *leading from* f_{b_1} *to* f_{b_2}.

◇

For function networks that result from the proposed translation of Simulink models, boundedness can always be decided. First, all sources are periodic and each function node is state-independent because it has only one state, except for period multiplier nodes. In the algorithm to decide boundedness from Def. 3.4.4 of Section 3.4, we need the property of state-independence only for cyclic causal dependencies. Function networks derived from block diagrams are always cycle-free by definition because algebraic loops are not allowed. Thus, to decide boundedness, it is sufficient to be able to propagate event patterns also for period multiplier nodes. This has been shown in Lemma 4.2.2. Please note that the state-independence of function nodes is only given because we abstract from the internal functionality of single blocks in the translation. For a translation that considers the internal behavior of blocks and also the dynamic triggering of blocks the concept of event pattern propagation would need to be extended to cover also those systems.

Furthermore, we know for function networks translated from Simulink models that synchronization buffers of input ports are bounded by definition. This is because within a synchronous set, all blocks have the same sample time and thus the periods of connected blocks are equal leading to bounded buffers (see Lemma 3.4.5). For communication between different synchronous sets via rate transitions, only single

signals are allowed leading to bounded buffers as well. To guarantee boundedness for activation buffers, it needs to be assured that any WCET of a block is smaller than its period (see Lemma 3.4.7). In the next section, we will define *feasible* timed block diagrams, where this property is given.

4.3. Preserving Semantics

To preserve semantics, the translated function network has to respect the partial order of block executions of the respective TBD and all block executions of a simulation step have to be finished before the next simulation step starts. The basic idea to show the first part is to relate signal updates, which occur due to block executions, to events in the function network translation.

In Simulink, a signal is updated whenever its producing block is executed. Thus, we define the set of signals a block updates as all output signals it produces.

Definition 4.3.1 (Signal Update) *Let* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *be a TBD. The update function returns the set of signals that is updated by a block* b *i.e.*

$$upd(b) = \{s \mid (b, s, b') \in E\}$$

◇

The simulation steps where signal updates occur are determined by the partial order of blocks i.e., a signal is updated whenever its producing block is executed. Accordingly, we define a partial order on signals - meaning signal updates - for a specific simulation step t by considering the partial order of blocks and the previous definition of a signal update.

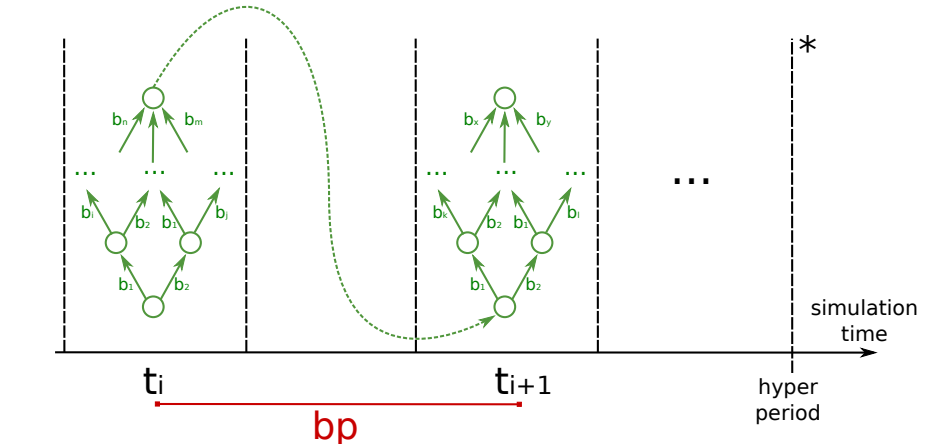
Definition 4.3.2 (Partial Order on Signals) *Let* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *be a TBD. The partial order on signals* $PO_S(tbd, t)$ *for simulation step* t *is defined as follows:*

$$(s_1, s_2) \in PO_S(tbd, t) \iff \exists s_1 \in upd(b_1),\ s_2 \in upd(b_2) : (b_1, b_2) \in PO_B(tbd, t)$$

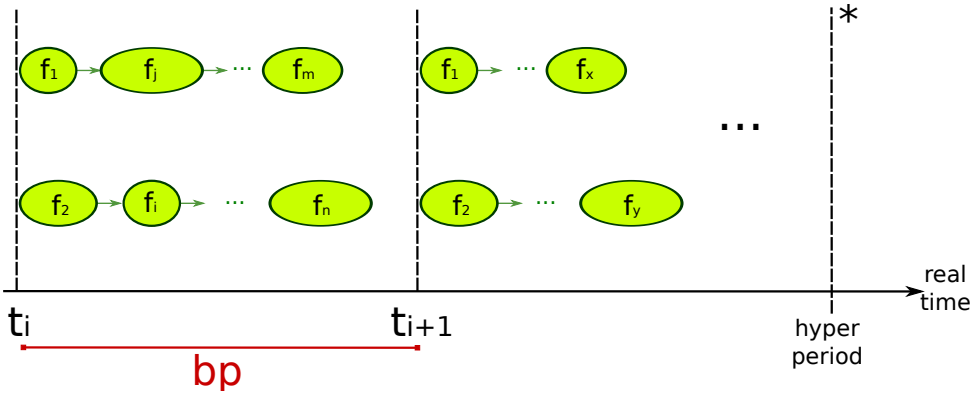
We write $s <_t s'$*, if* $(s, s') \in PO_S(tbd, t)$. ◇

To be able to relate events in the function network to signals updates, we need to assign events to simulation steps. To avoid overlapping simulation steps and thus a deadline violation, an event of a function network can only be part of the partial order of a simulation step t, if it occurs before the next simulation step starts. The length of a simulation step is determined as the base period bp i.e., each event of a simulation step t must occur within a time interval of $[t, t + bp)$. This means that all timing constraints are met if the partial order is preserved. As soon as an event occurs later than its deadline of bp time units, it is not part of the partial order of that simulation step and thus semantics would not be preserved.

This is illustrated in Figure 4.4a, where the execution semantics of a Simulink model is depicted in terms of a transition system for each simulation step. In Figure 4.4b, the



(a) Simulink Semantics



(b) Function Network Semantics

Figure 4.4.: Preserving Semantics from Simulink to Function Networks

execution of the function network translation is sketched. For each simulation step t_i, a set of function nodes needs to be executed with respect to the partial order of block executions. In contrast to Simulink block executions, in the function network each node execution takes time. Thus, we need to assure that at the end of a simulation step all function nodes that need to be executed have finished their execution. This leads to an end-to-end deadline with a length of bp time units for each path starting at source events and ending at events received by sink nodes.

To reason about the occurrences of events in a function network, we use event patterns. Comparable to the property $rdy(b, t)$ for a Simulink block b, we define when an event pattern is active with respect to a simulation step t. In contrast to Simulink, we additionally need to consider the execution delays of function nodes. This leads to an increase of the offset and possibly the jitter with each event pattern propagation

step from one node to its successor node. Thus, the sum of offset and jitter must not become greater than the deadline *bp*, where we subtract (t mod P), which is equal to the initial phase shift *init*. For Simulink translations we know that $J = 0$ because lower and upper delay bounds are always identical. This means that an offset that was initially determined to $O = init$ may only be increased by a value smaller than *bp* to be active in time step t. Furthermore, in the language of event patterns, the offset may occur at any time while we know from the definition of source nodes that the offset always occurs initially at system startup. This is also the case for sample times in Simulink. Thus, we must assure that the offset occurs initially.

We define this property also for superposition of event patterns. A superposition is active as soon as at least one of the superposed event patterns is active.

Definition 4.3.3 (Active Event Pattern) *Let bp be the base period and Σ^{EP} a set of events with $L(\Sigma^{EP}) = (e_1, r_1)(e_2, r_2)...$ were $e_i \in \Sigma^{EP}$. A periodic event pattern $EP = (\Sigma^{EP}, P, P, J, O)$ is active in a simulation step t if*

$$active(EP, t) \iff 0 \leq O + J - (t \bmod P) < bp \wedge r_1 \geq O$$

We define superposition of event patterns to be active if the following holds:

$$active(super(EP_1, ..., EP_n), t) \iff \exists i \in \{1, ..., n\} : active(EP_i, t)$$

◇

Now we are able to define a partial order on function network events, which we will use to show semantics preservation of the translation. To be part of the partial order, two events e and f have to satisfy three properties: There must exist a causal dependency from e to f, the event pattern of f must be active at time t, and the distance of f to an initial trigger event $tr_{bp} \in TR_{bp}$ from the event source of the network must be smaller than *bp*. The last property assures that the next time step does not start before each node execution of the previous time step has finished.

We need to claim this additionally for the *active* property because the event pattern abstracts from the common event source. Thus, we need to explicitly show that all blocks have a causal dependency to an event of this source with a maximum delay of *bp* time units. This causal dependency may have a condition meaning that f does not occur each time a trigger event has occurred. This is the fact for all events that do not occur with the base period. The correctness of this condition and hence the correct period and offset of an event is assured if the respective event pattern is active.

The partial order on function network events is defined as follows:

Definition 4.3.4 (Partial Order on Events) *Let tbd be a TBD, t be a Simulink simulation step and bp be the base period of tbd. Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be the function network translation of tbd, $e, f \in \Sigma$ be events and $EP(e)$ the event pattern of e where it holds $active(EP(e), t)$. Let furthermore $tr_{bp} \in TR_{bp}$ be an initial trigger*

event from the event source ϕ_{bp} producing events with the base period. Then the partial order $PO_\Sigma(fn, t)$ is defined as follows:

$$\begin{aligned}(e, f) \in PO_\Sigma(fn, t) \iff & (1)\ e[cond_e] \xrightarrow{[min,max]} f \ \wedge \\ & (2)\ active(EP(f), t) \ \wedge \\ & (3)\ \exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond_{bp}] \xrightarrow{[min,max]} f\ , max < bp\end{aligned}$$

We write $e <_t f$, if $(e, f) \in PO_\Sigma(fn, t)$. ◇

As a necessary condition for the function network to be able to preserve the timing constraints of the Simulink model, we define *feasibility* for TBDs. A TBD is feasible at a simulation step t if all blocks that need to be executed successively (due to the partial order on block executions) in step t have finished their execution before the next simulation steps starts. This corresponds to a set of end-to-end deadlines over maximal block chains leading from each start block b_1 (a block without any predecessor in the partial order) to each end block b_n (a block without any successor in the partial order) whose execution depends on b_1 i.e. $b_1 <_t ... <_t b_n$. Thus, the sum of WCETs of each such maximal chain of partially ordered blocks must be smaller than the base period *bp*. For this definition, we assume that WCETs are given for each block and that each two blocks that are not partially ordered may be executed concurrently. A TBD is (completely) feasible if it is feasible at all simulation steps t meaning all simulation steps until reaching the hyper period where the execution behavior repeats. If a TBD is not feasible, also its function network translation cannot preserve the timing constraints assumed by Simulink.

Definition 4.3.5 (TBD Feasibility) *Let $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ and $wcet : B \to \mathbb{N}^+$ be a total function that determines for each block its execution time and let bp be the base period of tbd. Feasibility of a tbd is defined as follows:*

$$\begin{aligned}& tbd \text{ is feasible at time } t \\ \iff\ & \forall (b_1, b_2), ..., (b_{n-1}, b_n) \in PO_B(tbd, t) \mid \nexists (b, b_1), (b_n, b') \in PO_B(tbd, t) : \\ & (wcet(b_1) + ... + wcet(b_n)) < bp\end{aligned}$$

tbd is (completely) feasible if and only if tbd is feasible at all time steps t. ◇

To be able to guarantee that a feasible TBD leads to a function network that satisfies the timing constraints, we need to show that no additional delays are induced by the function network translation. The worst case execution times of blocks are translated to transition delays of function nodes and thus there are no additional delays in the transition system. Also channels between function nodes always have a zero delay. Hence, the only place where additional delays may be induced are synchronization buffers of function nodes. For synchronization buffers of input ports it holds that we always have to wait until all signals of preceding nodes have been produced. This effect is already considered in the definition of feasibility by taking the sum of delays of each maximal chain of block executions, which also includes the longest of those chains.

Read channels from shared data nodes can be omitted for this consideration, because delays of read channels are zero and the delays of shared data nodes are assumed to be negligibly small denoted by *epsilon*. Thus, if the TBD is feasible, we will not get additional delays in synchronization buffers of input ports. What is left, is the activation buffer of a function node. To show feasibility, we need to assure that no input event arriving at the activation buffer has to wait for a $start_f$ event and thus a preceding execution to be finished.

To prepare this proof, we show that, under a specific condition, the wait delay for start events is always zero and can thus be omitted in the causality pattern. This condition states that the minimum inter-arrival time between any two input events is greater than the maximum delay of any transition of the transition system. In this case, the execution of the function node is always terminated before the next input event arrives and thus a $start_f$ event is always available. Hence, we never have to wait for it leading to a wait delay of zero.

Lemma 4.3.1 (No Wait Delay for Start Event) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ be a function node and let the output language of the activation buffer $L(EP(I_p))$ be defined as*

$$\begin{aligned} L(EP(I_p)) = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in I_p, \\ & (1)\ t_i \in [\max(0, (i-1)\cdot P^- - J), O + (i+1)\cdot P^+ + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m\cdot P^- - J), O + (m+2)\cdot P^+ + J) \\ \} & \ \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

According to Lemma 3.4.6 the language of start events is determined as

$$\begin{aligned} L(start_f) = \{\ & (start_f, u_1)...(start_f, u_i)...(start_f, u_{i+n})...\} \\ \mid\ & u_1 = 0, \\ & u_{i+1} = \max(u_i, t_i) + [\delta^{min}, \delta^{max}] \end{aligned}$$

Let further be δ^{max} be the maximum delay of any transition $t \in T$. Then it holds:

$$\begin{aligned} \Delta^-(L(I_p)) > \delta^{max} \implies\ & \forall i \in \mathbb{N}^+ :\ u_i \leq t_i\ \wedge \\ & wait_{start} = \Delta(L(I_p), L(start_f)) = 0 \end{aligned}$$

Proof: see Lemma B.2.1 in the appendix on page 219. □

Now it remains to show that the condition that leads to a waiting time of zero for the start event is always satisfied for function networks that arise from a Simulink translation. This is the case, because we claim that all blocks that are executed in a simulation step have finished their execution before the next simulation step starts leading to a deadline equal to the base period *bp*. Thus, we have no overlapping executions of the same node and a $start_f$ event is always available.

Lemma 4.3.2 (No Wait Delay for TBD Function Nodes) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *and a block* $b \in B$ *with n input signals. Let* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be the function network translation of tbd with a respective function node* $f_b \in \mathcal{F}$ *with one input port with* $n > 1$ *incoming activation channels, a maximum transition delay* $\delta^{max} = wcet(b)$*, and the following causal dependencies on input events* $in_1, ..., in_n$ *and* in_j *with* $j \in \{1, ..., n\}$ *and trigger events* $tr_{bp} \in TR_{bp}$*:*

(1) $\{p_{in_1}.in_1, ..., p_{in_n}.in_n\} \xrightarrow{[wait^-_{start}, wait^+_{start}]} ((p_{in_1}.in_1, ..., p_{in_n}.in_n), start_f) \wedge$

(2) $p_{in_j}.in_j \xrightarrow{[wait^-_{in}+wait^-_{start}, wait^+_{in}+wait^+_{start}]} ((..., p_{in_j}.in_j, ...), start_f)$

(3) $\forall j \in \{1, ..., n\},\ \forall tr_{bp} \mid tr_{bp}[cond_{bp}] \xrightarrow{[\delta_j, \delta_j]} p_{in_j}.in_j :\ \delta_j < bp$

Then it holds:

$$wait^+_{start} = 0,\ wait^-_{start} = 0,\ wait^+_{in_j} < bp - \delta^{max}$$

Proof: see Lemma B.2.2 in the appendix on page 220. □

We show semantics preservation inductively over a maximal chain of partially ordered signals $s_1 <_t ... <_t s_n$ of any simulation step t. What we need to show is that the respective events in the function network are partially ordered as well. An update on a signal s is represented in the function network as an event $M(s)$, where M is the mapping function defined in the translation. Furthermore, we need to consider the source event $tr_{bp} \in TR_{bp}$, which triggers the period multiplier node representing the firing time specification of the first signal s_1. Additionally, an event $p_{tr_{b_1}}.tr_{b_1}$ must occur triggering the producer block of s_1, which is b_1. Thus, for a maximal chain of partially ordered signals $s_1 <_t ... <_t s_n$ we need to show that there exists a respective chain in $PO_\Sigma(fn, t)$ i.e.

$$tr_{bp} <_t p_{tr_{b_1}}.tr_{b_1} <_t M(s_1) <_t .... <_t M(s_n).$$

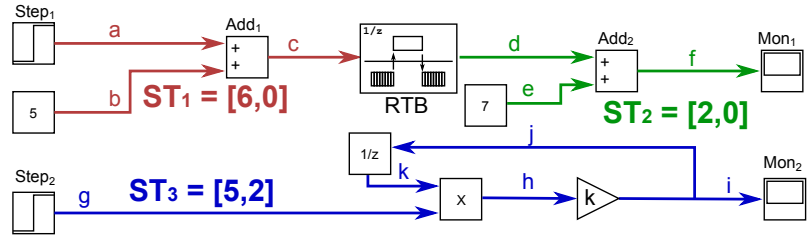
In Figure 4.5, the previous example of a Simulink translation into a function network is annotated with signals and events. As an example, we consider the maximal chain

$$a <_t c <_t d <_t f$$

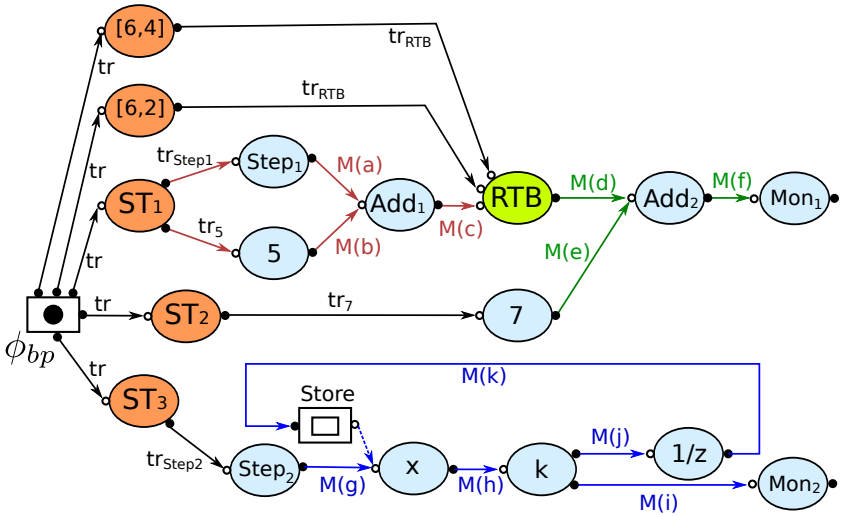
of partially ordered signals in the Simulink model of Figure 4.5a. In the function network translation of Figure 4.5b, we need to show that also the respective events are partially ordered and that the needed trigger events occur i.e.

$$tr <_t tr_{Step_1} <_t M(a) <_t M(c) <_t M(d) <_t M(f).$$

Because the proof works inductively, we first show that the partial order is preserved for each translation of a specific block type, and afterwards show the correctness for any maximal chain in the partial order. To be able to put the single proofs together, we will make some assumptions about the partial ordering of previous signals and



(a) Signals in a Simulink Block Diagram



(b) Function Network Events representing Signal Updates

Figure 4.5.: Preserving partial order in function network translation

events, which will be satisfied later in the inductive proof. There are three types of blocks that are translated differently to function nodes: rate-transition blocks, source blocks and all remaining blocks, which we summarize as *ordinary* blocks. Furthermore, we use period multiplier function nodes to model the needed firing time specifications leading to four different function node types to be considered. Furthermore, there are data store memory blocks, which do not induce a partial order on blocks. Thus, they are not relevant for semantics preservation and will not be considered in the following considerations and proofs.

We start with period multiplier nodes, which are connected by a channel to the source node ϕ_{bp} of the function network and convert the base period bp to a specific firing time specification used in the model. An example for a period multiplier node is the node labeled with ST_1 in Figure 4.5b. What we need to show here is that the execution of each source block b_1 is initiated correctly in the function network. Thus, there must exist a partial order from a trigger event $tr_{bp} \in TR_{bp}$ of the event source

ϕ_{bp} to the event $p_{tr_{b_1}}.tr_{b_1}$, which starts the execution of the function node representing block b_1. By applying the partial order definition for function network events from Def. 4.3.4, this leads to two statements to prove: First, it must hold that the event pattern of $(p_{tr_{b_1}}.tr_{b_1})$ is active at the considered simulation step t. This assures that there occurs an event starting the execution within this simulation step. To show this, we need to consider the definition of the period multiplier function node and the resulting event pattern at its output. By having chosen the appropriate parameters in the translation, we get the same period and offset as the firing time specification of b_1 claims. Thus, we can prove that the event pattern is active at each point in time t where $rdy(b_1, t)$ holds. Second, there must exist a causal dependency from tr_{bp} to $p_{tr_{b_1}}.tr_{b_1}$ with a delay smaller than the base period bp. This can be shown by considering the transition system of the period multiplier node. The worst case execution time of period multiplier nodes that are no rate transition blocks is always ϵ. Thus, we assume that the delay for converting firing time specifications is negligible small and thus does not influence the feasibility. The third condition, we need to show for the partial order, is already covered by the previous causal dependency.

Lemma 4.3.3 (Preserve Partial Order - Period Multiplier Nodes) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$, $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *its function network translation and* t *be simulation step. Let* s_1, $s_2 \in S$ *be signals with* $(b_1, s_1, b_2) \in E$, $(b_2, s_2, b_2') \in E$, $(s_1, s_2) \in PO_S(tbd, t)$ *and* $\nexists (s, s_1) \in PO_S(tbd, t)$. *Let furthermore* $fts = tr(b_1) = (per, init)$ *be the firing time specification of block* b_1, *and* tr_{b_1} *be a trigger event of* b_1 *produced by the period multiplier block* $f_{fts} = f^{Mult}(\frac{per}{bp}, \frac{init}{bp}, \{..., tr_{b_1}, ...\})$ *at port* $p_{tr_{b_1}}$. *Then the following holds:*

$$\exists tr_{bp} \in TR_{bp} : (tr_{bp}, p_{tr_{b_1}}.tr_{b_1}) \in PO_\Sigma(fn, t)$$

Proof: see Lemma B.2.3 in the appendix on page 221. □

The first block type that may occur in a maximal chain of partial ordered blocks is a source block. There are two block types which may act as source blocks. First, it may be a block without any input signals except from Moore-sequential blocks. The respective function node has no activation channels from any other nodes and is thus directly connected to the period multiplier node that creates the respective firing time specification of the block. An example is block $Step_1$ from Figure 4.5a. Second, a source block may also be a rate transition block if the preceding block is not executed in the considered simulation step t.

In this case, again a period multiplier node triggers the function node representing this rate transition block. An example is block RTB from Figure 4.5a. For both cases, we show in the following lemma that a function node modeling a source block b_1 induces a partial order on its input and output events. The input event is a trigger event $p_{tr_{b_1}}.tr_{b_1}$ from the period multiplier node creating the firing time specification of b_1. The output event is the event corresponding to the signal s_1 produced by b_1, which is determined by the mapping function to $M(s_1)$.

Lemma 4.3.4 (Preserve Partial Order - Source Blocks) *Let tbd be a feasible TBD with tbd=$(B, type, S, E, \mathcal{FTS}, tr)$, fn = $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ its function network translation and t be a simulation step. Let $s_1 \in S$ be a signal with $(b_1, s_1, b') \in E$ and $\nexists (s, s_1) \in PO_S(tbd, t)$. Then it holds:*

$$\begin{aligned} &(1)\ \exists tr_{bp} \in TR_{bp} : (tr_{bp}, p_{tr_{b_1}}.tr_{b_1}) \in PO_\Sigma(fn, t) \\ \implies &(2)\ (p_{tr_{b_1}}.tr_{b_1}, M(s_1)) \in PO_\Sigma(fn, t) \end{aligned}$$

Proof: see Lemma B.2.5 in the appendix on page 223.

□

Another special block is the rate transition block which converts a firing time specification fts_a of a block a to a firing time specification fts_b of a block b. An example for a rate transition block is block RTB from Figure 4.5a. Depending on the respective periods per_a and per_b, a rate transition block is either translated to a period multiplier node or a function node with additional input channels from a set of period multiplier nodes. For both cases, we show that the partial order of two signals s and s' also holds for the respective function network events $M(s)$ and $M(s')$.

In the following statements and proofs, we denote by *wcets* the sum of WCETs of block executions that occur between a trigger event of the event source and an event $M(s)$. For feasible TBDs, *wcets* is smaller than the base period *bp* by definition. The delay needed until we see an event $M(s')$, which represents the update of the successor signal, is denoted as $wcets'$. It is determined by the execution time $wcet(b)$ of the block b that produces s' and the waiting time $wait_{in}$ for synchronization. This delay is smaller than *bp* as well because it is again the sum of WCETs of a chain of partially ordered blocks.

Additionally, there occurs an ϵ delay in the causal dependencies from the trigger event to a signal update event, which results from the period multiplier node that was executed before. Please note, that for rate transition blocks, $wait_{in}$ is always zero because those blocks are assumed to have exactly one input channel and thus no synchronization is needed.

Lemma 4.3.5 (Preserve Partial Order - Rate Transition Blocks) *Let tbd be a feasible TBD with tbd = $(B, type, S, E, \mathcal{FTS}, tr)$, fn = $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ its function network translation and t be a simulation step. Let $(s, s') \in PO_S(tbd, t)$ and b be a block with $s' \in upd(b)$ and type(b) = 'rate-transition' connecting the blocks a and b with the firing time specifications $fts_a = tr(a) = (per_a, init_a)$ and $fts_b = tr(b) = (per_b, init_b)$. Let further the following hold:*

$$\begin{aligned} &active(EP(M(s), t) \wedge \\ &tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets, \epsilon+wcets]} M(s),\ wcets < bp \end{aligned}$$

Then it holds:

$$(1)\ M(s)[cond_{M(s)}] \xrightarrow{[wcet(b),wcet(b)]} M(s') \wedge$$
$$(2)\ active(EP(s'),t) \wedge$$
$$(3)\ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s'),\ wcets' < bp$$

Proof: see Lemma B.2.6 in the appendix on page 225. □

All remaining block types are summarized as ordinary blocks in the translation. Ordinary blocks are sequential, Moore-sequential or combinational blocks with at least one input and output signal. An example for an ordinary block is block Add_1 from Figure 4.5a. In the case of multiple input signals, the respective channels in the function network translation need to be synchronized, which may lead to a waiting time $wait_{in}$ greater than zero. The waiting time is determined as the maximum sum of preceding block execution times. In the following lemma, we show for an ordinary block b inducing a partial order on two signals s and s' that this partial order is preserved for the respective function network events $M(s)$ and $M(s')$.

Lemma 4.3.6 (Preserve Partial Order - Ordinary Blocks) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$*,* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *its function network translation and t be a simulation step. Let* $(s, s') \in PO_S(tbd, t)$ *and b be a block with* $s' \in upd(b)$*, n input signals* $in_1, ..., in_n$*, where* $\exists j : in_j = s$ *and* $type(b) \in \{$*'sequential', 'Moore-sequential', 'combinational'*$\}$*.*

Let $f_b = (\{p_b^{in}\}, (\{s_0\}, s_0, \{t\}), \{p_{out_1}, ..., p_{out_m}\}) \in \mathcal{F}$ *be the translation of b and* $M(s) = p_s.s$ *as defined in Def. 4.2.6. Let further the following hold:*

$$(A)\ active(EP(M(s),t)) \wedge$$
$$(B)\ \exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets,\epsilon+wcets]} M(s),\ 0 \leq wcets < bp$$

Then it holds:

$$(1)\ M(s) \xrightarrow{[wait_{in}+wcet(b),wait_{in}+wcet(b)]} M(s') \wedge$$
$$(2)\ active(EP(M(s')),t) \wedge$$
$$(3)\ \exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s'),\ wcets' < bp$$

Proof: see Lemma B.2.7 in the appendix on page 227. □

Now we can put everything together and show that the function network translation preserves the semantics of the TBD with respect to the partial order of signals and events. We start by showing that function nodes translated from source nodes are correctly triggered by events from the event source. Then, we show by induction over a maximal chain of partially ordered signals that the respective function network events are partially ordered as well.

Theorem 4.3.1 (Translation preserves Partial Order of Signals) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$*,* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *its function network translation and* t *be a simulation step. Let* $fts(s)$ *denote the firing time specification of the block that produces the signal* s*. Then it holds:*

$$
\begin{aligned}
& (1)\ (s_1, s_2), ..., (s_{n-1}, s_n) \in PO_S(tbd, t)) \ \wedge \\
& \quad \nexists (s, s_1) \in PO_S(tbd, t) \ \wedge \ \nexists (s_n, s) \in PO_S(tbd, t) \\
\implies & (2)\ (a)\ \forall i \in \{1, ..., n-1\} : (M(s_i), M(s_{i+1})) \in PO_\Sigma(fn, t) \ \wedge \\
& \quad (b)\ \exists tr_{bp} \in TR_{bp} : (tr_{bp}, p_{tr_{b_1}}.tr_{b_1}) \in PO_\Sigma(fn, t) \ \wedge \\
& \quad (c)\ (p_{tr_{b_1}}.tr_{b_1}, M(s_1)) \in PO_\Sigma(fn, t) \ \wedge \\
& \quad (d)\ \nexists (e, tr_{bp}) \in PO_\Sigma(fn, t) \ \wedge \\
& \quad (e)\ \nexists (M(s_n), e) \in PO_\Sigma(fn, t)
\end{aligned}
$$

Proof: (b) was proven in Lemma 4.3.3, (c) was proven in Lemma 4.3.4. (d) follows directly by the fact that any tr_{bp} *is produced by source node* ϕ_{bp}*. (e) follows from the fact that block* b_n*, which produces* s_n*, is a block without any outputs and thus is translated to a function node without any output channels. Thus, there is no causal dependency starting at* $M(s_n)$*. Now it remains to prove (a) which we will do by induction over* i*:*

1. *Base Case (i=1):*

   $$(M(s_1), M(s_2)) \in PO_\Sigma(fn, t)$$

   With Def. 4.3.4, we need to show all the following:

   $$
   \begin{aligned}
   & active(EP(s_2), t) \ \wedge \\
   & M(s_1)[cond_{M(s_1)}] \xrightarrow{[delay, delay]} M(s_2) \ \wedge \\
   & \exists tr_{bp} \in TR_{bp} : \ tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon + wcets', \epsilon + wcets']} M(s_2), \ wcets' < bp
   \end{aligned}
   $$

   We can conclude from (b), (c) and Def. 4.3.4 that the following holds:

   $$active(EP(s_1), t) \ \wedge \ tr_{bp}[cond] \xrightarrow{[\epsilon + wcets, \epsilon + wcets]} M(s_1), \ wcets < bp$$

   If b_2 *is a sequential, Moore-sequential or combinational block, we can apply Lemma 4.3.6. If* b_2 *is a rate transition block, we can apply Lemma 4.3.5 to show the base case.* b_2 *cannot be a block without any outputs and no data store write block because then there would not exist any signal* s_2*. It can also not be a block without any inputs or a data store read block because then there would not exist an* s_1*.*

2. *Inductive Step: We assume that the statement holds for* i *and show that it also holds for* $i + 1$ *with* $i \in \{1, ..., n-2\}$ *i.e.:*

   $$(M(s_i), M(s_{i+1})) \in PO_\Sigma(fn, t) \implies (M(s_{i+1}), M(s_{i+2})) \in PO_\Sigma(fn, t)$$

With Def. 4.3.4, we need to show the following:

$$
\begin{aligned}
& active(EP(s_{i+1}), t) \wedge \\
& M(s_i)[cond_{M(s_i)}] \xrightarrow{[delay,delay]} M(s_{i+1}) \wedge \\
& tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets,\epsilon+wcets]} M(s_{i+1}),\ wcets < bp \\
\Longrightarrow & \\
& active(EP(s_{i+2}), t) \wedge \\
& M(s_{i+1})[cond_{M(s_{i+1})}] \xrightarrow{[\delta'^{-},\delta'^{+}]} M(s_{i+2}) \wedge \\
& tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s_{i+2}),\ wcets' < bp
\end{aligned}
$$

If b_{i+2} is a sequential, Moore-sequential or combinational block, we can apply Lemma 4.3.6. If b_{i+2} is a rate transition block, we can apply Lemma 4.3.5 to show the inductive step. b_{i+2} cannot be a block of any other type for the same reasons as in the base case.

□

With this theorem, we have shown that for a feasible TBD the partial order of signal updates is preserved by the function network translation. Furthermore, each function node corresponding to a block of this partial order is executed before the next time step starts i.e., in less than *bp* time units. This could only be shown because the function network translation does not add any additional delays to the sum of block execution times on a path. Thus, any feasible TBD leads to a valid function network translation in terms of partial order and timing constraints.

4.4. Summary and Related Work

We first gave a formal definition of Simulink block diagrams based on timed synchronous block diagrams. Due to the background of this work, the focus lays on the timing behavior of Simulink models in terms of execution orders of blocks. Thus, we abstract from functional details in terms of concrete values of signals and restrict ourselves to updates of signals and the time instances where they occur. With the help of a transition system, the execution behavior of a Simulink model in a specific simulation step was defined based on the partial order of block executions.

As a next step, the translation of a Simulink model into a function network was defined, where blocks are translated to function nodes and signals to channels connecting them. The initial input events are produced by a single event source to assure the synchronous behavior of the original model. The period of the event source is chosen as the base period *bp* of the model. The most challenging part of the translation was to represent the behavior of rate transition blocks between different synchronous sets correctly in function networks. To achieve this, a special function node type named period multiplier was defined, which is able to transform an event pattern with a period P into an event pattern with a period $k \cdot P$ and an initial offset. This kind of

function node was not only used to model rate transitions but also to create all needed sample times in the model.

The correctness of the translation was proven by relating signal updates to events in the function network and showing that the partial order of signal updates is preserved by the partial order of events in the function network translation. Additionally, we had to assure that each signal update of a simulation step occurs before the next simulation step starts. This leads to an end-to-end deadline of bp time units for each maximal chain of partially ordered blocks executed in a simulation step. As a next step, we defined a partial order of function network events for a simulation step t. To capture also the timing constraints, an event belongs to the partial order only if it occurs within bp time units after simulation step t starts. Then, we proved for each maximal chain of partially ordered signals that the respective events in the function network translation are also partially ordered. Assuming a feasible Simulink block diagram, this also assures a correct timing behavior with respect to the implicitly defined deadlines.

An Extension to Translate Stateflow

We will give a short outlook to a possible extension of this work that would allow to model Stateflow charts by transition systems of function networks. Even if this is not necessary to represent the timing behavior of a Simulink model correctly, it may improve the results of timing analyses. As an example, the worst case execution time of a Stateflow block may depend on its state. These dependencies may be modeled in terms of a transition system of a function node, which reduces over-approximations. Furthermore, this would be a first step to support dynamic triggers where an output signal of one block determines at runtime whether the target block of this signal is executed or not.

Due to a missing 'official' semantics of Stateflow, we had to rely on scientific approaches to define semantics such as [35] and [72]. In the latter one, a safe subset for Stateflow is defined that precludes unbounded behavior by avoiding loops in any graph of junctions and transitions and permits the use of the backtracking mechanism of flow transitions. A translation would be restricted to Stateflow models that satisfy this safe subset and the TargetLink modeling guidelines [54].

Stateflow blocks could be translated to transitions systems of function nodes by building the flattened parallel composition of all its Statecharts. Single Stateflow transitions could be translated to function network transitions quite straight-forward if they do not involve junctions and its conditions and actions are only defined on input and output signals. The main challenge for such a translation would be the abstract interpretation of conditions that trigger Stateflow transitions. These conditions had to be represented by events in function networks while the presence of an event denotes that the respective condition is true. Furthermore, we had to do this abstract interpretation for any block and signal of the whole Simulink model. This is because the conditional events needed for a Stateflow block have to be produced somewhere i.e., the nodes that write the corresponding signal have to produce the correct events.

The behavior of a Simulink block can in general be derived from the generated code. Thus, we had to perform a code analysis to derive an abstract semantics in terms of a transition system based on events that are sufficient to cover all paths of the Stateflow blocks. There already exist methods that tackle related problems as in the area of model-checking [20, 19], timing validation [86] and data flow analysis [74]. A further interesting approach has been proposed in [8], where an abstract domain is efficiently obtained from c-code generated from models similar to synchronous block diagrams.

Related Work

The question which semantic properties need to be preserved when translating synchronous languages like Simulink has been posed by many publications. A very general approach has been pursued in [81], where a translation from Simulink to Lustre is defined, which are both synchronous languages. To preserve semantics, the authors claim that the Simulink model and the translated Lustre program should have an identical output behavior when given the same inputs. Because the output behavior in Simulink is guaranteed to be the same if the partial order of block executions is maintained, this is a refinement of our notion of semantics preservation.

A work with a quite similar approach to ours is [55], where also causality and partial order of Simulink blocks is considered to define semantic equivalence. Furthermore, the relative execution rates between Simulink blocks and the sequence of read and write accesses on delay blocks should be maintained. This is also covered by our partial order on function network events by including the timing constraints into the partial order definition.

A further work with similar objectives to our approach was presented in [80], where a synchronous model is implemented on a loosely time-triggered architecture. The authors assume, as we do, that single processes compute their values correctly and do not model the functional behavior explicitly. This results in a synchronous language definition that is quite similar to synchronous block diagrams of Simulink. For desynchronization, they use an intermediate model with similarities to Kahn process networks and translate their synchronous model into that formalism. They define semantic preservation in terms of a partial order of process executions as we do. They prove this by showing that any execution of the intermediate model is also a valid execution of the synchronous model because it respects the partial order. In the rest of the paper, they focus on the details of how the intermediate model is implemented on the target architecture and how throughput and memory size may be influenced.

Compared to our work, they did some simplifications. First, they assume, as we do, that all feedback loops are split by a unit delay block. The main simplification is that they only consider single-rate models while for our approach the semantic preservation for multi-rate models is one major part. Furthermore, they cannot always guarantee that no data is lost. On the one hand, this can happen in a feedback loop with a unit-delay because the data is read before it was written by the process of the previous step. This violates simulation semantics but not the partial order, which is split at unit delay blocks. This is the reason why we explicitly claim that the timing constraints assumed

by Simulink are satisfied. On the other hand, there may occur similar problems with data consistency due to clock drifts on the architecture.

In [71], tasks are identified manually from a Simulink model and scheduled in a fixed-priority preemptive scheduling. Tasks are assumed to be independent from each other and the partial order of Simulink blocks is represented by task priorities that are determined by task deadlines. Here, the overlapping execution of Simulink simulation steps is considered, which leads to several problems. First, if a task is executed that depends on values of a lower-priority task (which can only occur in case of an algebraic loop) there must be inserted a unit delay block. Thus, algebraic loops are split, as we assume for our work as well. A second problem arises if a higher priority block A interrupts the execution of a lower-priority block B that depends on the outputs of block A. Then, these outputs are changed during the interrupt and thus invalidated. When block B resumes, it uses the wrong data and thus creates wrong results. To solve this issue, the authors introduce a communication scheme that uses different buffers to avoid that old values are invalidated if they are still needed. This problem does however only occur because overlapping executions are allowed and buffers were initially assumed to be one-place buffers. In our translation, we exclude overlapping executions by defining respective end-to-end deadlines. Furthermore, function networks contain FIFO-buffers in terms of synchronization buffers by definition. This assures that data is read in the correct order and is not invalidated during execution.

In [7], an overview is given on the basic idea of synchrony and the most important synchronous languages such as Esterel, Lustre and Signal, and how they did evolve over time. Furthermore, problems and challenges are discussed as, for example, the question of how to map a synchronous software model to an asynchronous or only partially synchronous hardware platform. This may either be done by modeling the hardware in a synchronous language as well, or by defining constraints the hardware architecture model must satisfy to assure synchrony. It is shown how to translate a Lustre model into the asynchronous representation of Ptolemy while maintaining the functional behavior. However, this is not always possible, which is shown at the example of Signal. Here, the synchronous model has to be extended in a way that its behavior is changed. In our work, we abstract from any concrete target hardware architecture and explicitly model synchrony by the translation to function networks. This is mainly realized by defining a single event source and period multiplier nodes to trigger the source blocks of each synchronous set. Thus, the task model preserves synchrony independently from the hardware architecture it will be allocated to.

In [6], it is discussed how to translate a synchronous language to an asynchronous one in general. One focus lays on a property of many synchronous languages where decisions of a program may be taken by the absence of an event or signal. This kind of modeling is however not possible in Simulink models, where a signal always has a value and no decision can be taken by the absence of a value. Another part of the work deals with the parallel composition of processes and the question how to ensure that all processes compute the same signal values in the same order as if they were executed in isolation.

A material difference of our work to [6] and [7] is the fact that we do not aim at translating the whole functionality of a synchronous language. This means, that we are

not considering the concrete values a process or block computes for showing semantic preservation. Instead, we represent all updates of signal values by a corresponding event in the task network and show that their partial order is preserved. Correctness of the functional behavior of single blocks is assured by code-generators for Simulink.

5. Task Creation

In Chapter 4, we defined a translation of a specification model in Simulink to the formalism of function networks, which preserves the specification semantics in terms of partial order of signal updates and timing. The translation is performed based on the finest granularity of Simulink, which are atomic blocks connected by signals. Because each block is translated to a function node, also the resulting function network has this granularity. This typically leads to a function network with a large number of nodes having high variance in computational intensity. This is due to the fact that standard blocks from the Simulink library often represent very simple operations, as an addition or multiplication. Other blocks such as Stateflow blocks and user-defined S-Functions contain more complex operations, which need considerably more time to execute. To estimate the load a node potentially produces on an ECU, we introduce *node weights*, which correspond to the induced processor utilization. They are determined by the fraction of the worst case execution time and the period of the function node.

Furthermore, a Simulink translation typically leads to huge amount of communication in particular between nodes within a synchronous set. If we treat each node of such a function network as a single task, this would result in a large communication overhead because many lightweight tasks were spread over the distributed hardware resources in the pursuing design space exploration process. Accordingly, we want to obtain a more suitable task set by merging function nodes to build tasks.

The approach taken by Simulink *Embedded Coder* is to put all blocks with the same sample time into one task. These are not only blocks of the same synchronous set but also of independent sets sharing the same sample time, which may lead to tasks with very large weights. For simulation purposes this might be useful because all those blocks are executed in the same simulation step, execution times are neglected, and parallel execution is not regarded at all. But for the execution on a distributed system, this strategy precludes any of these blocks from being executed concurrently, which increases the risk of deadline violations. Nevertheless, nodes of the same synchronous set are still good candidates to be executed in the same task [23]. But due to the possibly high variety of the number and weights of nodes in synchronous sets, not all sets would necessarily result in useful tasks. For example, we do not want to allow arbitrary large task weights because those tasks may be either not executable on some ECUs, or they would reduce the number of possible schedules due to large blocking times. On the other hand, tasks should not be too lightweight, because the sum of task switching times would increase and waste a significant amount of ECU capacity leading to *thrashing*. This is an effect describing that the processor consumes more time for task switching than for task executions. From the perspective of the design space exploration, it is desirable to have tasks with balanced weights. This would reduce the

impact of computational density of tasks, and the decision where to allocate a task would be more driven by the actual optimization criteria.

Another important issue for task creation is the communication between tasks, which may get very expensive if tasks are mapped to different ECUs and a bus has to be used. A bus is not only comparably slow, but also often the bottleneck of such systems and can hardly be upgraded. Hence, another objective for task creation should be to minimize communication between tasks to relieve the bus. In summary, to find an appropriate task set, communication density between tasks should be minimized and the weights of tasks should be balanced to avoid thrashing and excessively heavy tasks.

To achieve all of this, we introduce a metric called *cohesion*, where nodes are attracted by a high communication density and repulsed by high node weights. To describe communication density, we define also *channel weights*. They are determined by the amount of data that is transfered over a channel, the period describing how often a message is sent, and the maximum bandwidth of all available buses. Here again, we assume an optimal allocation by considering the fastest bus with the greatest bandwidth to calculate channel weights.

While minimizing the cohesion metric is the optimization goal for task creation, the user has the opportunity to guide the process by setting parameters and adding constraints. First, there are weight factors to rate the two aspects that influence the cohesion function i.e., the balancing of node weights and the reduction of communication density between nodes. This enables the user to set the focus on one of these aspects, or even ignore one aspect by setting the respective factor to zero. In addition, constraints are defined for the cohesion function in terms of a *maximum task weight* and a *minimum number of tasks*. These constraints may be used to restrict the set of possible solutions and are guaranteed to be respected by the task creation process if they are not already violated by the initial function network. This is because nodes are not split and thus initial weights cannot be reduced and the initial number of nodes may only become smaller. A further parameter, which is determined implicitly, is the *desired task weight*. It is calculated from the minimum number of tasks and the sum of initial node weights. It defines the expectation value for the weight balancing part of the cohesion function.

Furthermore, we want to support a feature where the user has the possibility to define *partitioning constraints*, which allows to force or forbid that two nodes are in the same task partition. Constraints that forbid merging of nodes are called *prohibitive* and those forcing nodes to be merged are called *commanding* constraints. Prohibitive constraints are initially satisfied and can be easily respected during task creation by avoiding the respective merging operations. Commanding constraints, on the other hand, are constructive constraints and the start partition does not already satisfy them. To consider those constructive constraints, we introduce an intermediate step before starting task creation by merging all nodes that are forced to be in the same partition due to commanding constraints.

Applications for such constraints are manifold. A typical scenario would be a set of blocks that belong to a specific function in the specification model and the user wants them to be in the same task. Referring to Simulink, this might be nodes originating from blocks of the same subsystem. By this means, the user is able to specify the atomic

granularity of function nodes for task creation. Hence, this process is strongly user-guided and allows to express expert knowledge in terms of parameters and constraints. The constraints proposed in this work can be considered as an initial set of constraints that is expandable if necessary to express further system requirements.

Based on the cohesion metric and the set of constraints, we can now tackle the question of how the partitioning of function nodes into a set of tasks is performed. Here, we propose a two-step heuristic approach, where the start partitioning assigns each function node to an own partition. First, an *initial algorithm* iteratively merges whole partitions of function nodes until no improvement can be found anymore. As a second step, a combination of the Kernighan-Lin (KL) [43] and Fiduccia/Mattheyses (FM) [32] algorithms is used. It exchanges and moves nodes between partitions to further reduce cohesion. We investigated several algorithms and have chosen a modified combination of KL and FM because it offers the best trade-off between runtime and sub-optimality of results. More details on this can be found in Section 5.3.

The task creation algorithm finishes with a result that is a partitioning of function nodes into sets where all nodes of one set should be merged to a task. This merging is realized by defining appropriate operations for function networks that replace a part of the function network, which we denote as *component*, by another one with the same interface. The interface is defined by the ports that connect the nodes within the component with nodes outside. Beside the actual merging of two function nodes, we define an operation to eliminate self-activations, which arise when merging succeeding function nodes. This operation concatenates two succeeding transitions that are executed within a self-activation to a single transition. A third operation eliminates local data nodes that are data nodes exclusively connected to one function node. This may also be a consequence of merging operations.

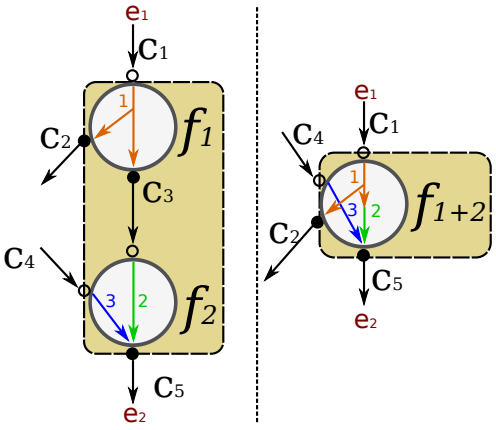


Figure 5.1.: Task Creation Example for a Task Chain

For all these operations we need to prove - as for the Simulink translation - that the semantics of the initial function network and thus the specification model is preserved. For a Simulink model, the partial order of signal updates occurring due to block

executions needs to be preserved. In the translated function network, each event corresponds to a signal update. This means for task creation that causality of all interface events must be preserved. Thus, if an input interface event e_1 of a component leads to an output interface event e_2, this causal dependency must still hold after each task creation operation. Concerning timing, we can only show that the delay between interface events is unaffected or becomes smaller. A complete timing analysis can first be done after the design space exploration phase, when tasks have been mapped to processors. Thus, we cannot validate the deadlines of Simulink in this process phase.

In Figure 5.1, we pick up the example from the introduction of Chapter 3, where a chain of two tasks is depicted that should be merged. In contrast to the previous consideration, we now use function networks for modeling instead of classic task networks. Thus, we have input and output ports and an internal transition system for nodes. Here, internal transitions are represented as arrows within a node. On the left, the starting situation is depicted where the rectangular box indicates the component that should be replaced by the merging operation. Node f_1 has one transition leading to events on channels c_2 and c_3 if an event e_1 has occurred on c_1. Node f_2 has two transitions, one for each input port, each leading to an event at its output port connected to channel c_5. On the right, the result after merging f_1 and f_2 and removing the self-activation is depicted. First, all input and output ports are maintained except the ports that connected f_1 and f_2. This connection is now represented as the concatenation of the respective transitions, which are here transitions 1 and 2. Accordingly, the channel that connected f_1 and f_2 is removed, which is here c_3. But still the causality of interface events is maintained, where we take here the events e_1 and e_2 as example. On the left, an event e_1 leads to an execution of transition 1 and thus an event on c_3. This activates f_2 with transition 2 leading to an event e_2 on channel c_5. On the right, an event e_1 triggers a concatenation of the transitions 1 and 2 leading to an output event e_2 on channel c_5 as well. The delay of the concatenated transition is the sum of the single transition delays of transitions 1 and 2. This kind of complex behavior resulting from node merging could not be modeled with classic task networks, especially when also internal states are considered.

Outline In Section 5.1, we introduce the optimization metric cohesion and define weights for function nodes and channels. In Section 5.2, we define formal composition operations that are used to realize merging of nodes in the function network formalism and prove that they preserve semantics in terms of causality of interface events. The algorithms that perform the partitioning of function nodes into task sets are presented in Section 5.3, including the assessment of alternative algorithms.

To evaluate the whole task creation methodology, we apply the approach in Section 5.4 to a case study of a driver assistance system model in Simulink. Additionally, we use benchmarks that imitate typical Simulink structures to also investigate the scalability of the approach for systems with a higher amount of nodes. In Section 5.5, we summarize this chapter and discuss related work for the task creation approach.

5.1. Cohesion and Weights

Task creation partitions the nodes of a function network, and merges all nodes of a single partition to get a set of tasks. We however do not want to partition only function nodes but also data nodes, because in the following design space exploration process also data nodes, such as shared variables and buffers, have to be allocated on ECUs. For deployment, we can think of a data node as a particular piece of code implementing the respective data object. Signal data nodes are typically not contained in a software specification model because they are used to model communication in a distributed system via a bus. Thus, also the function networks translated from a Simulink specification do not contain signals as defined in Chapter 4. Instead, signals will be added in the design space exploration phase to refine the communication delays due to mapping decisions. Thus, for the partitioning process, communication between nodes is represented solely by the channels connecting them. Please note, that for a function network that was derived from a Simulink model, all *period multiplier* nodes are excluded from the task creation process. This is because they are not part of the functional specification but only provide the correct event patterns to the different synchronous sets. Furthermore, they run at a different period than their successor nodes and thus must not be involved in a merging operation.

The optimization goal of task creation depends on utilization measures for computation and communication, which we refer to as node and channel weights. The weight $w(n)$ of a node n depends on its execution times in terms of transition delays and its event pattern. Execution times strongly depend on the compiler target. As for the Simulink translation, we define the delay of a transition as the minimum WCET among all potential processors of the target architecture. Thus, we assume that each node will be allocated to its best fitting processor. This allows to determine lower cost bounds in design space exploration as explained in Chapter 6.

More precisely, the weight of a node is defined as the sum of its port weights. The weight of a port is the maximum delay of all transitions starting at this port divided by the ports lower period bound. The period of a port can be retrieved by event pattern propagation for the class of function networks that we defined in Section 3.4. The same holds for function networks that were derived from a Simulink specification model as shown in Chapter 4. In the following, we define how weights of function nodes are determined. To also be able to obtain weights for data nodes, we consider their representation as function nodes as it was defined in the translation from extended to basic function networks in Section 3.2.

Definition 5.1.1 (Node Weight) *Let $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ be a function node. Its weight is defined as follows:*

$$w(f) = \sum\nolimits_{p_i \in \mathcal{P}^{in}} \left(\frac{1}{P_i^-} \cdot \max\nolimits_{t_{i,j}} (\delta^+(t_{i,j})) \right), \quad \textit{where}$$

$\delta^+(t_{i,j})$ is the upper delay bound of the jth transition starting from input port p_i and P_i^- is the lower period bound of p_i. ◇

Communication density is defined in terms of weights of channels depending on their data size, the communication rate, and the maximum bandwidth in *bytes/s* of all buses. The data size of a channel is derived from the specification model. For Simulink, the date size of each channel can be determined by considering the data type of the respective Simulink signal. Channel weights are defined as follows:

Definition 5.1.2 (Channel Weight) *Let $c = (p^{out}, \delta, p^{in})$ be a channel. The communication weight of c is defined as follows:*

$$com(c) = \frac{DataSize(c)}{\max Bandwidth} \cdot \frac{1}{P_c^-}, \quad where$$

$DataSize(c)$ is the data size of channel c and P_c^- is the lower period bound of c and its ports respectively. ◇

Formally, task creation partitions the set of function and data nodes denoted as $\mathcal{N} = \mathcal{F} \cup \mathcal{D}$ into a task set $\mathcal{T} = \{\tau_1, ..., \tau_m\}$ where $\tau_i = \{n_{i,1}, ..., n_{i,k}\}$, $n_{i,j} \in \mathcal{N}$. The communication structure of the resulting task set is determined by the set of channels $\mathcal{C}(\mathcal{T})$ between different partitions. The task set shall be chosen such that communication density is minimized and node weights are balanced. Node balancing is achieved by minimizing the standard deviation with respect to the desired task weight leading to preferably merging nodes with low weights. Communication is minimized by reducing the weight of the set of channels between partitions $\mathcal{C}(\mathcal{T})$. For the definition of cohesion, we introduce weight factors $\alpha, \beta \geq 0$ that are adjusted by user preference to control the process. Furthermore, we define m^- to be the minimum allowed number of tasks, which also determines the desired task weight w^*. This leads to the following definition of *cohesion*.

Definition 5.1.3 (Cohesion) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a set of nodes $\mathcal{N} = \mathcal{F} \cup \mathcal{D}$ and $\mathcal{T} = \{\tau_1, ..., \tau_m\}$ a set of tasks where $\tau_i = \{n_{i,1}, ..., n_{i,k}\}$, $n_{i,j} \in \mathcal{N}$. The cohesion function is defined as follows:*

$$\begin{aligned}
cohesion(fn, \mathcal{T}) &= \alpha \cdot \widehat{w}(\mathcal{T}) + \beta \cdot com(\mathcal{C}(\mathcal{T})), \quad where \\
\widehat{w}(\mathcal{T}) &= 1/m \cdot \sqrt{\textstyle\sum_{i=1}^{m} (w^* - w(\tau_i))^2} \quad (standard\ deviation) \\
w^* &= 1/m^- \cdot \textstyle\sum_{n \in \mathcal{N}} w(n) \quad (desired\ task\ weight) \\
w(\tau_i) &= \textstyle\sum_{n_{i,j} \in \tau_i} w(n_{i,j}) \quad (weight\ of\ task\ \tau_i) \\
com(\mathcal{C}(\mathcal{T})) &= \textstyle\sum_{c \in \mathcal{C}(\mathcal{T})} com(c) \quad (sum\ of\ communication\ weights)
\end{aligned}$$

◇

The partitioning process is intended to allow the user to guide and control the process to respect and satisfy his needs. Thus, beside the optimization goal of minimizing the cohesion function, we define a set of user-controlled constraints restricting the task creation process. First, additionally to the minimum allowed number of tasks m^-, we introduce a maximum achievable task weight w^+, which describes the maximum

utilization a single task should involve on a processor. The intention is to avoid creating tasks with too heavy weights, which might be hard to deploy on already utilized processors. The respective opposites, which are a maximum number of tasks and a minimum task weight, cannot be usefully considered because they might be incompatible with w^+ and m^- possibly leading to an empty set of solutions. Furthermore, the process strives for balancing task weights anyway leading to an increase of the minimum task weight and decrease of the number of tasks. Thus, the process needs to be restricted to not creating too heavy and too few tasks and not the opposite. In practice, the choice of all parameters α, β, m^- and w^+ will highly depend on the respective application and the expert knowledge of the user.

As a second means to influence the task creation process, we define further constraints named *partitioning constraints*, where we distinguish between *prohibitive* and *commanding* constraints. Prohibitive constraints forbid the merging of two nodes n_1 and n_2 written as $proh(n_1, n_2)$. Commanding constraints demand the merging of two nodes written as $command(n_1, n_2)$. Such constraints may be either defined manually by the user or derived from the specification model. As an example, one may obtain commanding constraints from the hierarchical structure of a Simulink model by claiming that all blocks of a specific subsystem should be merged to one task.

5.2. Formal Composition Operations and Semantics Preservation

When performing task creation, it is not sufficient to partition function nodes into sets that are meant to represent each a task because the semantics of the model remains the same. What task creation actually means is that a set of function nodes should be executed as one task. Thus, function node transitions within a single partition must not be executed concurrently because a task is mapped to one computation resource. Please note, that we assume single core processors for this work.

To represent this behavior also in the formal model, we merge all nodes of a task partition into one function node. Due to semantics of function nodes, this implies that there is only one transition of a merged node active at the same time. Hence, task creation can be regarded as a design decision in terms of a refinement step that reduces the concurrent execution of certain function node transitions. Thus, it changes semantics of the function network model. Nevertheless, we have to preserve certain semantic properties to respect the intended specification semantics. This has been shown for the translation of a Simulink specification model into a function network by considering the partial order of events and Simulink signal updates. While maintaining the partial order is sufficient to represent the semantics of the Simulink model, this is not the case for function networks in general. As already pointed out in Section 3.3, we exploit a more expressive formalism to capture function network semantics, which are causality patterns. Thus, we claim for task creation that semantics needs to be preserved in terms of causality from input to output events of a component interface. This might induce that intermediate events may not be observable anymore. But the

partial order of all remaining signal updates is preserved and thus the same input values still lead to the same outputs. Timing constraints of Simulink are considered for task creation in terms of end-to-end deadlines as they were defined in Chapter 4. Due to a missing allocation of tasks to processors, we cannot verify deadlines in task creation but only after design space exploration.

We define three operations for task creation starting with the actual merging of two function nodes into one. This operation is mandatory for task creation because it ensures that transitions are not executed concurrently anymore. The next operations are optional in the sense that they are not needed to perform a valid task creation. First, we define the elimination of local data nodes. This means that a data node that is exclusively connected to a single function node may be removed under specific conditions. A local data node is not shared by two or more processes and thus can be considered as local memory. Hence, the time delay to read the data is already included in the execution time of the function node. Furthermore, this operation reduces the complexity of the model by removing a node. Second, we define an operation to eliminate self-activations, which are self-loops from a function node output port to one of its input ports activating the node again. This is a typical result when merging two function nodes that were connected by an activation channel or a signal data node. Under specific conditions, this self-loop may be removed by concatenating the transitions that are executed sequentially during this loop to one transition. This leads to less task activations and thus also to less task switching because the task is not activated two times one after another. The delay of a concatenated transition is determined as the sum of the single transitions meaning that also the respective code segments are sequentialized. Potentially, this also leads to positive cache effects because the data for the second transition may be still cached. But due to the fact that we use approximate execution times in this process step anyway, the sum of delays is still a sufficient approximation.

The operations are defined with the help of a component concept where a component is a part of a function network with a well-defined interface of ports to the remaining network. Each operation replaces one component by another one with the same interface. For semantic correctness of an operation, we claim that the causality of interface events has to be maintained. This means that each causal dependency from input interface events to output interface events that is valid for the original component has to be also valid after the operation. Additionally, we need to ensure that the result is still a valid function network after Def. 3.2.1. In particular, this means that the transition system must be still deterministic and complete. A further property we would like to preserve is state-independence. As discussed in Section 3.4, this is the main property of the class of function networks where boundedness is decidable. If we can show that this property is preserved by task creation, we know that boundedness remains decidable as well. The other characteristics of the class of function networks are periodic event sources, which are not affected by task creation at all.

We will now define the different formal composition operations to perform task creation starting with the merging of nodes.

5.2.1. Merging nodes

When two function nodes are merged, this involves a restructuring of the function network by replacing a component of two function nodes f_1 and f_2 by a component with one function node f_{1+2} with the same interface. To realize this, each the sets of input ports and output ports of f_1 and f_2 are unified. The transition system T_{1+2} of f_{1+2} is obtained by building the interleaving composition $\|$ of the transition systems T_1 and T_2 of f_1 and f_2, respectively. We denote the states of the resulting transition system as the combination of the respective states of the original transition systems i.e. a state $s_1 s_2$ of T_{1+2} results from the combination of the states s_1 from T_1 and s_2 from T_2. Due to the fact that two transition systems of two function nodes always have different alphabets in terms of different input and output port events, the interleaving composition can be simply defined as follows:

Definition 5.2.1 (Interleaving Composition of Transition Systems) *Let* $\mathcal{A}_1 = (S_1, s_{01}, T_1)$ *and* $\mathcal{A}_2 = (S_2, s_{02}, T_2)$ *be two transition systems. Their interleaving composition is defined as follows:*

$$\mathcal{A}_1 \parallel \mathcal{A}_2 = (S_{1+2}, s_{01}s_{02}, T_1' \cup T_2') \text{ where}$$

- $S_{1+2} = \{s_1 s_2 \mid s_1 \in S_1,\ s_2 \in S_2\}$
- $T_1' = \{(p^{in}, E, s_1 s_2 \to \Psi, s_1' s_2) \mid (p^{in}, E, s_1 \to \Psi, s_1') \in T_1, s_2 \in S_2\}$
- $T_2' = \{(p^{in}, E, s_1 s_2 \to \Psi, s_1 s_2') \mid (p^{in}, E, s_2 \to \Psi, s_2') \in T_2, s_1 \in S_1\}$

◇

The node merging operation uses the interleaving composition to build the transition system of the new function node. The set of interface events, for which causality needs to be preserved, is the union of all input and output port events of both merged function nodes. Node merging is defined as follows:

Definition 5.2.2 (Node Merging) *Let* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be a function network and* $f_1 = (\mathcal{P}_1^{in}, \mathcal{A}_1, \mathcal{P}_1^{out}) \in \mathcal{F}$ *and* $f_2 = (\mathcal{P}_2^{in}, \mathcal{A}_2, \mathcal{P}_2^{out}) \in \mathcal{F}$ *be two function nodes. The merge operation is defined as follows:*

$$\begin{aligned} merge(fn, f_1, f_2) \ = \ & (\Sigma, \mathcal{P}, (\mathcal{F} \setminus \{f_1, f_2\}) \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C}), \\ & \text{where } f_{1+2} = (\mathcal{P}_1^{in} \cup \mathcal{P}_2^{in}, \mathcal{A}_1 \parallel \mathcal{A}_2, \mathcal{P}_1^{out} \cup \mathcal{P}_2^{out}) \end{aligned}$$

The set of interface events is defined as $\Sigma_{merge} = \bigcup_{p \in \mathcal{P}_1^{in} \cup \mathcal{P}_2^{in} \cup \mathcal{P}_1^{out} \cup \mathcal{P}_2^{out}} \Sigma(p)$.

◇

Please note, that the merging operation is associative because both the joining of ports and the interleaving composition of transition systems is associative. This becomes important for the application of this operation in the task creation algorithm.

When merging two function nodes, we create a new function network and thus have to assure that it is still valid concerning the properties we claimed in the function

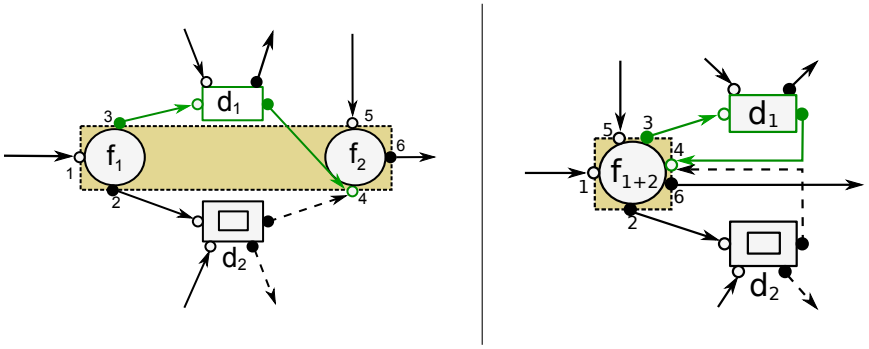


Figure 5.2.: Merging Function Nodes

network definition. The merge operation replaces one function node by another one by combining their transition system by interleaving composition. Thus, we need to show that this transition function is still a valid function. This means that there must exist exactly one transition for each combination of input events and internal state, which is shown in the following theorem.

Theorem 5.2.1 (Node Merging - Valid Transition System) *Let* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be a function network with two function nodes* $f_1 = (\mathcal{P}_1^{in}, \mathcal{A}_1, \mathcal{P}_1^{out}) \in \mathcal{F}$ *with* $\mathcal{A}_1 = (S_1, s_{01}, T_1)$ *and* $f_2 = (\mathcal{P}_2^{in}, \mathcal{A}_2, \mathcal{P}_2^{out}) \in \mathcal{F}$ *with* $\mathcal{A}_2 = (S_2, s_{02}, T_2)$*. Let further be* $fn' = merge(fn, f_1, f_2) = (\Sigma, \mathcal{P}, (\mathcal{F} \setminus \{f_1, f_2\}) \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C})$ *be the function network after merging where* $f_{1+2} = (\mathcal{P}_{1+2}^{in}, \mathcal{A}_{1+2}, \mathcal{P}_{1+2}^{out}) \in \mathcal{F}'$ *with* $\mathcal{A}_{1+2} = (S_{1+2}, s_{01+2}, T_{1+2})$*. Then it holds that* T_{1+2} *is a function i.e., it is deterministic and complete.*

Proof: For each transition $t \in T_1$*, a transition in* T_{1+2} *is created for each state* $s_2 \in S_2$*. We know that in* T_1 *a transition exists for each state and each combination of input events at each input port. By duplicating each transition for each state* s_2*, we also have a transition for each state in* T_{1+2} *. The same holds for transitions* $t \in T_2$*.* □

The semantic consequences of merging two function nodes f_1 and f_2 is that f_1 and f_2 are now executed on the same scheduling resource i.e., transitions of f_1 and f_2 cannot be executed concurrently anymore. But even though we change function network behavior by this operation, causality is still preserved for the interface events. This is because all events, ports, channels and data nodes are maintained as well as the transition systems of the original function nodes. Concerning timing, node merging may enlarge the delay between the arrival of an event at an input port and the emitted output event, because transitions that could be executed concurrently before cannot be executed concurrently after merging. Thus, the wait delay in the activation buffer may enlarge. Because computational weights of function nodes are the sum of their port weights, and all ports are maintained including their transitions, the weight of f_{1+2} is the sum of the single weights of f_1 and f_2 as claimed in the weight calculation.

In Figure 5.2 on the left, a component of a function network with two function nodes f_1 and f_2 is depicted where f_1 triggers f_2 via a signal data node d_1, and two activation

channels. Furthermore, there are read and activation channels to a shared data node d_2. The set of interface events is the set of all events that belong to ports on the edge of the component. The same function network part after merging f_1 and f_2 is depicted on the right of Figure 5.2. The activation path is now a self-activation i.e., f_{1+2} activates itself at a different input port via the signal data node d_1. The shared data node d_2 remains unaffected and the read channel moves with its target port to the new created function node f_{1+2}.

In the next theorem, we prove that the node merging operation preserves causality from its input to its output interface events. Causal dependencies for function nodes are determined in terms of transitions of the internal transition system. Thus, we consider each transition of f_1 and f_2 and show that the causal dependency it induces also holds for f_{1+2}. However, we have to regard that the set of states has changed by building the parallel product of both state sets. Thus, if a causal dependency starts in a state $s_1 \in S_1$ of f_1, this causal dependency has to hold for all states $s_1 s_2$ of f_{1+2} where $s_2 \in S_2$ is any state from f_2.

Theorem 5.2.2 (Node Merging - Preserving Causality) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with two function nodes $f_1 = (\mathcal{P}_1^{in}, \mathcal{A}_1, \mathcal{P}_1^{out}) \in \mathcal{F}$ with $\mathcal{A}_1 = (S_1, s_{01}, T_1)$ and $f_2 = (\mathcal{P}_2^{in}, \mathcal{A}_2, \mathcal{P}_2^{out}) \in \mathcal{F}$ with $\mathcal{A}_2 = (S_2, s_{02}, T_2)$. Let further $fn' = merge(fn, f_1, f_2) = (\Sigma, \mathcal{P}, (\mathcal{F} \setminus \{f_1, f_2\}) \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C})$ be the function network after merging where $f_{1+2} = (\mathcal{P}_{1+2}^{in}, \mathcal{A}_{1+2}, \mathcal{P}_{1+2}^{out}) \in \mathcal{F}'$ with $\mathcal{A}_{1+2} = (S_{1+2}, s_{01+2}, T_{1+2})$.*

1. *If there exists a causal dependency between input and output events of f_1 in fn due to a transition $t_1 = (p^{in}, \{i_1, ..., i_n\}, s_1 \rightarrow \{...., (p^{out}, o, \delta), ...\}, s_1') \in T_1$ (Theorem 3.3.4), then this causal dependency also exists in f_{1+2} in fn' with respect to the state set of f_{1+2} and its start event $start_{f_{1+2}}$ i.e.*

$$
\begin{aligned}
&(1)\ (p^{in}.(i_1, ..., i_n), start_{f_1})[state_{f_1} = s_1] \xrightarrow{\delta} p^{out}.o[state_{f_1} = s_1'] \\
&\qquad \text{with } i_1, ..., i_n, o \in \Sigma,\ s_1, s_1' \in S_1 \\
\Longrightarrow\ &(2)\ \forall s_2 \in S_2 : \\
&\qquad (p^{in}.(i_1, ..., i_n), start_{f_{1+2}})[state_{f_{1+2}} = s_1 s_2] \xrightarrow{\delta} p^{out}.o[state_{f_{1+2}} = s_1' s_2] \\
&\qquad \text{with } i_1, ..., i_n, o \in \Sigma', s_1 s_2, s_1' s_2 \in S_{1+2}
\end{aligned}
$$

Proof: From (1) we know that there exists a transition $t_1 = (p^{in}, (i_1, ..., i_n), s_1 \rightarrow \{...., (p^{out}, o, \delta), ...\}, s_1') \in T_1$. Following Def. 5.2.2, in the merging operation the transition system of f_{1+2} is determined by the interleaving composition as defined in Def. 5.2.1. Thus, for each $s_2 \in S_2$ there exists a transition $t_1' = (p^{in}, (i_1, ..., i_n), s_1 s_2 \rightarrow \{...., (p^{out}, o, \delta), ...\}, s_1' s_2) \in T_{1+2}$. With Theorem 3.3.4 follows immediately that (2) has to hold.

□

2. *If there exists a causal dependency between input and output events of f_2 in fn, then this causal dependency also exists in fn' i.e.*

$$\begin{aligned}&(p^{in}.(i_1,...,i_n), start_{f_2})[state_{f_2} = s_2] \xrightarrow{\delta} p^{out}.o[state_{f_2} = s_2'] \\ &\quad with\ i_1,...,i_n, o \in \Sigma,\ s_2, s_2' \in S_2 \\ \Longrightarrow\ &\forall s_1 \in S_1 : \\ &(p^{in}.(i_1,...,i_n), start_{f_{1+2}})[state_{f_{1+2}} = s_1 s_2] \xrightarrow{\delta} p^{out}.o[state_{f_{1+2}} = s_1 s_2'] \\ &\quad with\ i_1,...,i_n, o \in \Sigma', s_1 s_2, s_1 s_2' \in S_{1+2}\end{aligned}$$

Proof: This proof works in the same way as the first one.

□

To preserve the decidability of boundedness for periodic state-independent function networks, it is necessary that the function network is still in this class after two nodes have been merged. The property of state-independence is defined for output ports. Thus, if we know that each output port stays state-independent also the function node resulting from the merging is state-independent. We show in the next theorem that state-independence is preserved by the operation of merging nodes.

Theorem 5.2.3 (Node Merging - State-Independence) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with two function nodes $f_1 = (\mathcal{P}_1^{in}, \mathcal{A}_1, \mathcal{P}_1^{out}) \in \mathcal{F}$ with $\mathcal{A}_1 = (S_1, s_{01}, T_1)$ and $f_2 = (\mathcal{P}_2^{in}, \mathcal{A}_2, \mathcal{P}_2^{out}) \in \mathcal{F}$ with $\mathcal{A}_2 = (S_2, s_{02}, T_2)$. Let further $fn' = merge(fn, f_1, f_2) = (\Sigma, \mathcal{P}, (\mathcal{F} \setminus \{f_1, f_2\}) \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C})$ be the function network after the merging operation where $f_{1+2} = (\mathcal{P}_{1+2}^{in}, \mathcal{A}_{1+2}, \mathcal{P}_{1+2}^{out}) \in \mathcal{F}'$ with $\mathcal{A}_{1+2} = (S_{1+2}, s_{01+2}, T_{1+2})$. Let $p^{out} \in \mathcal{P}_1^{out} \cup \mathcal{P}_1^{out}$ be a state-independent output port. Then it also holds that $p^{out} \in \mathcal{P}_{1+2}^{out}$ is state-independent.*

Proof: We show the proof for f_1 i.e., $p^{out} \in \mathcal{P}_1^{out}$. The proof for f_2 works similar. From the definition of state-independence (see Def. 3.2.7) we know that if there exists a transition $t = (p_i^{in}, E, s \rightarrow \{...(p^{out}, b, \delta)...\}, s') \in T_1$ from an input port p_i^{in} with $b \in \Sigma(p^{out})$, then there exists such a transition for each set of events $E' \in \Sigma^{act}(p_i^{in})$ i.e.

$$\forall s_j \in S_1,\ E' \in \Sigma^{act}(p_i^{in}) :\ \exists t' = (p_i^{in}, E', s_j \rightarrow \{...(p^{out}, b', \delta_j)...\}, s_j') \in T_1 \quad with\ b' \in \Sigma(p^{out})$$

When merging f_1 and f_2 after Def. 5.2.2 to a function node $f_{1+2} = (\mathcal{P}_{1+2}^{in}, \mathcal{A}_{1+2}, \mathcal{P}_{1+2}^{out}) \in \{f_1, f_2\}$ with $\mathcal{A}_{1+2} = (S_{1+2}, s_{01+2}, T_{1+2})$, the transition t is represented as a set of transitions: For each state $s_2 \in S_2$ there exists a transition $(p_i^{in}, E, s\, s_2 \rightarrow \{...(p^{out}, b, \delta)...\}, s'\, s_2) \in T_{1+2}$. Now it remains to proof that there exists such a transition also for each $E' \in \Sigma^{act}(p_i^{in})$ and each state of f_{1+2}. This follows again from Def. 5.2.2, because also for each transition $t = (p_i^{in}, E', s_j \rightarrow \{...(p^{out}, b', \delta_j)...\}, s_j') \in T_1$ a transition for each state $s_2 \in S_2$ is created as follows:

$$(p_i^{in}, E', s_j s_2 \rightarrow \{...(p^{out}, b', \delta_j)...\}, s_j' s_2) \in T_1.$$

□

We have shown for the merge operation that it preserves causality on its interface events, creates a valid transition system and preserves the property of state-independence. Furthermore, it should be noted that even if the transition delays remain the same, the time an activation event spends in the activation buffer may be enlarged. This is due to the fact that f_{1+2} has more input ports than each single node f_1 and f_2. Thus, the needed capacity of the activation buffer may become larger and thus also the maximum waiting delay. How a sufficient buffer capacity may be determined has been discussed in Section 3.3.

5.2.2. Elimination of Local Data Nodes

A data node d is local if it is exclusively connected to a function node f and in the same task partition as f. When eliminating a data node, also the corresponding read and activation channels are removed. The transition system of f is modified such that all events that are used to access the date node are removed from each transition. To be able to ensure that semantics is preserved correctly by this operation, the function node execution semantics must not depend on any event that is read from d. Thus, if there would exist two transitions that define different output behavior only depending on the occurrence of two different read events r and r', we cannot remove this data node without losing causality. Additionally, the transition system would not be deterministic anymore because there would be two transitions for the same combination of input events and state. The set of interface events is defined as the set of all input and output port events, except the read and write events of the removed data node. This leads to the following definition.

Definition 5.2.3 (Data Node Elimination) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network, $f = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out}) \in \mathcal{F}$ a function node and $d \in \mathcal{D}$ a data node with $\mathcal{P}^{in}(d) = \{p_d\}$, $\mathcal{P}^{out}(d) = \{p'_d\}$, an incoming activation channel $c_w = (p_w, \delta_w, p_d) \in \mathcal{C}^A$ with $p_w \in \mathcal{P}^{out}$ transmitting an event set $W = \Sigma(p_w)$, and an outgoing read channel $c_r = (p'_d, \delta_r, p_r) \in \mathcal{C}^R$ with $p_r \in \mathcal{P}^{in}$ transmitting an event set $R = \Sigma(p_r)$.*

- *Assumption A1: There exist no transitions that describe different behavior only depending on a read event $r \in R$ i.e.*

$$\nexists (p, \{i_1, ..., i_n, r\}, s \to \Psi_1, s'_1),\ (p, \{i_1, ..., i_n, r'\}, s \to \Psi_2, s'_2) \in T$$

with $r, r' \in R$, $r \neq r'$ and $\Psi_1 \neq \Psi_2$ or $s'_1 \neq s'_2$.

If fn satisfies assumption A1, then data node elimination is defined as:

$$elim_d(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}'), \text{ where}$$

- $\Sigma' = \Sigma \setminus R$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_{d'}\}$,
- $\mathcal{D}' = \mathcal{D} \setminus \{d\}$, $\mathcal{C}' = \mathcal{C} \setminus \{c_r, c_w\}$ *and*

- $f = (\mathcal{P}^{in}, (S, s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})$, T' *contains all transitions from* T *where each event* $r \in R$ *is deleted in* E *if it contains* r *i.e.*

$$
\begin{aligned}
T' = \{\ & (p^{in}, E^*, s \to \Psi, s') \mid \\
& \exists t = (p^{in}, E, s \to \Psi, s') \in T, \\
& E^* = \begin{cases} (i_1, ..., i_n) & , if\ E = (i_1, ..., i_n, r) \mid r \in R \\ E & , else \end{cases} \\
\}&
\end{aligned}
$$

The set of interface events is defined as $\Sigma_{elim_d} = \bigcup_{p \in \mathcal{P}^{in} \cup \mathcal{P}^{out}} \Sigma(p) \setminus (R \cup W)$. ◇

To ensure that the function network resulting from this operation is valid, we have to show that the transition system is still deterministic and complete. When eliminating a data node, each transition that depends on a read event is modified such that this read event is removed. The assumption that there exists no other transition that is triggered by the same input events but a different read event, ensures that the transition system is still valid as proven in the following theorem.

Theorem 5.2.4 (Data Node Elimination - Valid Transition System) *Let* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be a function network with a function node* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ *with* $\mathcal{A} = (S, s_0, T)$ *and a data node* $d \in \mathcal{D}$ *as defined in Def. 5.2.3 and* $fn' = elim_d(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ *where* $f = (\mathcal{P}^{in}, (S, s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})) \in \mathcal{F}$ *be the function network after* d *has been eliminated. Then it holds that* T' *is a function i.e., it is deterministic and complete.*

Proof: Each transition that is triggered by an event set $\{i_1, ..., i_n, r\}$ *with* $r \in R$ *is contained also in* T' *while the read event* r *is removed leading to* $\{i_1, ..., i_n\}$*. Due to Assumption A1, we know that there is no other transition that is triggered by an event set* $\{i_1, ..., i_n, r'\}$ *containing the same events except a different read event* $r' \in R$ *with* $r' \neq r$*. This is the only case that would lead to non-determinism. Thus, we have still exactly one transition for all combinations of states and event sets. The remaining transitions stay unchanged. Thus,* T' *is deterministic and complete.*

□

With the assumption that the behavior of the function node does not depend on the read event of the removed data node, the data node elimination operation maintains the causality of all interface events. All input ports of the function node are obtained together with all activation events of that node. The transitions of the function node are maintained as well while the read event r is removed. Thus, the causality between input and output events of the components interface is still valid. Concerning timing, the delay between any input and output signal that involves the reading of event r becomes smaller because the data is now available locally and the time for reading the event is saved. Thus, any end-to-end deadline that was valid before this operation is still valid afterwards. In Figure 5.3 on the left, a component is shown with a function node f and a local data node d that is eliminated on the right. The arrows in the

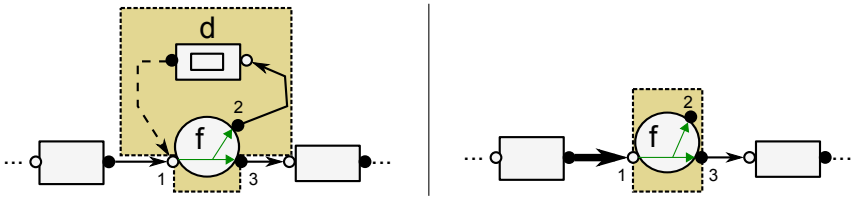


Figure 5.3.: Elimination of Local Data Nodes

function node indicate the affected transitions to show that these are maintained even if the local data node is removed. The output port where the eliminated data node was connected to remains but is not connected to any channel now. Thus, the event is still observable but is not received by any other node.

The next step is to formally show that the data node elimination function preserves the causality of all interface events as we have claimed before. First, we show this for all transitions that involve a read event $r \in R$. Thus, all causalities must be preserved for all events that are part of the interface, which excludes all read events $r \in R$ and write events $w \in W$.

Theorem 5.2.5 (Data Node Elimination - Read Causality) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ and a data node $d \in \mathcal{D}$ as defined in Def. 5.2.3. Let further $fn' = elim_d(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after d has been eliminated where $f = (\mathcal{P}^{in}, (S', s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})) \in \mathcal{F}$.*

If there exists a causal dependency between input and output events of f in fn involving a read event $r \in R$ caused by a transition

$$t = (p^{in}, \{i_1, ..., i_n, r\}, s \to \{..., (p^{out}, o, \delta), ...\}, s') \in T,$$

then this causal dependency also exists in fn' while omitting the read and write events from R and W i.e.

$$\begin{aligned} & (p^{in}.(i_1, ..., i_n, r), start_{f_1})[state = s] \xrightarrow{\delta} p^{out}.o[state = s'] \\ & \quad with\ i_1, ..., i_n, r, o \in \Sigma,\ s, s' \in S \\ \Longrightarrow\ & (p^{in}.(i_1, ..., i_n), start_{f_1})[state = s] \xrightarrow{\delta} p^{out}.o[state = s'] \\ & \quad with\ i_1, ..., i_n, o \in \Sigma',\ s, s' \in S \end{aligned}$$

Proof: Following Def. 5.2.3, the transition t is contained as $t' = (p^{in}, \{i_1, ..., i_n\}, s \to \{..., (p^{out}, o, \delta), ...\}, s')$ in T' because $r \in R$ is a read event. With Theorem 3.3.4 this immediately leads to the statement to prove.

□

Second, we prove for all transitions that do not contain the read event that causality is preserved as well.

Theorem 5.2.6 (Data Node Elimination - Non-Read Causality) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ and a data node $d \in \mathcal{D}$ as defined in Def. 5.2.3. Let further $fn' = elim_d(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the data node d has been eliminated where $f = (\mathcal{P}^{in}, (S', s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})) \in \mathcal{F}$.*

If there exists a causal dependency between input and output events of f in fn without involving a read event $r \in R$ caused by a transition

$$t = (p^{in}, \{i_1, ..., i_n\}, s \to \{..., (p^{out}, o, \delta), ...\}, s') \in T,$$

then this causal dependency also exists in fn' i.e.

$$\begin{aligned} & (p^{in}.(i_1, ..., i_n), start_{f_1})[state = s] \xrightarrow{\delta} p^{out}.o[state = s'] \\ & \quad with\ i_1, ..., i_n, o \in \Sigma,\ i_1, ..., i_n \notin R,\ s \in S \\ \Longrightarrow\ & (p^{in}.(i_1, ..., i_n), start_{f_1})[state = s] \xrightarrow{\delta]} p^{out}.o[state = s'] \\ & \quad with\ i_1, ..., i_n, o \in \Sigma',\ s \in S' \end{aligned}$$

Proof: Following Def. 5.2.3, t is also contained in T' because $i_1, ..., i_n \notin R$ are no read events. With Theorem 3.3.4 this immediately leads to the statement to prove.

□

We have now shown that causality is preserved for transitions with and without read events. What is interesting beyond the question of causality, is how the data node elimination influences the timing delays between the affected events. In particular, we claimed before that this operation reduces the delay until the function node is activated. Referring to the translation of read channels into basic function networks, there is a specific function node for each input port with read channels that requests the data from the involved data nodes. The delay induced by a reading process is saved and instead the function node may be activated immediately if there are no other read dependencies and no other execution is active. In particular, we are interested in the delay from the time where all events needed for an activation are available to the time where the function node execution starts. Due to Def. 3.2.3, we denote events which activate the function node as $a_1, ..., a_m$ and events that are read from data nodes as $r'_1, ..., r'_n$. In Figure 5.4, the translation of read channels and data nodes is shown on the left, and the induced delays are annotated on the right.

The first delay, we consider, is the delay of the activation channel $c_{a_j} = (p^*_j, \delta_{a_j}, p)$ leading to input port p of function node f. According to Def. 3.2.3, an activation channel is translated to a function node if its delay δ_{a_j} is greater than zero. Thus, in addition to δ_{a_j} there may also occur a wait delay at this function node, which we denote as $wait_{start_{a_j}}$. Accordingly, the wait delay for a function node resulting from a read channel $c_{r_i} = (p^{out}_{d_i}, \delta_{r_i}, p)$ from a data node d_i is denoted as $wait_{start_{r_i}}$, and the wait delay of d_i itself as $wait_{d_i}$. The wait delay of the function node f_p that represents the input port p is denoted as $wait_{start_{f_p}}$. The transition delay of f_p was defined in Def. 3.2.3 to be $[\epsilon, \epsilon]$.

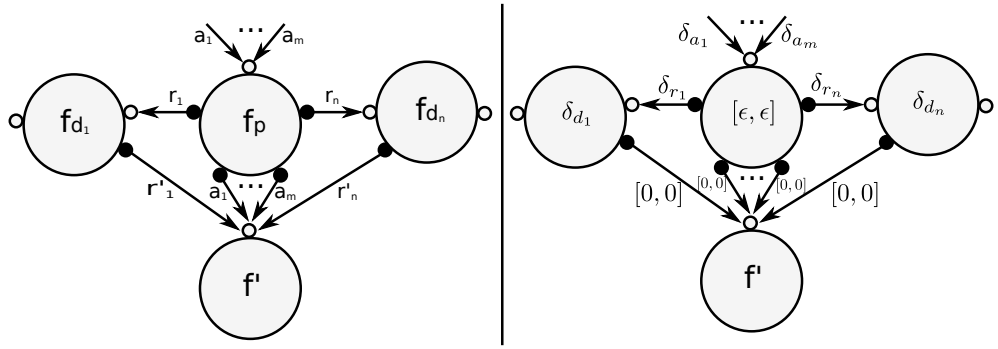


Figure 5.4.: Left: Translation of Read Channels into Basic Function Network
Right: Delays of Reading Data Nodes

Theorem 5.2.7 (Delay for Reading from Data Nodes) *Let p be an input port of a function node f with a non-empty set of incoming read channels $\mathcal{C}_p^R = \{c_{r_1}, ..., c_{r_n}\}$ where each $c_{r_i} = (p_{d_i}^{out}, \delta_{r_i}, p) \in \mathcal{C}_p^R$ with $i \in \{1, ..., n\}$ reads data from a data node $d_i \in \mathcal{D}_{shared} \cup \mathcal{D}_{fifo}$ and a set of activation channels $\mathcal{C}_p^A = \{c_{a_1}, ..., c_{a_m}\}$ with $c_{a_j} = (p_j^*, \delta_{a_j}, p) \in \mathcal{C}_p^A$ and $j \in \{1, ..., m\}$. Then the following causal dependency holds:*

$$\{p_1^*.a_1, ..., p_1^*.a_m\} \xrightarrow{delay} \{p.a_1, ..., p.a_m, p.r_1', ..., p.r_n'\}$$
$$\text{where } a_j \in \Sigma(p_j^*), r_i' \in \Sigma(p_{d_i}^{out}),$$
$$delay = \max_j(wait_{start_{a_j}} + \delta_{a_j}) + wait_{start_{f_p}} + [\epsilon, \epsilon]$$
$$+ \max_i(wait_{start_{r_i}} + \delta_{r_i} + wait_{start_{d_i}} + \delta_{d_i})$$

Proof:

1. *From Def. 3.2.3 case 3.(b), Theorem 3.3.3 and Corollary 3.3.6 it follows*

$$\{p_1^*.a_1, ..., p_1^*.a_m\} \xrightarrow{\max_j(wait_{start_{a_j}} + \delta_{a_j})} p'.(a_1, ..., a_m)$$

2. *From Def. 3.2.3 case 3.(c) and Corollary 3.3.1 it follows that it holds:*

$$p'.(a_1, ..., a_m) \xrightarrow{wait_{start_{f_p}} + [\epsilon, \epsilon]} \{p_{r_1}^{out}.r_1, ..., p_{r_n}^{out}.r_n, p_{a_1}^{out}.a_1, ..., p_{a_m}^{out}.a_m\}$$

3. *For each read channel c_{r_i} leading to a data node d_i it holds:*
   a) *For read channel c_{r_i} we know from Def. 3.2.3 case 3.(d)i.:*

$$\forall i : p_{r_i}^{out}.r_i \xrightarrow{wait_{start_{r_i}} + \delta_{r_i}} p_{d_i}^{r}.r_i$$

b) *For data node d_i we know from Corollary 3.3.3 (Shared) and Corollary 3.3.4 (FIFO) that:*

$$p^r_{d_i}.r_i \xrightarrow{wait_{start_{d_i}}+\delta_{d_i}} p^{out}_{d_i}.r'_i,\ r'_i \in \Sigma(p^{in}_{d_i})$$

c) *For the outgoing channel from data node d_i we from Def. 3.2.3 case 3.(d)ii.:*

$$p^{out}_{d_i}.r'_i \xrightarrow{[0,0]} p.r'_i$$

4. *From Def. 3.2.3 case 3.(e) and Theorem 3.3.1 it follows:*

$$\forall j \in \{1,...,m\} : p^{out}_{a_j}.a_j \xrightarrow{[0,0]} p.a_j$$

5. *1. to 4. lead together by transitivity to*

$$\{p^*_1.a_1,...,p^*_1.a_m\} \xrightarrow{delay} \{p.a_1,...,p.a_m,p.r'_1,...,p.r'_n\}$$
$$\text{where } delay = \max_j(wait_{start_{a_j}} + \delta_{a_j}) + wait_{start_{f_p}} + [\epsilon,\epsilon]$$
$$+ \max_i(wait_{start_{r_i}} + \delta_{r_i} + wait_{start_{d_i}} + \delta_{d_i})$$

□

If a data node d_i is removed but there are still other data nodes, this only reduces the overall delay if all other read delays are smaller. This would reduce the term $\max_i(wait_{start_{r_i}} + \delta_{r_i} + wait_{start_{d_i}} + \delta_{d_i})$. If the removed data node was the only node where f reads from, there is no extra function node f_p needed for port p and the delay reduces to the delay of the activation channels leading to

$$\{p^*_1.a_1,...,p^*_1.a_m\} \xrightarrow{delay'} \{p.a_1,...,p.a_m\}$$
$$\text{where } a_j \in \Sigma(p^*_j),\ delay' = \max_j(wait_{start_{a_j}} + \delta_{a_j}).$$

Thus, the execution and waiting delay for the date node is saved as well as the delay of the read channel and the delays of the additional function node f_p.

As the last proof for the elimination of local data nodes, we show that the property of state-independence is preserved and thus boundedness remains decidable.

Theorem 5.2.8 (Data Node Elimination - State-Independence) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ and a data node $d \in \mathcal{D}$ as defined in Def. 5.2.3. Let further $fn' = elim_d(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the data node d has been eliminated where $f = (\mathcal{P}^{in}, (S, s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})) \in \mathcal{F}$. If $p^{out} \in \mathcal{P}^{out}$ is state-independent before the operation, it is still state-independent afterwards.*

Proof: Each transition $t \in T$ that leads to p^{out} is also contained in T' while the event tuple may be changed if it contains a read event $r \in R$. This has no influence on the property of state-independence because no transition has been deleted and the output specifications are maintained.

□

We have shown that the elimination of local data nodes preserves causality under the assumption that read events do not influence the causality induced by the transition system. This is always the case for a function network originating from a Simulink model. In the translation defined in Section 4.2, data nodes are defined for outgoing signals of Moore-sequential blocks closing non-algebraic loops, and for data store memory blocks. In the first case, there is only one event that may be read from or written into a local data node. Thus, the transition system behavior cannot depend on different read events, because we abstract from the concrete signal value here. For a data store memory block b this also holds because we assume that there is only one data store write block b_w writing to b. Hence, the elimination of local data nodes may always be applied for those data nodes.

5.2.3. Elimination of Self-Activations

Self-activations are self-loops of a function node f either via a signal data node or a direct activation channel. They particularly arise when two function nodes with an activation dependency are merged. Thus, their elimination is a typical continuation of the node merging operation. As a consequence of eliminating self-activations the involved channels are removed including the respective ports and events. If a signal is part of the self-loop it is removed if it is not accessed by other function nodes.

To be able to apply this operation without violating causality of events, the input port of the self-activation loop must not have any other incoming channels from other nodes. Otherwise, we could not remove the input port due to the synchronization with other channels. This also holds for read channels. Thus, the elimination of local data nodes should be applied before removing self-activations. A further necessary condition to eliminate a self-activation containing a data node d is that d must not have *both* incoming and outgoing channels to other function nodes than f. In this case, it would not be possible to remove the self-activation without affecting activations from or to other nodes.

Before defining the operation for eliminating self-activations, we define some help functions. The first one adds an output delay to a given set of output specifications of a transition. An output specification is a tuple of a port, an event and a delay interval.

Definition 5.2.4 (Output Delay Addition) *Let $\psi = \{(p'_1, e'_1, \delta_1), ..., (p'_n, e'_n, \delta_n)\}$ with $\delta_i = [\delta_i^-, \delta_i^+]$ be a set of output specifications, and $\delta = [\delta^-, \delta^+]$ a delay interval. Output delay addition is defined as:*

$$\begin{aligned}\delta_{add}(\Psi, \delta) \; &= \{(p'_1, e'_1, \delta_1 + \delta), ..., (p'_n, e'_n, \delta_n + \delta)\}, \\ &\quad \textit{where } \delta_i + \delta = [\delta_i^- + \delta^-, \delta_i^+ + \delta^+]\end{aligned}$$

◇

Next, we define how a transition system changes when a self-activation via an output port p_w and an input port p_a is eliminated. For each transition that does not contain one of these two ports nothing changes. But all pairs of transitions that would execute

successively in the case of a self-activation need to be concatenated. This means, that the left part of the first transition, consisting of the input port, input event, and origin state, becomes also the left part for the concatenated transition. The right part is determined by the target state of the second transition, and the unified set of output specifications, where the delay of output specifications containing output port p_w is added to each output specification of the second transition. All other output specifications remain unchanged.

Definition 5.2.5 (Self-Transition Concatenation) *Let T be a transition system, p_a be an input port and p_w an output port of a self-activation. Let further $keep_{p_a}$ and $keep_{p_w}$ be boolean flags that indicate whether p_a or p_w are maintained or not. The Self-Transition Concatenation operation is defined as:*

$$concat(T, p_a, p_w, keep_{p_a}, keep_{p_w}) = T', \text{ where}$$

1. $\forall t = (p, E, s \to \Psi, s') \in T \mid (p \neq p_a \ \vee \ keep_{p_a}) \ \wedge \ \nexists \psi = (p_w, w, \delta) \in \Psi : \ t \in T'$
2. *For each pair of transitions*
   - $t_1 = (p_1, E_1, s_1 s_2 \to \Psi_1, s_1' s_2) \in T$ *where* $\exists \psi = (p_w, w, \delta_1) \in \Psi_1$, *and*
   - $t_2 = (p_a, E_2, s_1' s_2 \to \Psi_2, s_1' s_2') \in T$ *:*

$$\exists t_{1+2} \in T' \mid t_{1+2} = \begin{cases} (p_1, E_1, s_1 s_2 \to \Psi_1 \cup \delta_{add}(\Psi_2, \delta_1), s_1' s_2') & , if \ keep_{p_w} \\ (p_1, E_1, s_1 s_2 \to \Psi_1 \setminus \psi_w \cup \delta_{add}(\Psi_2, \delta_1), s_1' s_2') & , else \end{cases}$$

$\diamond$

Elimination of self-activations is defined for a function node f if it activates itself either via a signal data node d or a direct channel. Self-activation is resolved by replacing it by a set of concatenated transitions. This means that succeeding executions of the self-activation are merged into one using the previously defined functions. There are two assumptions that need to be satisfied to remove self-activations. First, the involved input port must not have any other incoming channels. Second, the state of the erstwhile function f_2 must not be changed between two concatenated transitions by another transition starting at another input port. This is satisfied, for example, if the original function nodes f_1 and f_2 each have only one input port. In this case, the respective state can only be changed by this port, which also must be the port of the self-activation. If we consider the Simulink translation from Section 4.2, the second assumption always holds because each function node originating from an ordinary block has one input port where all input channels are synchronized. The operation to eliminate self-activations is defined as follows:

Definition 5.2.6 (Self-Activation Elimination) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network after the merging operation of two nodes f_1 and f_2 to f with $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$. Let there further be a self loop leading from an output port $p_w \in \mathcal{P}^{out}$ sending the event set W to an input port $p_a \in \mathcal{P}^{in}$ receiving the event set A. If f fulfills the following assumptions*

- *A1: p_a has exactly one incoming channel namely the channel of the self loop*
- *A2: There exists no transition of f that starts at another port than p_a and changes the state of the (erstwhile) function node f_2 i.e.*

$$\nexists t = (p, E, s_1 s_2 \to \Psi, s_1 s_2') \in T \text{ with } p \neq p_a,\ s_2' \neq s_2$$

then the self-activation elimination is defined as follows:

$$elim_a(fn, f, p_w, p_a) = (\Sigma', \mathcal{P}', \mathcal{F}', \Phi, \mathcal{D}', \mathcal{C}')$$

where we distinguish the following cases:

1. *If the self loop only consists of a direct channel $c = (p_w, \delta_c, p_a)$ then*
   - $\Sigma' = \Sigma \setminus (W \cup A)$, $\mathcal{P}' = \mathcal{P} \setminus \{p_a, p_w\}$, $\mathcal{D}' = \mathcal{D}$, $\mathcal{C}' = \mathcal{C} \setminus \{c\}$,
   - $\mathcal{F}' = \mathcal{F} \setminus \{f\} \cup \{f'\}$ *with*

   $$f' = (\mathcal{P}^{in} \setminus \{p_a\}, (S, s_0, concat(T, p_a, p_w, false, false)), \mathcal{P}^{out} \setminus \{p_w\})$$

   The set of interface events is defined as $\Sigma_{elim_a} = \bigcup_{p \in \mathcal{P}^{in} \cup \mathcal{P}^{out}} \Sigma(p) \setminus (W \cup A)$.

2. *If the self loop involves a signal data node $d \in \mathcal{D}_{signal}$ that has an incoming activation channel $c_w = (p_w, \delta_w, p_d) \in \mathcal{C}^A$ with $p_d \in \mathcal{P}^{in}(d)$ transmitting an event set W and an outgoing activation channel $c_a = (p_{d'}, \delta_a, p_a) \in \mathcal{C}^A$ to f transmitting an event set A, we get the following cases:*

   a) If d has no other channels than c_w and c_a, then
   - $\Sigma' = \Sigma \setminus (W \cup A)$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_{d'}, p_a, p_w\}$, $\mathcal{D}' = \mathcal{D} \setminus d$, $\mathcal{C}' = \mathcal{C} \setminus \{c_w, c_a\}$,
   - $\mathcal{F}' = \mathcal{F} \setminus \{f\} \cup \{f'\}$ *with*

   $$f' = (\mathcal{P}^{in} \setminus \{p_a\}, (S, s_0, concat(T, p_a, p_w, false, false)), \mathcal{P}^{out} \setminus \{p_w\})$$

   The set of interface events is defined as $\Sigma_{elim_a} = \bigcup_{p \in \mathcal{P}^{in} \cup \mathcal{P}^{out}} \Sigma(p) \setminus (W \cup A)$.

   b) If d has an additional activation channel to another or the same function node, then
   - $\Sigma' = \Sigma \setminus A$, $\mathcal{P}' = \mathcal{P} \setminus \{p_{d'}, p_a\}$, $\mathcal{D}' = \mathcal{D}$, $\mathcal{C}' = \mathcal{C} \setminus c_a$,
   - $\mathcal{F}' = \mathcal{F} \setminus \{f\} \cup \{f'\}$ *with*

   $$f' = (\mathcal{P}^{in} \setminus \{p_a\}, (S, s_0, concat(T, p_a, p_w, false, true)), \mathcal{P}^{out})$$

   - $d = (\mathcal{P}_d^{in}, \delta, \mathcal{P}_d^{out} \setminus \{p_{d'}\})$

   The set of interface events is defined as $\Sigma_{elim_a} = \bigcup_{p \in \mathcal{P}^{in} \cup \mathcal{P}^{out}} \Sigma(p) \setminus A$.

c) If d has an additional activation channel from another or the same function node, then

- $\Sigma' = \Sigma \setminus W$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_w\}$, $\mathcal{D}' = \mathcal{D}$, $\mathcal{C}' = \mathcal{C} \setminus \{c_w\}$,
- $\mathcal{F}' = \mathcal{F} \setminus \{f\} \cup \{f'\}$ *with*

$$f' = (\mathcal{P}^{in}, (S, s_0, concat(T, p_a, p_w, true, false)), \mathcal{P}^{out} \setminus \{p_w\})$$

- $d = (\mathcal{P}_d^{in} \setminus \{p_d\}, \delta, \mathcal{P}_d^{out})$

The set of interface events is defined as $\Sigma_{elim_a} = \bigcup_{p \in \mathcal{P}^{in} \cup \mathcal{P}^{out}} \Sigma(p) \setminus W$. ◇

Also for this operation, we need to show that the transition system of the resulting function node is deterministic and complete to guarantee a valid function network. To show this, we consider the function of self-transition concatenation from Def. 5.2.5.

Theorem 5.2.9 (Self-Activation Elimination - Valid Transition System) *Let T be a transition system, p_a be an input port and p_w an output port of a self-activation and let $T' = concat(T, p_a, p_w, keep_{p_a}, keep_{p_w})$ be transition system after applying the self-transition concatenation function. Then it holds that T' is a function i.e., it is deterministic and complete.*

Proof: For case 1 of Def. 5.2.5 holds that if $keep_{p_a} = false$ (meaning that the input port p_a is removed), all transitions of T that start at a port $p \neq p_a$ and do not produce any events at p_w are also contained in T'. If $keep_{p_a} = true$, then also transitions that start at p_a are maintained as long as also the second condition holds. Thus, all transitions are maintained correctly except those that write to p_w. These transitions are covered by case 2 of Def. 5.2.5. Here, we consider each transition t_1 that leads to p_w and each transition t_2 that starts at p_a. Independently from $keep_{p_w}$, a transition t_{1+2} is created with the same input events and source state as t_1, which maintains completeness and determinism. □

The semantic consequences of eliminating self-activations is the change of causal event chains that include events $w \in W$ and $a \in A$. All these event chains are shortened by removing a sub-chain from w to a. This is realized by concatenating the appendant transitions. But even if these events are removed, the causality of the interface events of the component is still preserved.

This is exemplified in Figure 5.5, where a function node with a self-activation by a direct activation channel is shown, which corresponds to case 1 of Def. 5.2.6. The arrows in the function node indicate two transitions t_1 and t_2 that are executed successively. On the right, the situation is shown after the ports 2 and 3 were removed by eliminating the self-activation. Here, t_1 and t_2 are concatenated to one transition denoted as t_{1+2}. But an activation at port 1 still leads to an event at port 4 as on the left side. What is different, is the fact that both transitions are now executed as one transition. While on the left, it was possible that another activation occurs between these transitions, this is not possible on the right anymore. In Figure 5.6, the same situation is depicted for a self-activation involving a signal data node without any further channels from or to other function nodes.

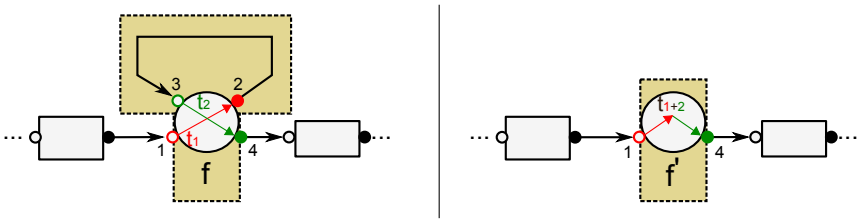


Figure 5.5.: Simple Self-Activation with Direct Channel

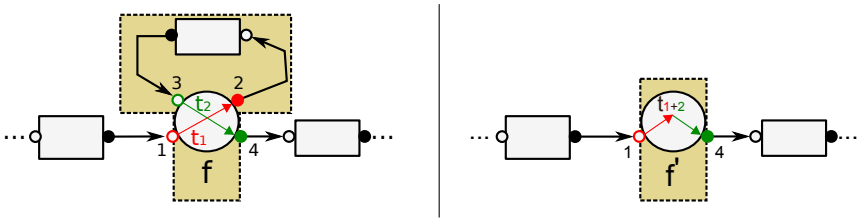


Figure 5.6.: Simple Self-Activation with Local Data Node

Figure 5.7 shows another example, where the involved data node has a further outgoing activation channel to another function node f_2. Thus, the data node is still existent after self-loop elimination but the channel leading back to f is removed. Additionally, the output port 2 still exists to activate f_2. So, even if t_1 and t_2 are concatenated to one transition t_{1+2}, the firing of port 2 is maintained. This keeps the causality of the interface events leading to f_2. Concerning timing, the delay between any input and output event of the interface either stays the same (if it is not affected by the self-activation) or is even shortened because the delay of the self-activation is no longer existent. Furthermore, the number of task switches is reduced because two activations are now executed as one. In Figure 5.8, a similar situation is shown where the data node has a further incoming channel from f_2. In this case, the incoming channel to port 3 has to be maintained while the channel starting at port 2 is removed.

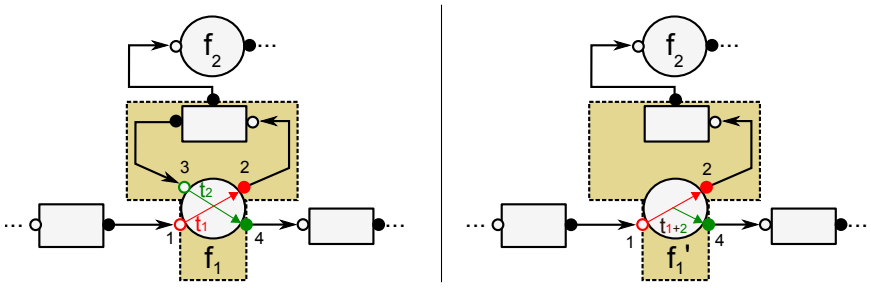


Figure 5.7.: Self-Activation with Data Node with Additional Outgoing Channel

To formally show that elimination of self-activations preserves causality of interface events, we first state which causal dependencies and delays hold for a self-activation involving a signal data node connected to an output port p_w and an input port p_a of

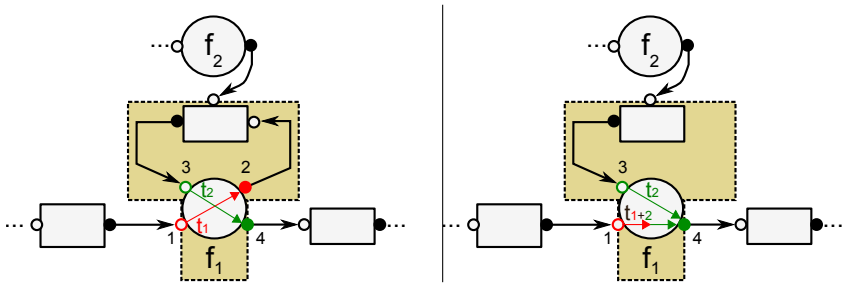


Figure 5.8.: Self-Activation with Data Node with Additional Incoming Channel

a function node f. We introduce the following notations for the different delays that occur during a self-activation: δ_{write} denotes the delay for the channel leading from the function node to the signal, which writes the data by producing an event. δ_{signal} denotes the delay of the execution of the signal. δ_{act} denotes the delay for the channel leading from the signal back to the function node to activate it again. These delays are also annotated in Figure 5.9.

Lemma 5.2.1 (Causality of Signals and Channels) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further be $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the self loop has been eliminated.*
Then it holds:

$$\begin{aligned} &\exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \wedge \\ &\exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \wedge \\ &\exists c_a = (p_{d'}, \delta_a, p_a),\ p_{d'} \in \mathcal{P}_d^{out} \\ \implies &\forall e \in \Sigma(p_w) :\ p_w.e \xrightarrow{\delta_{write}+\delta_{signal}+\delta_{act}} p_a.e \\ \text{where } &\delta_{write} = wait_{start_w} + \delta_w, \\ &\delta_{signal} = wait_{start_d} + \delta_d, \\ &\delta_{act} = wait_{start_a} + \delta_a \end{aligned}$$

Proof: see Lemma C.1.1 in the appendix on page 231. □

Based on the previous lemma, we will show now which causal relations hold if there exists a transition t_1 producing an event at output port p_w leading into a self-loop to input port p_a. This triggers another transition t_2, which produces an output event at output port p_o. The delay that is induced by the loop depends on whether it contains a signal data node or not. In Lemma 5.2.1, we have shown the delays for a self-loop with a signal. For a self loop consisting of a direct channel, the delay can be derived from Corollary 3.3.6 of the function network chapter. To abstract from the details of the self loop, we denote its delay as δ_{loop}.

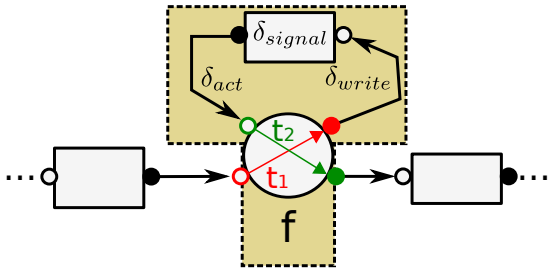


Figure 5.9.: Delays for Self-Activation

Additionally, there are delays induced by the involved transitions t_1 and t_2 and a delay to wait for the start event before the second transition can be executed, which is denoted as δ_{start_f}.

Lemma 5.2.2 (Causality of Self Loop) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further be $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ the function network after the self loop has been eliminated.*

Then the following holds:

$$
\begin{aligned}
&\exists t_1 = (p_1, in, s \to \Psi_1, s') \in T \mid \exists \psi_1 = (p_w, e, \delta_1) \in \Psi_1 \wedge \\
&\exists t_2 = (p_a, e, s' \to \Psi_2, s'') \in T \mid \exists \psi_2 = (p_o, o, \delta_2) \in \Psi_2 \wedge \\
&(\\
&\exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \wedge \\
&\exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \wedge \\
&\exists c_a = (p_{d'}, \delta_a, p_a),\ p_{d'} \in \mathcal{P}_d^{out} \textit{(signal self loop)} \\
&\vee \\
&\exists c_c = (p_w, \delta_c, p_a) \textit{ (channel self loop)} \\
&) \\
\implies &(p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1 + \delta_{loop} + \delta_{start_f} + \delta_2} p_o.o[state_f = s'']
\end{aligned}
$$

Proof: see Lemma C.1.2 in the appendix on page 232. □

Until now we have shown the causal dependencies that hold for the original function node. To show semantic preservation of the operation to eliminate self-activations, we will prove that this causal dependency still holds after this operation has been performed. We further show that the delay of this relation is reduced by the delay δ_{loop} induced by the loop and the delay δ_{start_f} to wait for the start event for the second transition. This is because both transitions are executed now as one.

Theorem 5.2.10 (Self Loop Elimination - Loop Causality) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further be $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ the function network after the self loop has been eliminated. Then it holds:*

$$\begin{aligned} &(1)\ (p_1.(i_1,...,i_n), start_f)[state_f = s] \xrightarrow{\delta_1+\delta_{loop}+\delta_{start_f}+\delta_2} p_o.o[state_f = s''] \\ &\quad \textit{with } i_1,...,i_n, o \in \Sigma,\ s, s'' \in S \\ \implies &(2)\ (p_1.(i_1,...,i_n), start_{f'})[state_{f'} = s] \xrightarrow{\delta_1+\delta_2} p_o.o[state_{f'} = s''] \\ &\quad \textit{with } i_1,...,i_n, o \in \Sigma' \end{aligned}$$

Proof: As stated in Lemma 5.2.2, there exist two transitions $t_1 = (p_1, in, s \to \Psi_1, s') \in T$ with $in = \{i_1,...,i_n\}$ and $\psi_1 = (p_w, e, \delta_1) \in \Psi_1$, and $t_2 = (p_a, e, s' \to \Psi_2, s'') \in T$ with $\psi_2 = (p_o, o, \delta_2) \in \Psi_2$ leading to (1). These two transitions are concatenated in fn' after Def. 5.2.5 to a transition

$$t_{1+2} = \begin{cases} (p_1, E_1, s_1 s_2 \to \Psi_1 \cup \delta_{add}(\Psi_2, \delta_1), s_1' s_2') & , if\ keep_{p_w} \\ (p_1, E_1, s_1 s_2 \to \Psi_1 \setminus \psi_w \cup \delta_{add}(\Psi_2, \delta_1), s_1' s_2') & , else \end{cases}$$

where $s = s_1 s_2, s' = s_1' s_2$ and $s'' = s_1' s_2'$. As defined in Def. 5.2.4 with $\delta_{add}(\Psi_2, \delta_1)$, the delay interval δ_1 is added to each output event specification $\psi_2 \in \Psi_2$ leading to an overall delay of $\delta_1 + \delta_2$. Then with Theorem 3.3.4 it follows

$$\begin{aligned} &(2)\ (p_1.(i_1,...,i_n), start_{f'})[state_{f'} = s] \xrightarrow{\delta_1+\delta_2} p_o.o[state_{f'} = s''] \\ &\quad \textit{with } i_1,...,i_n, o \in \Sigma'. \end{aligned}$$

□

It remains to show that this operation also does not violate any other causalities of the function node that were not part of the self-activation. To prove this, we need to consider all cases that were also part of the definition of this operation.

Theorem 5.2.11 (Self Loop Elimination - Non-Loop Causality) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the self loop has been eliminated. If there exists a transition*

$$t = (p, E, s \to \Psi, s') \in T \mid p \neq p_a \ \vee\ \nexists \psi = (p_w, e_w', \delta_w) \in \Psi$$

that is not part of the self-loop, then the causal dependency this transition involves (see Theorem 3.3.4) also exists in the function network after self loop elimination i.e.

$$\begin{aligned} &(1)\ (p.E, start_f)[state_f = s] \xrightarrow{\delta} p'.e'[state_f = s'] \\ &\qquad \textit{with } (p', e', \delta) \in \Psi,\ in, e' \in \Sigma,\ s \in S \\ \Longrightarrow\ &(2)\ (p.E, start_{f'})[state_{f'} = s] \xrightarrow{\delta} p'.e'[state_{f'} = s'] \\ &\qquad \textit{with } in, e' \in \Sigma',\ s \in S' \end{aligned}$$

Proof: We have to distinguish the same cases as in Def. 5.2.6.

1. *Case 1 of Def. 5.2.6 considers a self-loop that consists of a direct channel where it holds that $p \neq p_a \ \wedge\ \nexists \psi = (p_w, E_w, \delta_w) \in \Psi$. This leads to an application of Def. 5.2.5 where both $keep_a$ and $keep_w$ are set to false. Thus, with case 1 of Def. 5.2.5, it holds that t is also contained in the transition system of the derived function node T'. With Theorem 3.3.4, this immediately leads to the causal dependency (2).*

2. a) *Case 2(a) of Def. 5.2.6 considers a self loop that consists of a signal data node d with no other channels than c_w and c_a and it holds that $p \neq p_a \wedge \nexists \psi = (p_w, E_w \rightarrow \delta_w) \in \Psi$. This leads to an application of Def. 5.2.5 where both $keep_a$ and $keep_w$ are set to false. Thus, with case 1 of Def. 5.2.5 it holds that t is also contained in T'. With Theorem 3.3.4, this immediately leads to the causal dependency (2).*

   b) *Case 2(b) of Def. 5.2.6 considers a self loop that consists of a signal data node d with an additional activation channel to another or the same function node and it holds that $p \neq p_a$. This leads to an application of Def. 5.2.5 where $keep_a$ is set to false and $keep_w$ is set to true because p_w remains at function node f'. If $p' \neq p_w$ and thus $e' \notin W$, it follows with case 1 of Def. 5.2.5 that t is also contained in T', which leads with Theorem 3.3.4 to the causal dependency (2).*

      If otherwise $p' = p_w$ and thus $e' \in W$, then we know from case 2 of Def. 5.2.5 that in the concatenation of $t = t_1$ and t_2, the complete set of output specifications of t_1 is also contained in t_{1+2} because $keep_w = true$. Thus, also (p', e', δ) is contained in t_{1+2}, which leads with Theorem 3.3.4 to the causal dependency (2).

   c) *Case 2(c) of Def. 5.2.6 considers a self loop that consists of a signal data node d with an additional activation channel from another or the same function node and it holds $\nexists \psi = (p_w, E_w, \delta_w) \in \Psi$. This leads to an application of Def. 5.2.5 where $keep_a$ is set to true and $keep_w$ is set to false because p_a remains at function node f'. Thus, t is part of T' as defined in case 1 of Def. 5.2.5. With Theorem 3.3.4, this immediately leads to the causal dependency (2).* □

As a last step, we finally show that this operation also preserves the property of state-independence, which is important to keep the decidability of boundedness.

Theorem 5.2.12 (Self Loop Elimination - State-Independence) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the self loop has been eliminated. Then it holds that each output port p^{out} that was state-independent before is still state-independent after this operation.*

Proof: The proof immediately follows from Def. 5.2.5, where each transition that is contained in T' depends on the same combination of input events and state as a transition that was already contained in T. Because we know that the transition system is still complete and deterministic (see Lemma 5.2.9), it follows that state-independence is preserved for each output port that is not removed. □

We have now shown how task creation is performed formally on a function network by applying a set of operations. The merging of nodes can always be applied and thus we can merge any two function nodes. The result of merging is always a valid function network and it preserves the causality of all events because no port or channel is removed. After node merging, we may also remove local data nodes and eliminate self-activations if the needed conditions are satisfied. For these conditions, we have proven that the result is a valid function network and the causality of the interface events is preserved as well. If the function network was originally translated from a Simulink model, as defined in Chapter 4, this ensures that also the partial order on signal updates of this Simulink model is preserved.

We further showed for the second and third operation how the delays change in the causality pattern due to the removal of a data node or self-activation. Another important property that is preserved by all operations is state-independence, which assures that boundedness remains decidable after the task creation process.

5.3. Task Creation Algorithm

The objective of the task creation algorithm is to partition function and data nodes into a set of at least m^- partitions while minimizing the cohesion function and respecting the user-defined constraints. As defined in Section 5.1, m^- denotes the minimum number of allowed tasks. The function nodes of each individual partition are merged afterwards into a single task by the operations defined in Section 5.2. From the semantic point of view, each two function nodes may be merged without violating any causality of events. But at least for function networks derived from Simulink models, we restrict to exclusively merge nodes with the same sample time and thus the same period. This avoids that nodes of different synchronous sets are merged, which is important to assure that the end-to-end deadlines defined over chains of nodes within a synchronous set can be verified. Otherwise, an event that is referenced by an end-to-end deadline might not be observable anymore because the respective port where it occurs was removed by the operation to eliminate self-activations. This may induce that a deadline cannot be verified in the resulting model because the removed event will never occur.

One of our requirements for the algorithm is that it should offer a good trade-off between its runtime and the quality of its results with respect to the optimal solution. Furthermore, it should not depend on a complex configuration of parameters to deliver good solutions, because the user should not be obliged to get a deep understanding of the algorithm to be able to apply this approach.

We investigated a number of algorithms and compared them with respect to their runtime and their optimality. Some algorithms needed to be extended to be applicable to our problem. The first algorithm, we considered, is the Kernighan-Lin (KL) algorithm [43]. Assuming a given initial partitioning, it aims at finding the minimum cut set of a graph by minimizing the edge weights between two partitions of nodes. The algorithm has two nested loops that are repeated until no better solution can be found. In the inner loop, it iteratively exchanges pairs of nodes between partitions until a sequence of exchange operations leads to a better solution with respect to the minimum cut. It chooses the sequence of operations to be performed that offers the highest gain in terms of reducing edge weights between partitions. The exchanged nodes are marked and the process is repeated with the unmarked nodes until no better solution can be found or all nodes are marked. Then, the inner loop is finished and the currently best solution is saved. Based on this intermediate solution, the outer loop starts the process again with all nodes unmarked and ends if no inner loop execution leads to better results. The complexity of this algorithm is $O(n^2 \cdot log(n))$, where n is the number of nodes. Additionally, in [43] some improvements for the KL algorithm are suggested how to efficiently compute the gain of an exchange operation by only considering the weights of those edges that are affected by the exchange operation. Because it was not clear at the time of evaluation whether these improvements can be applied to our scenario, which has a different optimization goal, we first implemented a "pure" version of KL as a reference. Another implementation, we refer to as KL^+, makes use of these improvements and additionally combines the KL with the FM algorithm, which is presented next.

The Fiduccia/Mattheyses (FM) algorithm [32] is a modification of KL where nodes are not exchanged but moved from one partition to the other. The partition size may be bounded to avoid too small or even empty partitions. The complexity of this algorithm is $O(n)$. Quick Cut (QC) [25] is also a modification of KL with the goal to further improve the runtime. To this end, it maintains a neighborhood relation to reduce the number of nodes that are considered in the inner loop and more efficient data structures. The complexity is $O(m \cdot log(n))$, where m is the number of edges. Please note that the KL algorithm and all its extensions are usually restricted to two partitions. For the evaluation, we extended these approaches to be able to handle n partitions as it is needed for the problem we want to solve.

Another candidate is the Simulated Annealing (SA) algorithm [44]. It is a probabilistic approach based on the idea of annealing and cooling of solids that become more and more stable with less temperature. It allows with a certain probability also solutions that are worse than previous ones to escape from local minima. The probability is reduced with less temperature to let the algorithm converge to a solution. [44] also contains a review on simulated annealing and its use in practice.

#Tasks	#Edges	#Partitions	KL	KL$^+$	QC	FM	SA
Scale Tasks							
400	100	2	186	<1	<1	3	13
500	100	2	426	<1	<1	6	16
750	200	2	-	1	<1	50	50
1000	200	2	-	1	<1	93	69
5000	200	2	-	20	1	-	-
10000	200	2	-	81	1	-	-
50000	200	2	-	-	4	-	-
Scale Partitions							
100	100	40	17	<1	<1	22	7
100	100	50	16	<1	<1	32	7
1000	1000	4	-	10	2	-	-
1000	1000	50	-	132	19	-	-
Scale Edges							
100	1000	2	39	<1	<1	2	27
1000	1000	4	-	9	2	-	-
1000	10000	4	-	52	28	-	-

Table 5.1.: Comparing Algorithms w.r.t Runtime (in Seconds)

These algorithms have been applied to a set of benchmarks to compare them with respect to runtime and optimality. First, Table 5.1 shows the runtimes of the different algorithms in seconds. The first three columns show the number of tasks, edges and partitions. Entries marked with '-' were aborted after a timeout of ten minutes. It shows that both the KL^+ and the QC algorithm scale well also for larger systems where the others were terminated by the timeout. This shows that the improvements that were used in these two KL variants lead to a significant speed-up in runtime where the QC algorithm still performs better than the KL^+.

Table 5.2 compares the quality of the results for a number of task networks. As reference, we implemented a complete algorithm that delivers the optimal solution for each considered task network. The task network is characterized in the first column by its number of tasks ($\#T$), edges ($\#E$), and partitions ($\#P$). The second column shows the costs of the optimal solution for the respective system. Furthermore, for each algorithm the concrete costs and the degree of optimality with respect to the optimal solution is depicted in each two columns. A value of 100% is the best value meaning that the heuristic has found the optimal solution.

The table shows, that the variation of the results is relatively high for all algorithms. Interestingly, the quite simple KL and KL^+ algorithms perform only slightly worse than SA, although SA should be better in escaping from local minima. Overall, the results of the SA algorithm reached the best optimality with an average value of 88.1% (shown in the the bottom line). But due to the extensive runtime and the potential high sensitivity to the parameter configuration, this algorithm was not chosen for our approach. The KL^+ and KL algorithm lead to exactly the same qualitative results, which means that the FM algorithm does not improve the solutions for these benchmarks and that the runtime improvements of the KL^+ have no influence on the quality of the results. The results of the QC algorithm slightly differ from the KL^+ results and the average optimality is only 80% compared to 84,5%. This may be caused

#T/#E/#P	Opt. Costs	KL	Opt. %	KL$^+$	Opt. %	QC	Opt. %	SA	Opt. %
10/20/2	75	118	63.6	118	63.6	118	63.6	237	31.6
	180	180	100.0	180	100.0	180	100.0	267	67.4
15/40/2	272	273	99.6	273	99.6	353	77.1	346	78.6
	301	424	71.0	424	71.0	495	60.8	478	63.0
17/40/2	265	302	87.7	302	87.7	490	54.1	297	89.2
	192	331	58.0	331	58.0	331	58.0	376	51.1
	214	257	83.3	257	83.3	418	51.2	309	69.3
	199	209	95.2	209	95.2	209	95.2	330	60.3
20/50/2	271	314	86.3	314	86.3	372	72.8	367	73.8
	279	280	99.6	280	99.6	280	99.6	374	74.6
	245	311	78.8	311	78.8	567	43.2	245	100.0
	218	327	66.7	327	66.7	327	66.7	356	61.2
20/100/2	721	842	85.6	842	85.6	1082	66.6	911	79.1
	832	892	93.3	892	93.3	892	93.3	851	97.8
	803	829	96.9	829	96.9	829	96.9	836	96.1
	846	902	93.8	902	93.8	971	87.1	957	88.4
20/200/2	1751	1998	87.6	1998	87.6	2060	85.0	1914	91.5
	1579	1920	82.2	1920	82.2	1920	82.2	1901	83.1
	1711	1901	90.0	1901	90.0	1934	88.5	1891	90.5
	1853	2077	89.2	2077	89.2	1931	96.0	1884	98.4
12/40/3	416	534	77.9	534	77.9	622	66.9	521	79.8
	370	588	62.9	588	62.9	588	62.9	772	47.9
	537	609	88.2	609	88.2	624	86.1	594	90.4
	421	521	80.8	521	80.8	521	80.8	460	91.5
15/80/3	979	1050	93.2	1050	93.2	1192	82.1	1095	89.4
	967	1155	83.7	1155	83.7	1242	77.9	1054	91.7
	878	1042	84.3	1042	84.3	1008	87.1	908	96.7
	890	929	95.8	929	95.8	929	95.8	936	95.1
12/100/4	1500	1607	93.3	1607	93.3	1607	93.3	1647	91.1
	1625	1792	90.7	1792	90.7	1792	90.7	1654	98.2
12/10/4	85	85	100.0	85	100.0	85	100.0	85	100.0
12/40/4	473	617	76.7	617	76.7	617	76.7	473	100.0
			84.5		84.5		80.0		88.1

Table 5.2.: Comparing Algorithms w.r.t Optimality (in %)

by the fact that QC uses different data structures and a different order in searching for solutions and thus could not escape from local minima for some cases where KL and KL^+ found better solutions.

Overall, the KL^+ and the QC algorithm provide the best trade-off between optimality and performance and it remains the question if or in which degree the improvements leading to the speed-up in runtime can be applied to our approach. The answer for the QC algorithm is that its main improvement, which is the neighborhood relation only based on the communication between nodes, cannot be applied to our approach, because the cohesion metric does not only consider communication but also weight balancing. For the KL^+ algorithm the situation is better because the improvements on calculating the gain in communication can also be used for calculating the communication part of the cohesion function. For the weight balancing part we implemented a similar idea by only re-calculating the weights for those nodes that are involved in an

exchange or move operation. Thus, we decided to chose the KL^+ algorithm as it offers good performance and quality and its improvements are applicable to our approach.

This decision has mainly two consequences: First, we need to replace the optimization function of the KL^+ algorithm by the cohesion metric we defined previously. Second, the KL^+ algorithm always assumes an initial partitioning to start with. This is the reason why we define an *initial algorithm*, which produces such a start solution by merging only those partitions that contain nodes that communicate with each other. This is intended to retain the concurrency of the model by not merging nodes without any causal dependencies. The algorithm starts with a partitioning where each node is assigned to a single partition. Thus, the communication structure of the model is initially maintained. Then, the algorithm iteratively merges two partitions that communicate via at least one channel. Thus, with each merging step communication is reduced. The decision which pair of partitions is merged is determined by the best cohesion gain. The gain is defined as the difference of the cohesion value before and after a merging operation.

Definition 5.3.1 (Initial Algorithm for Task Creation) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a set of nodes $\mathcal{N} = \mathcal{F} \cup \mathcal{D} = \{n_1, ..., n_m\}$ and $\mathcal{T} = \{\tau_1, ..., \tau_m\}$ be a set of partitions and tasks with $\tau_i = \{n_i\}, i \in \{1, ..., m\}$. The initial algorithm is defined as follows:*

1. *Calculate current partitioning costs $co = cohesion(fn, \mathcal{T})$ and set $G_{max} := 0$.*
2. *For each channel c connecting two nodes $n \in \tau_i$ and $n' \in \tau_j$ with $i \neq j$*
   a) *Build a task set $\mathcal{T}'_{i,j}$ by moving all nodes from τ_j to τ_i.*
   b) *Calculate costs of $\mathcal{T}'_{i,j}$ as $co_{new} = cohesion(fn, \mathcal{T}')$.*
   c) *Check all constraints i.e., check if all the following holds:*
      - *Maximum Task Weight: $co_{new} \leq w^+$,*
      - *Minimum number of tasks: $|\{\tau \in \mathcal{T}'_{i,j} \mid \tau \neq \emptyset\}| \geq m^-$*
      - *Prohibitive constraints: $\nexists\ n_i \in \tau_i, n_j \in \tau_j : proh(n_i, n_j)$*
   d) *If merging of $\{\tau_i, \tau_j\}$ is valid w.r.t to constraints, calculate gain for i,j as $G(i,j) = co - co_{new}$. If $G(i,j) > G_{max}$, set $G_{max} := G(i,j)$.*
   e) *Otherwise, set $G(i,j) = -MAX\_INT$ where MAX_INT denotes the maximum integer value.*
3. *If $G_{max} > 0$ with $G_{max} = G(i,j)$, then set $\mathcal{T} := \mathcal{T}'_{i,j}$ and goto step 1.*
4. *If $G_{max} = 0$, return $\mathcal{T}_{start} := \mathcal{T}$ as solution.*

$\diamond$

The constructive nature of this algorithm makes it easy to guarantee that constraints are not violated. The maximum task weight w^+ and the minimum task number m^- are not violated as long as the initial partitioning does not violate them. We

can simply check for each merging operation whether the resulting task number is too small or the task weight becomes to large. Concerning partitioning constraints, prohibitive constraints can be easily respected during the algorithm as well because they are initially satisfied. Then again, for each potential merging operation it can be checked whether this operation is allowed and the solution is rejected if not. As already sketched before, commanding constraints can be considered already before starting the actual task creation process. This is possible, because we always start on the finest level of granularity by assuming initially that each node is modeled as one task. Thus, we can merge each two function nodes f_1 and f_2, if there exists a commanding constraint $command(f_1, f_2)$ before starting the algorithm. For this, we use the node merging operation from Section 5.2 and get a new function node f_{1+2}. This leads to a significant reduction of runtime of the algorithm, because the number of nodes is reduced with each merging operation.

The complexity of the initial algorithm is $O(|N| \cdot |\mathcal{C}|)$ because initially each partition consists of one node $n \in N$ and can be merged with any other node connected with a channel $c \in \mathcal{C}$ leading to n. Based on the start partitioning $\mathcal{T}_{start}$ as the result of the initial algorithm, the KL^+ algorithm is applied. Prohibitive constraints can be checked in the KL^+ algorithm for each potential node exchange or move operation in the same way as for the initial algorithm.

The final result of both algorithms is a set of partitions of nodes where all function nodes of the same partition are merged to create a task. The nodes may be merged in any order, because the merge operation is associative. Empty partitions do not result in a task. To complete the task creation process, it is checked for each local data node and each self-activation if it can be eliminated with the respective operation. Here, local data nodes have to be eliminated before self-activations are tackled, because local data nodes may induce read channels that prevent a semantic-preserving self-loop elimination. Even though both elimination operations are not necessary for task creation, they play an important role to reduce task switches and communication times as motivated in Section 5.2.

5.4. Case Study and Evaluation

As a representative case study to evaluate the approach presented in this thesis, we chose a system from the automotive domain. The system is an advanced driver assistance system named *Virtual Driver Assistant* (ViDAs) [4] and is specified in Simulink as a single-rate model. Beside an adaptive cruise control it additionally contains a lane change assistant and a module to spot speed-limit signs to adjust the speed. Although in reality Simulink models might be of course also multi-rate models, this case study is still relevant because the impact on evaluation results of the task creation approach is relatively small. In the end, different synchronous sets only define additional partitioning constraints because nodes with different rates must not be merged leading to a restricted set of valid solutions.

In a first step, the ViDAs Simulink model was translated into a function network consisting of 183 nodes with 342 channels interconnecting them. To be able to apply

	Parameters					Init. Values		Result Values				
	w^+	m^-	w^*	α	β	$\widehat{w}$	com	T	$w\downarrow$	$w\uparrow$	$\frac{\widehat{w}'}{\widehat{w}}$	$\frac{com'}{com}$
A	0.10	2	0.10	1	1	0.10	0.23	5	0.05	0.09	44%	1%
B	0.20	1	0.20	1	1	0.20	0.23	2	0.15	0.15	26%	1%
C	0.20	7	0.04	1	1	0.04	0.23	7	0.02	0.09	46%	2%
D	0.15	7	0.04	4	1	0.04	0.23	7	0.04	0.05	7%	5%
E	0.15	7	0.04	1	4	0.04	0.23	7	<0.01	0.13	103%	1%
F	0.20	5	0.06	0	1	0.06	0.23	5	<0.01	0.19	119%	1%
G	0.20	1	0.20	0	1	0.20	0.23	3	0.04	0.19	61%	<1%
H	0.15	1	0.15	0	1	0.15	0.23	2	0.15	0.15	2%	1%
I	0.15	1	0.15	1	0	0.15	0.23	3	0.07	0.13	35%	6%

Table 5.3.: Task Creation Results for ViDAs model

task creation for this model, we need worst case execution times for each block to determine weights for function nodes. For standard blocks from the Simulink library a WCET analysis needs to be done only once per target processor. For user-defined blocks, such as Stateflow charts or S-functions, the WCET has to be determined for each model again. To obtain WCETs for our case study, we used the code generation of TargetLink and applied the timing analysis tool aiT [30] to the produced code. As target processors, we defined different variants of the LEON3 and ARM7 processors running at 60 MHz. The transition delays of the respective function nodes were determined as the minimum WCET of all available processor variants. To also be able to calculate channel weights, we assume a FlexRay bus with a maximum bandwidth of 10 MBit/s. The amount of data that needs to be transfered was derived from the data type of the respective signal in the Simulink model.

To evaluate the impact of different parameter configurations, we applied task creation repeatedly to the function network translated from the ViDAs model resulting in a set of alternative task networks. The evaluation results are depicted in Table 5.3. Starting at the left, the table shows a label for the task network from A to I, the maximum allowed task weight w^+, the minimum allowed number of tasks m^-, the desired task weight w^* (determined by m^-) and the weight factors α (for weight balancing) and β (for reducing communication). Then, the initial values for the standard deviation $\widehat{w}$ and communication com are listed.

The node weights of the initial function network vary between 0.0002 and 0.027 leading to an initial average node weight $\overline{w}$ of 0.0016. They are not listed in the table because the are the same for all examples. The result is characterized by the number of tasks T, the minimum and maximum task weight denoted as $w\downarrow$ and $w\uparrow$, and the relation of the final standard deviation $\widehat{w}'$ and communication com' to the respective initial values, which we denote as $\frac{\widehat{w}'}{\widehat{w}}$ and $\frac{com'}{com}$. The runtime for these examples is not shown explicitly and varies between 21 and 32 seconds.

For task network A, we chose a maximum task weight of $w^+ = 0.1$ and a minimum number of tasks $m^- = 2$ leading to a task network with five tasks. The desired number of two tasks could not be reached due to the relatively small maximum allowed task

weight. It can be observed that the communication density could be significantly reduced to 1% of the initial value while the standard deviation of task weights was 'only' reduced to 44%. This has mainly two reasons: First, the initial value of *com* is around three times larger than the initial value of $\widehat{w}$, and thus there is a lot of more saving potential in communication. The other reason is that the maximum task weight forbade more task merging operations to further reduce the standard deviation. For task network B, we increased w^+ to 0.2 and reduced m^- to one. This leads to the smaller number of two tasks because larger task weights were allowed. The reduction of *com* is similar to the previous task network but $\widehat{w}$ is reduced to 26%. This is because we allowed larger tasks, which leads to more merging and thus also better balancing options. In example C, we increased m^- to seven resulting also in seven tasks. Here, $\widehat{w}$ was again only reduced to 46% because already the initial value was significantly lesser than for task network B. Thus, the general savings potential is lesser and a reduction of communication is more promising to reduce the overall cohesion value.

For task network D, we changed the weight factors by increasing α to four and thus preferring the balancing of weights, which results in a task network with seven tasks again. We can observe here, that the adjusted weight factors led to a stronger reduction of the standard deviation to 7%. Contrastingly, the communication was not reduced as much as in the cases before but still significantly leading to a value of 5%. For task network E, we switched the weight factors α and β to prefer the reduction of communication. This is also reflected in the result, where $\widehat{w}$ even increased to 103% but *com* was reduced to 1%. This shows, that the weight factors push the result values into the intended direction. For task network F, we switched off node balancing completely by choosing $\alpha = 0$. In the result, we can observe that $\widehat{w}$ was increased to 119%, which is reasonable because it was ignored for optimization. Instead, communication was again reduced significantly to 1%. In example G, we decreased m^- to 1 leading to an increase of w^* to 0.2. The result shows that also the resulting number of tasks decreases from five to three leading also to a better value of 61% for $\widehat{w}$.

For task network H, we took the same weight factors as for G but reduced w^+ from 0.2 to 0.15. The result is a task network with two tasks and almost the same communication weight *com* as before. In contrast to G, also $\widehat{w}$ was reduced significantly to 2%, even if it was excluded from optimization because α was set to zero. A possible reason is that the heuristic coincidentally chose a solution with a very low standard deviation and this solution was also the best one with respect to communication. In the last network I, we used the same parameters as in H but set $\alpha = 1$ and $\beta = 0$, which ignores the communication part of the cohesion function and should optimize the standard deviation of weights. This leads to a task network with three tasks and a reduction of $\widehat{w}$ to 35%. This value is clearly worse than in the previous example H where balancing weights was not considered at all. A reason for this unexpected behavior may be that the weights of the two tasks found in example H were both very close to the maximum allowed weight of 0.15. Thus, the result is an almost perfectly balanced task network with very few communication, which can hardly be improved. In example I, the resulting network consists of three tasks. This means that the heuristic probably has chosen a different order of merging operations and could not merge two of the three remaining tasks into one without violating the maximum task

	Result Values (Initial Algorithm)					Result Values (KL^+)				
	T	$w\downarrow$	$w\uparrow$	$\frac{\widehat{w}'}{\widehat{w}}$	$\frac{com'}{com}$	T	$w\downarrow$	$w\uparrow$	$\frac{\widehat{w}'}{\widehat{w}}$	$\frac{com'}{com}$
A	5	0.04	0.09	45%	14%	5	0.05	0.09	44%	1%
B	2	0.10	0.20	37%	<1%	2	0.15	0.15	26%	1%
C	7	<0.01	0.09	59%	1%	7	0.02	0.09	46%	2%
D	7	0.02	0.06	24%	3%	7	0.04	0.05	7%	5%
E	7	<0.01	0.13	103%	1%	7	<0.01	0.13	103%	1%
F	11	<0.01	0.20	114%	2%	5	<0.01	0.19	119%	1%
G	11	<0.01	0.20	92%	2%	3	0.04	0.19	61%	<1%
H	2	0.15	0.15	2%	1%	2	0.15	0.15	2%	1%
I	3	0.07	0.13	39%	3%	3	0.07	0.13	35%	6%

Table 5.4.: Comparing ViDAs Results of Initial Algorithm and KL^+

weight. This leads to the significantly higher difference in task weights and also to additional signals between tasks and thus a higher communication density. A further observation is that the value for *com* of 6% is the worst of the examples considered here. However, this is reasonable because it was not considered for optimization in example *I*. Nevertheless, a value of 6% is still a significant reduction.

The reason why communication is reduced even if it is excluded from optimization lies in the approach of the initial algorithm, where only partitions are merged that communicate with each other. Thus, with each merging step communication is always reduced. A general observation is that the minimum task weight ($w\downarrow$) was significantly increased for most examples from initially 0.0002 to values between 0.05 (factor of 250) and 0.15 (factor of 750). Two exceptions are the examples E and F with minimum task weights below 0.01. In both cases, the reason is the choice of the weight factors α and β leading to a focus on optimizing communication. While for task network E the minimum weight could be increased at least by a factor of 10, in example F the minimum task weight remains the same as for the initial function network. This is because α was set to zero and thus weight balancing was completely ignored.

In summary, we can state that the main goals of task creation, which are the reduction of communication and an increase of the minimum task weight by balancing node weights, are satisfied by the implemented algorithm. Furthermore, we can observe that the weight factors α and β lead the result in the intended direction as long as both are greater than zero. If one factor is set to zero, the behavior seems to become more random and unpredictable, which may lead to good solutions as well but leads to missing means of control. Furthermore, it can be observed that the constraints in terms of the maximum task weight w^+ and the minimum number of tasks m^- are always satisfied.

To investigate the influences of the KL^+ algorithm, in Figure 5.4, the intermediate results of the initial algorithm (shown on the left) are compared with the final results (shown on the right). For most examples the number of tasks is equal to the final result because the initial algorithm already merges partitions until the minimum number of tasks is reached or the maximum task weight is violated. Nevertheless, this may still

System		Param.		Init. Values		Result Values				Time (s)	
#N	#C	m^-	w^*	$\widehat{w}$	com	T	$\overline{w}$	$\frac{\widehat{w}'}{\widehat{w}}$	$\frac{com'}{com}$	IA	$\sum$
50	73	5	0.26	0.23	0.34	6	0.21	23.5 %	8.4 %	<1	<1
75	109	7	0.24	0.22	0.54	9	0.18	57.9 %	9.8 %	<1	1
100	144	10	0.24	0.22	0.68	13	0.19	32.4 %	14.4 %	3	4
150	215	15	0.21	0.19	1.06	16	0.20	23.9 %	8.8 %	8	11
200	285	20	0.23	0.21	1.40	22	0.21	25.6 %	7.7 %	19	23
250	356	25	0.21	0.16	1.78	25	0.21	22.2 %	8.1 %	37	52
300	427	30	0.24	0.21	2.18	34	0.21	28.2 %	9.8 %	64	90
400	568	40	0.21	0.19	2.87	42	0.20	28.3 %	6.5 %	152	179
500	710	50	0.22	0.20	3.59	51	0.22	35.9 %	7.8 %	312	355

Table 5.5.: Task Creation Benchmark Results

happen as in the examples F and G. One reason may be that the initial algorithm is restricted to merge only those partitions that communicate with each other. If there are partitions that do not communicate, they may be first merged by the KL^+ algorithm. Furthermore, the initial algorithm does not move single nodes between partitions as the KL^+ does. Thus, the main influence of the KL^+ algorithm is to balance the weights of the already existing partitions. This can be best observed when looking at the minimum task weight, which is significantly increased for most created task networks after the KL^+ algorithm was applied. For example, for task network C the minimum task weight is increased from 0.002 to 0.02 and for task network G from 0.0002 to 0.04. But also the communication is often improved by the KL^+ algorithm, as it can be observed at task network A, where the initial algorithm decreased *com* to 14% and the KL^+ algorithm further improved the solution to 1%.

To further evaluate the approach in terms of scalability with the number of nodes, a generator for artificial benchmarks was developed. It creates function networks that are adapted from the typical structure of Simulink models. This means in particular, that there may exist several synchronous sets that are connected blocks with the same sample time. One synchronous set typically consists of parallel block chains that may be joined at certain blocks and split again at another block. Those blocks often model a controller in terms of a Stateflow chart, taking decisions based on input signals and forwarding output signals to further blocks. We generated artificial function networks consisting of between 50 and 500 function nodes with weights between 0.002 and 0.1. The input parameters restrict w^+ to 0.3 and m^- to 10% of the number of nodes #N. The weight factors for the cohesion function where chosen as $\alpha = \beta = 1$.

The results are shown in Table 5.5. The different benchmarks are characterized by their number of nodes #N and their number of channels #C. The parameters and result values are shown in the same manner as known from the previous tables, while here the average task weight $\overline{w}$ is regarded as well. On the right, the runtime in seconds $Time(s)$ of the initial algorithm IA and the overall runtime of both algorithms $\sum$ is depicted. The results confirm the general observations we made for the ViDAs system: The communication density *com* is always decreased clearly to values between 6.5%

and 14.4% of the original communication. The standard deviation of task weights is also reduced but not in the same extent leading to values between 22.2% and 57.9%. Furthermore, the average task weight $\overline{w}$ is clearly increased and is very close to the desired weight w^*, which is the expected value for the standard deviation. Nevertheless, the reduction of communication is less significant compared to the task networks we created for the ViDAs case study. One reason is that we chose as minimum number of tasks a value of 10 % of the initial number of nodes, which is greater than the values we used for the ViDAs system. Furthermore, the benchmarks are only approximations for Simulink models and thus they may offer less potential for saving communication than real models.

The runtime scales well also for networks with up to 500 nodes, where an overall runtime of 355 seconds (5 minutes and 55 seconds) is needed to get a result. It can be observed that the initial algorithm consumes a larger part of the overall runtime than the KL^+ algorithm. The reason is that the initial algorithm starts with the maximum number of partitions, which is equal to the number of nodes of the initial function network. Contrastingly, the KL^+ algorithm only has to deal with the number of partitions that results from the initial algorithm, which is significantly less.

5.5. Summary and Related Work

The task creation approach was first published in [12]. Its starting point is a function network modeling several software functions that should be deployed on a distributed hardware platform. Typically, this function network is derived from a specification model, where we assume Simulink as specification language for this work.

Task creation is necessary to get rid of the fine granular structure of the original specification model and the respective function network translation. This structure is usually not suitable to represent application tasks because it consists of a high number of nodes that are often of very small computational size. Furthermore, there is a lot of communication between nodes, which has to be allocated to a bus if the communication partners are mapped to different processing units. The goal of task creation is to partition the nodes of a function network into a set of tasks, where communication is minimized and computational weights are balanced. This is realized by defining an optimization metric called cohesion where nodes are attracted by high communication density and repulsed by high weights. Furthermore, we defined constraints that restrict the set of allowed task partitions. Beside the possibility to set limits for the number and weight of tasks, also structural constraints were defined, where certain partitioning options may be demanded or forbidden. Thus, the user may guide and restrict the process and also define the desired granularity of structural elements to start with.

When partitioning nodes with the goal to build a task from each partition, this also has impacts on the function network representation. Thus, it is not sufficient to partition the nodes because this has no semantic consequences. Instead, task creation means a merging of a set of nodes into a single node. To realize this, a set of formal composition operations has been defined, where each operation replaces a component of the function network by another one with the same interface. The first operation

merges two nodes while maintaining all transitions and ports of the original nodes. This is a mandatory operation and is defined for any two function nodes without any restrictions. To complete the formal process of task creation, we defined two further operations, which may be performed if specific conditions hold. The first operation is the elimination of local data nodes, which is useful to get rid of data nodes that are only used by one function node, and thus can be considered as local memory. The second operation removes self-activations by concatenating subsequent transitions to one transition. To preserve semantics of the original network and also the specification model, we have shown for all operations that they preserve causality of input and output events of the component interface. Furthermore, we proved that the property of state-independence is preserved, which allows to decide boundedness.

With the knowledge how to formally perform task creation, we continued with defining an heuristic approach to obtain a set of tasks by partitioning the nodes of the original function network. For this purpose, we applied an extension of the Kernighan-Lin algorithm where its start solution is determined by an initial algorithm. The optimization goal for both algorithms is the cohesion metric defined previously. This approach has been applied to a case study of a driver assistance system modeled in Simulink. The results show that in particular the amount of inter-task communication could be significantly reduced, and task weights were balanced, as intended by the optimization metric. Furthermore, we defined a set of artificial benchmarks in terms of function networks that were created by emulating the typical structure of Simulink specifications. The algorithm was applied to these benchmarks as well, where the number of nodes was varied to also evaluate the scalability of the approach. The results show an improvement of the cohesion similar to the case study, and the runtime was acceptable also for function networks with up to 500 nodes. Furthermore, the scalability of the approach can be significantly improved by adjusting the granularity of the initial function network by defining respective partitioning constraints.

Concerning future work, possible extensions of the task creation approach can be found, for example, in the algorithmic part. Here, optimizations may be applied to the currently used algorithms to further improve the quality of the results or reduce complexity. Additionally, it may be useful to investigate in more detail how an engineer would decide which nodes should be merged at concrete examples. Based on this knowledge, it could be analyzed whether this is already covered by the current approach or how it may be extended to also consider these issues.

Furthermore, investigations are useful concerning the behavior of worst case execution times when two node transitions are concatenated to one. Currently, we assume approximately that the delay of the concatenated transition is the sum of the single transition delays. Performing a WCET analysis for each potential merging is not viable because the code must be annotated, for example, with loop bounds, and the analysis is too time-consuming to perform it such frequently. Additionally, execution times often behave differently for different processor architectures and compilers, which makes the effect of node merging hard to predict.

When turning to the formal part of composition operations for node merging, we can think of an extension of the operation to eliminate self-activations to make it completely applicable to function networks translated from Simulink. Up to now, this

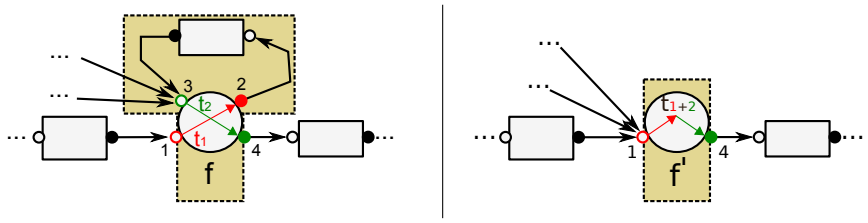


Figure 5.10.: Extension of Self-Activation Elimination to Multiple Input Channels

operation can only be applied if the input port of the self-activation has no further incoming channels because otherwise causality could not be preserved. Although this is correct, a relaxation of the claim for causality would also allow to consider an input port with multiple input channels.

The basic idea is exemplified in Figure 5.10, where a self-activation loop with a signal data node is depicted with multiple input channels at input port 3. On the right, it is shown how we may also remove such a self-loop by redirecting all remaining input channels of port 3 to port 1. This moves the channel synchronization to another input port of the same function node. This had to be done for all input ports with causal dependencies to port 2 i.e., the port where the self-activation loop starts. By doing so, we would add new causal dependencies because the events of the redirected channels have to be available already for the activation at port 1 instead of port 3. Thus, all executions starting at port 1 have to wait for these events to be available. This may delay node executions by the respective maximum wait delay of the synchronization buffer. Nevertheless, the partial order of Simulink would still be valid because we only refine this partial order by adding new relations. The advantage of this extension would be that more self-activations could be eliminated, which reduces the number of task activation points in terms of input ports.

Related Work

One of the publications with most similarities to the task creation idea has been published by Di Natale et al. [23]. The authors propose an optimization of the multi-task implementation of Simulink models with real-time constraints on a single processor. Their optimization goal is to reduce the use of rate transition blocks between different synchronous sets to minimize buffering and latencies. Tasks are either determined by the synchronous sets, or their creation is modeled as a part of the optimization problem. Furthermore, task priorities and execution order of functional blocks within a task are optimized. The main difference to our work is a different hardware target architecture, which has several implications on the remaining parts of the work. While Di Natale et al. consider a single processor where tasks should be executed with a rate monotonic scheduling scheme, we are aiming at a distributed hardware architecture with a pre-deployed task network. This is of central importance for our optimization function, which is specialized to this problem. Their optimization goal is in a way orthogonal to ours, because it is targeted to a single processor, where communication

is known to be local. Thus, buffering latencies can be minimized, which is not useful for a distributed system before tasks are mapped to concrete processing units.

Another work from Kugele et al. [46] is also based on a synchronous language and presents a way to deploy clusters, which are actually tasks, specified by the COLA language on a multi-processor platform. This allocation process is completed by a scheduling analysis involving address generation and estimation of memory requirements for a pre-defined middle-ware. In this process, the authors also raise the question of how to generate clusters of nodes but assume that this is a decision that is taken manually by the user. Thus, there is no optimization goal given comparable to ours and no automated process as our algorithmic approach. Furthermore, their focus lays on a scheduling analysis, which is not part of our work and there exists no pre-deployed task network in their scenario.

In the work of Tripakis et al. [18], a Simulink model is translated into a Lustre model to partition the generated code into modules that are executed on different processors communicating via a time-triggered bus. Formally, Simulink models are also defined as timed synchronous block diagrams, and semantics is preserved by considering the partial order on block executions as well. Contrary to our work, the focus lies on separating the generated code into different modules respecting the partial order, and performing a scheduling analysis for user-specified timing constraints. Furthermore, the code partitions are assumed to be given by the user and not derived automatically with a specific optimization goal. Producing modular sequential code from synchronous data-flow networks is also addressed by Pouzet et al. [63]. They decompose a given system into a minimum number of classes executed atomically and statically scheduled without restricting possible feedback loops between input and output. For both approaches holds that the question of efficient and modular code generation lies beyond our approach and can be esteemed as supplementary.

Formal Composition Operations. A work that is related to our formal composition operations is presented in [1]. There, a modeling formalism called *model algebra* is defined, which may be used for model specification and refinement. It consists of executable components, called behaviors, which may communicate either by channels or variables. Channels have a a double-handshake synchronization semantics while variables allow asynchronous read and store of values. Thus, variables can be compared to data store memory blocks in Simulink and shared data nodes in function networks. The execution semantics is defined in terms of a so-called Behavior Control Graph (BCG), which has some similarities to Kahn Process Networks. The author defines an equivalence relation called functional equivalence on BCGs where two BCGs are equivalent if the value-traces of all their variables are identical. In our work, we do not explicitly model functionality in terms of concrete values of variables. If we consider a Simulink model, we claim that functionality is preserved if the partial order of block executions is maintained. Data store memory blocks are explicitly excluded from this notion of functional equivalence and the same holds for shared data nodes in function networks. Thus, it is not specified in which order such a variable is read or written.

As a next step, the author defines a set of transformation laws defined as, for example, the flattening of hierarchical behaviors, or the relaxation of control flow. Based on these laws, functionality-preserving refinements are defined that aim at representing design decisions in the model ending in a cycle-accurate model of hardware and software elements. For example, one refinement is to represent the mapping of behaviors to processing elements by creating copies of each node in each processing element. Another refinement aims at representing a static schedule of tasks that are mapped to the same processing element by sequentializing them accordingly. It is shown that this operation preserves functionality if there are no data dependencies in terms of variables between tasks. These refinements have some similarities to task creation in their basic idea to represent design decisions. Nevertheless, they are different in detail. The main difference is, that in [1] processing elements are assumed to have no scheduling mechanism and thus a fixed static scheduling of tasks in terms of a total order is determined. In contrast to this, we assume a scheduling scheme to be applied on processors at runtime and do not need to determine a static schedule. Thus, we can restrict to show that our operations do not violate the causalities derived from the partial order of Simulink blocks.

Another related work stems from Henzinger and Matic, who describe in [37] a concept of *bounded-delay interfaces* and operations on these interfaces. These allow a composition of interfaces, a composition of task groups, a connection of an interface with a task-sequence and a refinement relation between interfaces. This enables incremental design and independent implementation. The composition of task groups has, on the first glance, some similarities to the task creation idea. But the details and, most of all, the goal is quite different. First, their goal is to find an abstract representation in terms of an abstract resource model of a task group running on the same resource. Second, they consider a concrete scheduling strategy and this directly influences the resulting abstraction. This leads to the fact that their operation is not associative. Third, their overall goal is to compose a system from different components without the need to know their implementation but only their interface. Our goal is to represent a merging of tasks without knowing anything about the resource that should compute them or the applied scheduling strategy. Additionally, we do not apply our operation on an abstract model, as event streams or arrival functions, but on the task network itself. Our goal is to maintain the causality of input and output events even if we are still able to derive an abstraction in terms of event patterns.

A related work from another community is the theory of latency insensitive design [17]. There, a problem is addressed that is highly relevant for system on chip (SoC) design. Here, the length of wires may lead to the problem that the induced communication delay becomes greater than the cycle period, which does not fit anymore to the synchronous assumption. Thus, some components cannot be executed in the correct clock cycle. To solve this problem, relay stations (buffers) are introduced into the design to store intermediate data. Thus, the respective components can still compute with correct values even if the clock cycle has already passed. The number of needed relay stations can be computed from the delay length and the clock cycle length. Those relay stations may be inserted without violating the functional correctness of the model in terms of ordered sequence of data that is passed on channels.

This equivalence property is called *latency equivalence*. It is further shown that this property is compositional and therefore allows to construct e.g. a model from single components of different vendors without violating functional correctness. In principle, the described problem may also occur in a model derived from a Simulink specification. Here, it does not occur due to the cable length, but due to the time needed for execution or communication. A significant difference is that in a function network, we always have buffers within function nodes that store incoming events such that no data is lost. Nevertheless, in the case of non-algebraic control loops, it has still to be guaranteed that data is available in time leading to the end-to-end deadlines we defined for Simulink models.

Algorithms for Graph Partitioning and HW/SW Partitioning. Because hardware software partitioning can be considered as a subclass of graph partitioning, we will also refer to work in this area and in particular the algorithms that are used. An algorithm that is often used, especially in the earlier times of hardware software partitioning, is Simulated Annealing (SA). Beside the COSYMA system [28], which uses SA to partition the components starting with all components realized in software, there is another approach described in [27], which uses SA for partitioning.

One popular algorithm that is widely used, is the heuristic of Kernighan-Lin (KL), which is for example applied in [45] for hardware software partitioning. Another field, where KL is very popular and commonly used, is the area of parallel computing. In [57], also the graph partitioning problem is addressed, where the authors propose a new method to compute such partitions while focusing on the aspect of parallelism. Furthermore, an overview of techniques is given that were used so far in the area of system on chip design. They state that most of the state-of-the-art approaches use a variant of the KL heuristic. But due to the fact that this algorithm is sequential, it is not well suited for parallelization and alternative techniques have been developed like Bubbble-FOS/C [58]. With the help of a diffusion scheme, it is determined how 'well connected' two nodes are, which means that they are connected by many paths of small length. This delivers high quality results but is comparably slow in computation time. In this paper, the authors propose a new method that overcomes this drawback and can compete with the state-of-the- art implementations while offering even better results. Compared with our work, there seem to be some similarities of our cohesion metric to the diffusion scheme described in [57], which also considers a measure on the connectedness of nodes. Nevertheless, the question of parallelization is out of scope for this work but may be worth to be considered in future.

A variant of the KL algorithm is the Fiduccia Mattheyses (FM) heuristic, which is compared in [59] to a complete approach, which delivers an optimal solution. Because FM is a heuristic, it does not necessarily create an optimal solution but has a significantly shorter runtime than the optimal methodology and is thus applicable also for systems with a larger amount of nodes.

Vehicle Routing Problem. Similar problems as in the area of graph partitioning also occur in other domains, where we pick the vehicle routing problem (VRP) as an

example from the transportation and logistics domain. The vehicle routing problem [50] can be described as a generalized problem of the traveling salesman problem. It is motivated from the logistical background of delivering goods from a depot to a set of customers. It can be described as a graph where a set of n vertices represent $n-1$ customers and the depot, and the arcs represent direct connections between two customers. The distance of customers connected by arcs is given by a distance matrix. The problem is to find an optimal set of routes starting and ending at the depot to deliver the goods to all customers while satisfying certain constraints. The criteria for optimality are typically defined in terms of overall travel costs or travel time. To solve this problem, beside well-known techniques like Tabu Search, Simulated Annealing and genetic algorithms, also some special algorithms are applied, where the *Savings* and the *Sweep* algorithm are the most common.

[64] presents an overview of the history and application of the *Savings* method. It starts with a single route for each customer and calculates savings in terms of distance for each pair of customers if both would be on the same route. Beginning with the pair with the highest saving, two customers are linked to the same route until a constraint is violated such as the maximum route length or vehicle capacity. New links can only be added to the start or end of a route. This method has a lot of similarities to our initial algorithm. We also start with a solution where nodes are completely separated in partitions and iteratively merge two partitions with the best savings in terms of communication costs and task weight imbalance.

Another heuristic often found when solving the VRP, is the Sweep algorithm [70]. It sorts all customers by the angle their location has to the location of the depot. Then, it creates a new route starting with an arbitrary customer and adds customers from the list to the route until any constraint is violated. The constraint is here the maximum load of the vehicle in terms of weight and volume. This approach is hardly applicable to the area of graph partitioning because the geographical position of a node does not have any relevance for the solution. Nevertheless, it could be regarded as kind of neighborhood relation, which is often used for optimizing algorithms as, for example, in the Quick Cut algorithm [25].

6. Design Space Exploration

While this work deals with the creation of task structures from a Simulink specification model, it is embedded into a design space exploration framework that covers the complete design flow down to the implementation level. This framework addresses a scenario where an existing system of software tasks mapped to a distributed hardware architecture is extended by a new feature in terms of a Simulink model. The hardware architecture is assumed to be hierarchical, as it can be found in the automotive domain. Those architectures typically have a backbone bus which connects a number of subsystems. Each subsystem consists of a set of ECUs connected by a local bus.

In Figure 6.1, an example of an architecture with three subsystems is depicted. The backbone bus at the top is a *FlexRay* bus while the local buses within the subsystems are *CAN* buses. The ECUs within a subsystem may already be utilized by tasks of the existing task network depicted on the left. The dotted arrows indicate that a certain signal is allocated to a bus slot or a task to an ECU. This induces a utilization for each ECU. An ECU with utilization zero is currently not used and thus considered as a potentially new ECU, which can be added to get more computation capacity. Furthermore, ECUs that are already in use can be upgraded e.g. by increasing the clock frequency or by switching to another processor type.

With the methods presented in Chapter 4, a Simulink model is translated into a function network and with the approach from Chapter 5 a task network is created, which should be allocated to the available hardware resources. The goal for the remaining design space exploration step is to keep the monetary costs that are needed to modify the hardware architecture as low as possible. Costs may be induced by either adding new ECUs or by replacing existing ECUs by other types that offer more

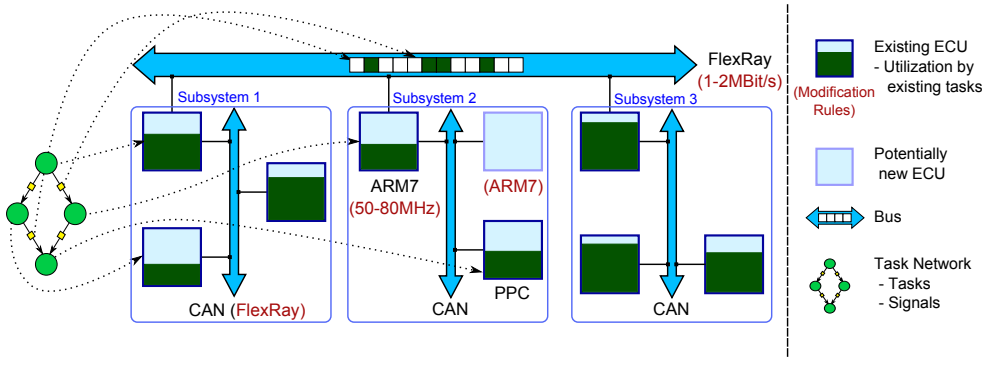


Figure 6.1.: Example of Hierarchical Target Architecture (Source:[11])

computing capacity for the allocated task set. Furthermore, there may be defined mapping constraints, which allow or forbid the mapping of tasks to the same resource or to a specific ECU type.

For the design space exploration process in the context of this framework, there currently exist three competing approaches: The first one was published in [10, 11] and pursues a two-step approach where the problem is divided into a global and a local analysis phase. The work of [77] is also based on a local and a global phase but proposes alternative techniques to tackle the problem. Furthermore, based on the work in [2], a holistic approach is investigated that aims at solving the problem as a whole. In this work, we will rely on the first approach.

In [11], the hierarchy of the target architecture is used to split the process into two phases: First, in the *global analysis*, each task of the task network is mapped to a subsystem while a share-based metric is used to estimate the costs induced by hardware modifications in a subsystem. Here, the minimum WCET of a task over all processor types is considered, which enables the global analysis to guarantee a lower bound for needed modification costs. In a second step, a *local analysis* is performed where tasks that were assigned to a subsystem by the global analysis are allocated to ECUs of that subsystem. Here, an exact scheduling analysis is performed to assure system feasibility. Thus, we gain a response time for each task and each signal that could be allocated successfully. For the local analysis, end-to-end deadlines are divided into local deadlines for each subsystem they affect by the so-called *deadline synthesis*. These deadlines are re-synthesized after each local analysis, where the already obtained response times of tasks and signals are regarded. This may lead to relaxed deadlines for remaining tasks and signals.

In a *backtracking* phase, those tasks that could not be mapped to a subsystem under the globally proposed costs, are passed back to the global analysis in a so-called *odd set*. Based on the knowledge gained in the local analysis, the global analysis proposes a new distribution of all tasks of an odd set. The allocation of tasks that could already be placed successfully by a previous local analysis remains unaffected. This process is repeated until a feasible solution is found or all mapping options for tasks and signals were considered as not being feasible with the given set of hardware modifications.

In this section, we will discuss the role of task creation within this design space exploration scenario and propose some patterns how to adjust the task network based on the previous analysis results. This may be, for example, realized by changing the minimum and maximum task weights or adding further partitioning constraints. Furthermore, we will evaluate the task creation approach in interaction with the design space exploration by considering different task networks for the ViDAs case study and observing the effects for the resulting deployment and costs.

Outline In Section 6.1, we give an overview of the design space exploration approach as it was proposed in [10, 11]. In Section 6.2, we reason about the role of task creation in the context of the design space exploration process and use the ViDAs case study to evaluate the overall approach in Section 6.3. Section 6.4 concludes the chapter with a summary and discussion of the results.

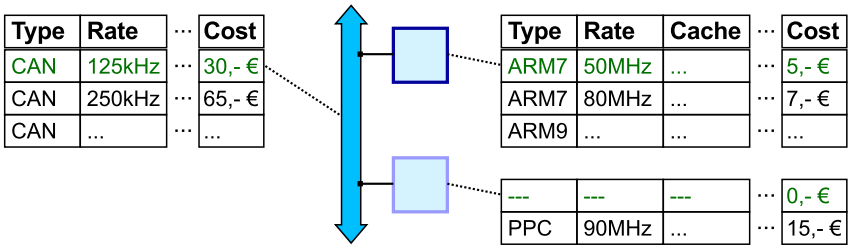


Figure 6.2.: Examples for Modification Rules (Source:[11])

6.1. Overview of Design Space Exploration Process

The design space exploration consists of a global and a local analysis, which are iteratively repeated until either a solution was found or the system is not feasible under the considered set of modifications and costs bounds. After each local analysis, the results are used to further constrain the global analysis in a backtracking step.

The starting point is an existing hardware architecture with a pre-deployed task network as it is depicted in Figure 6.1. Tasks are mapped to ECUs and signals between tasks are mapped to local buses or the global bus, if the respective tasks are located on different ECUs. This leads to a certain utilization for each ECU and a bus schedule for the global bus. There may exist ECUs where no task is allocated to, leading to a utilization of zero. Those ECUs are considered as unused and are not part of the existing system. During design space exploration, a new task network should be mapped to the existing system. Because this system is already utilized by the pre-deployed task network, the *new* task network cannot necessarily be deployed without doing any modifications. Such modifications allow to add further ECUs, which were unused before, by mapping tasks to them or by upgrading already used ECUs. Those upgrades may change the frequency or memory of a processor, or replace it by another more powerful ECU type. This is exemplified in Figure 6.2, where a snapshot of the architecture is shown. On the left, possible updates of a local bus are listed and on the right for two ECUs different modifications are shown. The processor at the top is currently of the type ARM7 running at 50 MHz and may be replaced either by an ARM7 with 80 MHz or by an ARM9 processor. The processor at the bottom is not part of the existing system and thus creates zero costs. If this processor is added as a PowerPC (PPC) running at 90 MHz, it would induce additional costs of 15 EUR.

6.1.1. Global Analysis

In the global analysis, all tasks of the new task network should be mapped to subsystems. To estimate the costs for each considered mapping option, *costs functions* are defined. First, for each ECU the remaining capacity is determined by considering the utilization by pre-deployed tasks. This is done for the currently used ECU type but also for each allowed modification. The different capacities are related to the costs of the respective modification resulting in a step function depicted in Figure 6.3 on the left. The x-axis describes the offered capacity and the y-axis the needed costs. Thus,

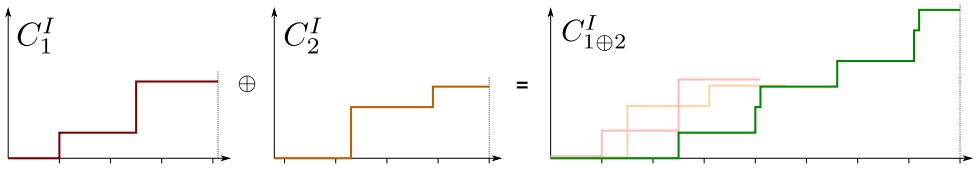


Figure 6.3.: Calculating Costs Functions (Source:[11])

each step represents a modification, which increases costs but also offers additional capacity. In this example, we create two costs functions C_1^I and C_2^I for two ECUs namely 1 and 2. Each of them offers two modifications to increase the initially offered capacity. For the global analysis, we need the costs function for a whole subsystem because concrete ECUs are not regarded. The costs function of a subsystem considers all possible combinations of modifications that offer the needed capacity and returns the minimum overall costs of all such modifications. It is depicted in Figure 6.3 on the right. For example, to get an overall capacity of 3, we might take the modification of ECU 1 with costs of $C_1^I(3)$ or the modification of ECU 2 with costs of $C_2^I(3)$. But we may also allow for a combination of modifications of both ECUs leading to costs of e.g. $C_1^I(2) + C_1^I(1)$. The minimum of all such modification options determines the costs function of a subsystem. More details on the calculation of costs functions can be found in [10, 11].

To be able to apply the costs functions during global analysis, the needed capacity for a certain allocation of tasks to subsystems needs to be estimated. This is done by considering for each task that should be mapped to the respective subsystem the minimum WCET over all processor types that are allowed for this subsystem. Taking the minimum WCET enables us to get lower bounds for the costs that are needed to place a set of tasks on a subsystem. Each real allocation of tasks would induce higher or equal costs as the costs proposed by the costs function, but never less costs. If we would take the maximum WCET or an average over all ECUs, the real costs could be higher or lower as well, because the response times of tasks are not only determined by WCETs but also, for example, by blocking times from other tasks on the same ECU.

This covers the estimation of the computational capacity that is needed to realize the system. To also estimate the global communication induced by a certain task partitioning, we need to find a valid allocation of signals that are sent between tasks to the global bus. The global bus is assumed to be a time-triggered bus with a static bus schedule i.e., we reduce the FlexRay bus to its static segment. This *bus feasibility analysis* returns response times for each global signal if there exists a feasible schedule. Otherwise, the currently considered task partitioning is not feasible.

Based on the costs functions and the bus feasibility analysis, the global analysis has the goal to find an allocation of tasks to subsystems with minimum overall costs and a valid global bus schedule. The first part is a variant of the classic graph partitioning problem, where costs are not determined by edge costs but by allocation costs. Due to the results of the comparison of different algorithms from Section 5.3, we implemented again the KL^+ algorithm to perform the global analysis. The bus feasibility analysis

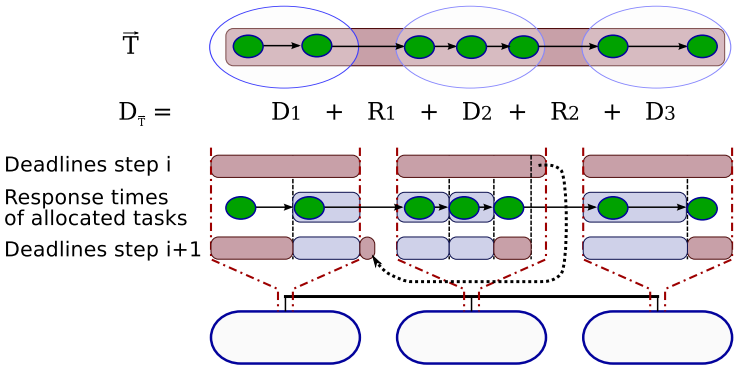


Figure 6.4.: Deadline Synthesis and Re-Synthesis (Source:[11])

is performed for each solution proposed by the KL^+ algorithm. A solution is rejected if there does not exist a feasible schedule.

When the global analysis step has finished with a valid task allocation, another step is needed before the local analysis can be performed. This step is called *deadline-synthesis*. Assuming a deadline of a task chain $\tau_1, ..., \tau_n$ that is distributed over different subsystems, this deadline cannot be checked by a local analysis which only considers one subsystem at a time. Thus, we need to determine deadlines that are local to a subsystem, which we denote as deadline-synthesis. The basic idea is to split the deadline into a set of deadlines for each chain of tasks that are allocated to the same subsystem. For deadline synthesis, the response times of already deployed global signals are considered. The length of each synthesized deadline is determined by relating the sum of its task weights to the length of the remaining deadline.

This is exemplified in Figure 6.4. At the top, a task chain $\vec{T}$ is depicted where those tasks that are surrounded by ellipses are allocated to the same subsystem. Between succeeding tasks of different clusters, response times occur due to global bus communication, which are denoted here as R_1 and R_2. The deadlines D_1 to D_3 for the three clusters are determined such that their sum and the global bus response times result in the original deadline $D_{\vec{T}}$. The remaining part of the figure describes the deadline re-synthesis, which is explained in the next paragraph.

The result of one global analysis step consists of an allocation of tasks to subsystems, the estimated costs for each subsystem, a feasible schedule for the global bus, and a set of synthesized deadlines for each subsystem.

6.1.2. Local Analysis and Backtracking

The local analysis is performed for each subsystem, where tasks were allocated by the previous global analysis. Its goal is to find a feasible ECU allocation for all tasks of the subsystem and to determine response times for them. The maximum costs for modifications is limited by the estimated costs of the global analysis. Please note that there may exist several modification options resulting in the same costs. Due to the

fact that the cost limit is based on a share-based estimation, there may be no feasible solution for mapping all tasks to ECUs under this cost limit. In this case, the local analysis determines a so-called *odd set*. This is a minimum set of tasks that are not deployable while for all remaining tasks a feasible allocation exists. If the odd set is empty for all subsystems, the allocation proposed by the global analysis is feasible under the estimated costs and the process finishes with this allocation as final result. If any odd set is not empty, we enter the *backtracking* phase.

This phase consists of different measures that may be taken to improve the results of the global analysis by considering the knowledge gained during the local analysis. First, synthesized deadlines are updated with the help of the response times of tasks that were successfully deployed by the local analysis. This is depicted at the bottom of Figure 6.4 starting with the deadlines of step i. These are the deadlines before the local analysis has started. In the next line, the response times of those tasks are considered that have been deployed to ECUs by the local analysis. Because the odd set is not empty, there are still undeployed tasks without response times, which is the leftmost task here. The goal of the deadline re-synthesis is to relax those deadlines by adding time that is not used by already deployed tasks. This means, that the difference of deadline and response times of all deployed tasks can be distributed on the remaining tasks. For the example in Figure 6.4, this is indicated by a dotted line from the middle cluster to the left cluster. The deadline of the middle cluster is greater than the sum of all its response times and this difference is added to the deadline of the undeployed task on the left. With these relaxed deadlines another local analysis is performed while all deployed tasks remain deployed. Thus, the new odd set may only contain tasks of the former odd set.

The second measure of backtracking is to modify local buses to obtain lesser response times for communication. If this does not lead to a feasible solution, the global analysis is started again for the tasks of all odd sets. All remaining tasks are assumed to be deployed and their calculated response times are taken into account when determining the costs function. This process is repeated until either a feasible solution with empty odd sets can be found or under consideration of all possible modifications the new task network cannot be completely deployed. In the latter case, the process is finished with an intermediate solution still containing undeployed tasks.

6.2. On the Role of Task Creation

The role of task creation during the design space exploration phase is to determine the granularity and number of tasks of the new task network. This affects the quality of the results in terms of needed modification costs, the degree of freedom in terms of possible allocation options, and also the runtime of the different analysis steps. For example, task creation may merge only a few nodes leading to a very high number of tasks with comparatively low weights. This allows a lot of allocation options for design space exploration but also raises the runtime of the KL^+ algorithm and the local analysis. Furthermore, many tasks also lead to more task switches, which increases thrashing as discussed in the task creation chapter. This may also lead to higher costs because

ECUs cannot be used efficiently. Contrastingly, task creation may also deliver a task network with a small number of tasks but with large task weights. This would reduce the runtime of the KL^+ algorithm but also restricts the freedom of the global and also the local analysis to distribute the needed capacity on different computation resources. This may lead to higher costs as well if a task is very computational intensive and may only be executed by a powerful but expensive ECU type. And because design space exploration cannot split a task, it has to choose this ECU to make the system feasible.

This shows that the interaction between task creation and design space exploration is highly sensitive to the concrete application, and its structure and distribution of node and channel weights. Thus, this step needs to be transparent and cannot be completely automated because it requires the expert knowledge and experience of the user. Nevertheless, we will sketch some typical scenarios that might occur during design space exploration, describe how they could be detected, and propose proper refinement measures. These measures describe how the task creation parameters in terms of weight factors, maximum task weight, minimum number of tasks and partitioning constraints may be adjusted to improve the overall result.

Bus Overload One typical scenario is a situation where many promising and good solutions are rejected due to a missing feasible schedule for the global bus. Thus, the global analysis has to choose an allocation with potentially higher costs where the global bus is feasible. To detect such a situation, each time an allocation led to an infeasible bus schedule, the estimated costs may be memorized. When the global analysis has finished, the final result can be compared to the rejected results to determine if those solutions would be better with respect to costs. If there were many solutions rejected with less costs, this is an indicator to conduct measures to release the global bus. This may be done, for example, by increasing the weight factor β (or decreasing α) to prefer task networks with less inter-task communication by allowing more imbalance of task weights. Additionally, the maximum task weight w^+ may be increased and the minimum number of tasks m^- may be decreased because a less number of tasks with higher weights may also lead to fewer communication density.

For local bus overload, this might be done similarly while it highly depends on the chosen local analysis technique whether and how this problem might be detected.

Wasted Bus The converse scenario is a global bus that is empty or almost unused. In this case, it might be worth to take measures to allow more inter-task communication by decreasing the weight factor β (or increasing α). More global bus utilization does not lead to higher costs as long as there is a feasible bus schedule. But a more balanced task network allows more freedom in task placement because task weights will be closer to the desired task weight. To support this process, also the maximum task weight might be decreased to avoid heavy tasks, which need powerful and expensive ECU types. Smaller tasks also allow to make better use of remaining capacities of already existing ECUs without involving a modification.

Another option would be to increase the minimum number of tasks, which decreases the aspired task weight as well, and thus the expectation value of the standard deviation. This would also lead to smaller tasks.

Strongly connected tasks Another scenario is a situation where comparatively many tasks with small weights are allocated to a single ECU. This means, that there exists one or more ECUs where the number of tasks is significantly higher than the average number of tasks per ECU. This may be an indicator for a set of strongly connected tasks with many communication channels. If this is the case, a proposed measure would be to add commanding constraints that merge all these tasks or a subset into one. Whether this is an appropriate measure depends on the resulting weight of the merged tasks, which must not violate the maximum task weight w^+. Additionally, we may add a mapping constraint assuring that these tasks are mapped to a single ECU, or to the ECU type they were mapped to before. This would help to assure that the process converges and does not propose completely different solutions.

The effect of these measures would be a reduction of task switching times between strongly connected tasks, and thus possibly less needed capacity. This leads to more capacity left on the respective processor allowing that other tasks may be placed there. Thus, the costs cannot become greater if we leave aside the fact that the global analysis is a heuristic and may find different solutions if the set (or order) of input tasks changes.

Tasks that are hard to place Tasks that are hard to place are characterized as tasks that are often part of an odd-set of different subsystems. This means that they could not be placed on these subsystems under the estimated costs limit. One reason why a task is hard to place may be a large task weight. In this case, there might exist no ECU with sufficient capacity to place this task under the estimated costs, and it would always be rejected. If the weight of this task is significantly higher than the weights of the remaining tasks, a proper measure would be to reduce the maximum task weight to force the task creation to split this task. Another reason for a refused task may be a high communication density with other tasks, which occupies the local bus. This may lead to a high response time and possibly a deadline violation. If a task could not be placed on any subsystem, this is an indicator that it may have communication channels to several tasks placed on different ECUs or subsystems.

A possible measure would be to force the task creation to split the task by forbidding the merging of function nodes that are part of that task. The basic idea is to add prohibitive constraints for those function nodes that have communication to different other tasks but are not connected with each other, neither directly nor indirectly. Thus, if there are independent clusters of nodes within the task, these clusters should be separated if they communicate with different tasks. Such constraints would forbid to create the same task again because at least one of the affected function nodes has to be partitioned into another task.

6.3. Case Study and Evaluation

To evaluate the task creation approach in the overall context of design space exploration, we created a set of task structures for the ViDAs case study with a number of tasks between one and 17. Because currently design space exploration is not able to analyze function networks with cycles, we additionally had to transform the function networks resulting from task creation to task networks that are accepted by the DSE process. Thus, we resolved all cycles, which leads to the splitting of some tasks. Obviously, this may have influence on the characteristics of the network by increasing the number of tasks and decreasing the minimum and maximum task weights. Nevertheless, those task networks are useful for an evaluation because our goal is to evaluate if task networks with differing numbers of tasks, task weights and communication density also lead to different results in design space exploration. In particular, we want to show that putting all nodes with the same sample time into one task does not lead necessarily to cost-optimal solutions. Furthermore, we also claimed that too many tasks may also lead to higher costs. Reasons for this may be too small task weights or a very high communication density between tasks which may increase the bus load.

To model global communication, if needed, we insert a signal in each resulting function network for each channel between two function nodes. Furthermore, we estimate new WCETs for each task. To enable code generation, a functional equivalent Simulink model is created where each task is represented as a single subsystem. Thus, all blocks of the original model are moved to the subsystem of their respective task, as described in the bachelor thesis of Matthias Stasch [76]. Based on this transformed Simulink model, for each subsystem, and thus each task, code can be generated and used for WCET estimation.

The existing system for this evaluation is assumed as a distributed hardware architecture with two subsystems $Sub0$ and $Sub1$ connected by a FlexRay Bus with 12 static slots. Each subsystem initially consists of two ECUs connected by a CAN bus. We defined four different ECU types consisting of two different types of ARM7 and LEON3 processors with the following costs:

- ARM7-slow: 14 EUR
- ARM7-fast: 18 EUR
- Leon3-slow: 28 EUR
- Leon3-fast: 38 EUR

Each existing ECU is of type 'ARM7-slow' and may be replaced by each available processor type. Furthermore, there may be added up to four additional ECUs from any type in each subsystem. Buses cannot be updated in this scenario. The initial hardware architecture is pre-deployed with a set of 20 tasks, which are distributed on the four existing ECUs leading to an overall utilization of 92% for the ECUs in subsystem $Sub0$ and 99% for the ECUs in subsystem $Sub1$. This means, that there is around 9% overall free capacity on the existing system *without* any modifications.

T	S	$\overline{w}$	$w\downarrow$	$w\uparrow$	$\widehat{w}$	com	$Sub0$	$Sub1$	Costs
1	0	31.2%	31.24%	31.24%	0	0	1	0	14
5	7	5.9%	0.06%	16.76%	0.152	0.042	3	2	8
6	10	4.9%	0.06%	14.28%	0.162	0.055	5	1	8
7	10	4.2%	0.16%	6.26%	0.061	0.044	7	0	4
7	12	4.2%	0.06%	14.22%	0.123	0.056	4	3	8
8	11	3.7%	0.02%	16.36%	0.057	0.043	8	0	4
9	13	3.3%	0.06%	10.96%	0.034	0.043	9	0	4
10	15	3.0%	0.06%	6.26%	0.022	0.044	8	2	8
13	28	2.3%	0.16%	4.7%	0.026	0.051	11	2	8
17	32	1.7%	0.06%	9.84%	0.136	0.052	10	5	12*

* incomplete solution

Table 6.1.: Design Space Exploration Results

For the evaluation, we estimated WCETs of the tasks for each task network and each processor type by using the aiT tool suite. In Table 6.1, the results of design space exploration are shown when applied to ten different task networks created from the ViDAs function network. Starting on the left, the table columns show the characterization of the task network in terms of number of tasks T and signals S. The next columns contain the average task weight $\overline{w}$, the minimum task weight $w\downarrow$, the maximum task weight $w\uparrow$, the standard deviation $\widehat{w}$, and the communication density com. Please note, that task weights are given in percent for better readability. The results of design space exploration are shown on the right of the table. In the columns $Sub0$ and $Sub1$ the number of tasks that were mapped to the respective subsystem is listed. The column $Costs$ shows the additional costs that had to be invested to modify the existing system to place the new task network. We do not consider the parameters of task creation in this part because their influence on the resulting task network has already been evaluated in Section 5.4. In this section, the goal is to evaluate the effects of the number of tasks and signals, the weights of tasks, and the communication density on the results of design space exploration. The task networks are listed in increasing order of the number of tasks they contain.

In general, it can be observed that, as we claimed, the resulting costs first decrease with an increasing number of tasks, and then again increase as soon as the number of tasks reaches higher values. Thus, the best results were gained for task networks between seven and nine tasks leading to the cost minimum of 4 EUR. The maximum costs are produced by the task network shown in the first line. This is the task network as it would be proposed by the Embedded Coder implementation of Simulink by putting all tasks with the same sample time into one task. Because the ViDAs model has only one sample time, this results in one task with a weight of 31.24%. Please remember, that this weight is determined by assuming the optimal processor, and thus the weight for the initially used ECU might be much higher. Because a task network with one task is perfectly balanced and has no communication, the respective values are zero. The result for this task network shows that we need to invest 14 EUR

to be able to place this heavy-weighted task. These costs result from adding a new ECU of type 'ARM7-slow' to subsystem $Sub0$ because the capacity of existing ECUs was not sufficient.

For the next networks with five and six tasks, we get costs of 8 EUR and the tasks are distributed on both subsystems while in each subsystem one ECU was upgraded from 'ARM7-slow' to 'ARM7-fast' to offer more capacity. The best results were achieved for the task networks with seven,eight and nine tasks, where all tasks could be allocated to one subsystem by only upgrading one ECU from 'ARM7-slow' to 'ARM7-fast' leading to costs of 4 EUR. Interestingly, another task network with seven tasks needed 8 EUR to be successfully placed. Possible reasons for this may be, on the one hand, the higher communication density of 5.6% compared to 4.4% resulting from two additional signals. This potentially leads to more communication on the local bus and thus larger response times. On the other hand, the standard deviation is comparatively high. Thus, there are more tasks with small weights than in the other task network with seven tasks.

The task networks with 10 and 13 tasks lead again to costs of 8 EUR and need modifications in both subsystems to allocate all tasks. Here, the higher number of tasks and the smaller average task weight might play a role leading to more task switches and less efficient processor use. Special attention has to be payed to the last network with 17 tasks and 32 signals. Here, no solution was found where all tasks could be allocated. Instead, the process returned a solution where only 15 tasks were placed with costs of 12 EUR induced by three ECU upgrades from 'ARM7-slow' to 'ARM7-fast', one in subsystem $Sub0$ and two in subsystem $Sub1$. Although this is an incomplete solution, it is still correct for all tasks that could be already mapped. The reason why the remaining tasks could not be allocated is the global bus. Due to the high number of tasks and signals, there was no free slot on the global bus but the remaining tasks could not be placed without the need for global communication between both subsystems.

The overall runtime of the design space exploration process for the different examples varies between one and 46 seconds, except for the last task network with 17 tasks. This example needs about 73 seconds because it had to consider all modifications that are allowed for each subsystem. This leads to a lot of iterations of global and local analysis steps until reaching the maximum cost limit.

6.4. Summary

We have shown how the task creation is integrated into the design space exploration process. After presenting the basic concepts of design space exploration, we discussed the role of task creation. First, it is the initial step that is needed to make the design space exploration applicable to a Simulink specification model. Second, it can be considered as another backtracking step by refining task creation parameters and constraints with the knowledge gained in the previous design space exploration steps. For the second aspect, we sketched some typical scenarios that might occur during design space exploration and how parameters and constraints may be refined to improve the result. However, this refinement highly depends on the concrete specification model

and how the existing architecture is pre-deployed by tasks and signals. Thus, these scenarios are thought as suggestions that might be offered to the user to help him in the refinement step.

To show how different task networks of the same specification model behave on a concrete architecture, we used the ViDAs case study and created a set of ten task networks with one to 17 tasks. First, it was shown that the task network with one heavy weighted task did not deliver the solution with the minimum costs. And also the task network with the highest number of 17 tasks did not deliver a cost-optimal solution due to high communication and low task weights. In this case, there was not even any solution where all tasks could be allocated because the global bus was fully utilized. Instead, the best results were achieved for task networks with between seven and nine tasks where a good trade-off between task weights and communication density was found. But there is also an example where a task network with seven tasks needed higher costs. This shows that the design space exploration is not only sensitive to the number of tasks but also to their weights and the communication between them. Thus, the effort to reason about a 'good' task network is worthwhile because it may lead to significant costs savings in the resulting architecture. These results confirm the basic claims we did for this work, which state that task weights should neither be too small nor too large, and that communication plays a significant role and should be minimized to keep bus utilization low.

Concerning future work, it might be further investigated which factors influence the needed modification costs and how these factors could be considered during task creation. Furthermore, the problem of resolving cycles may be explored in more detail to minimize the number of task splittings as long as the design space exploration is not capable of handling cycles. Additionally, it may be discussed whether and in which situations cycles, as function networks allow, are desired in general for a task network. It may be noted that we did some approaches to avoid cycles during task creation but it turned out that this increases the runtime to unacceptable values. Thus, it might be interesting to look for alternative approaches to solve this issue.

7. Conclusion

We presented an approach called *task creation* to derive a task structure from a Simulink specification model, which should be allocated to a distributed hardware architecture in a subsequent design space exploration. The architecture is assumed to be pre-deployed by a task network representing the software parts of an existing system. The overall optimization goal is to find a feasible allocation of the new task structure with a minimum of costs that occur due to hardware modifications.

The first main goal of this work was to define a translation of a Simulink model into a task network formalism where semantics in terms of partial order of block executions is maintained. Furthermore, it had to be assured that each sequence of block executions of a specific simulation step is finished before the next simulation step starts. This is needed to guarantee that all signals are updated in time. Before reaching this goal, an extended task network formalism had to be defined that is able to model the execution behavior of a Simulink model. To achieve this, the *function network* formalism was defined as a graph of tasks called function nodes connected by ports and channels. Events are produced at event sources. This formalism offers complex activation of function nodes in terms of superposition and synchronization. Furthermore, function node behavior is described by an internal transition system where each transition represents a node execution. Depending on the current state and the set of received events, different output events may be produced with different execution time delays. To also be able to model e.g. data store memory blocks of Simulink, another node type was defined called data node. To describe the occurrence of events in a function network we introduced event patterns covering the most common event models known from literature.

Semantics of function networks was defined in terms of a composition of timed automata communicating via synchronization events as known from UPPAAL. An important component of a function node is a synchronization buffer, which is used to store received events until all synchronization partners are available. To be able to implement a function network, it is necessary that the function network is bounded. This means that there exists a finite capacity for each buffer such that no buffer overflow occurs indicated by a fail state. We showed for a specific class of function networks, which is relevant for the context of this work, that boundedness is decidable by defining an algorithm based on event pattern propagation. To be able to show semantics preservation, we defined so-called causality patterns, which describe a conditional causal dependency between input and output events with a minimum and maximum time delay. This pattern enables to abstract from the concrete automaton semantics and restrict to those properties we like to preserve. Thus, we derived causal dependencies for function nodes and channels, which are used to show semantics preservation for the Simulink translation and task creation operations.

Based on function networks, a translation of Simulink models was defined, where blocks are translated to function nodes and signals to channels. For rate transition blocks between different synchronous sets, special function nodes were defined translating from one sample time to another. To keep the synchronous behavior of the Simulink model, we defined a single event source in the translated function network, which produces events with the base period. Each sample time of the model is realized by a special function node named *period multiplier*, which transforms the base period into the respective sample time. This guarantees that blocks of different synchronous sets are synchronized and run in those simulation steps they are supposed to.

We have shown that the specification semantics is preserved by defining a partial order on function network events for Simulink translations. This partial order was derived from the causal dependencies of function nodes and channels in terms of causality patterns. The function network events were associated by a mapping function to signal updates that occur due to block executions in Simulink. We proved that for each partial ordering of signal updates a partial ordering of the respective function network events exists. To show that also the timing assumptions of Simulink are preserved, we defined *feasible* block diagrams as those diagrams where in each simulation step all needed block executions are finished in time. Then, we showed for feasible block diagrams that also the execution of the translated function nodes is finished before the next simulation step starts. Thus, the function network translation does not add any delays, such as waiting times in buffers, except the negligible delays caused by the execution of period multiplier nodes. We further showed that we can decide boundedness also for the networks derived from Simulink by applying the algorithm proposed in the function networks chapter.

In the current translation we abstract from the concrete behavior of state flow blocks in terms of state flow charts. In general, it would be possible do represent such charts by the internal transition systems of function nodes. However, this has also major impacts on the translation of the remaining Simulink blocks because all conditions that are used in a state flow chart to trigger a transition need to be represented by a finite set of events in the function network. All these events have to be emitted by the respective producer nodes when the respective condition holds. Thus, we would need an abstract interpretation of all concrete signal values that influence the behavior of state charts. This would also allow to explicitly model dynamic triggers of Simulink blocks and thus get more precise estimations on the points in time where such blocks are executed. Tackling these issues may be subject to future work.

The second main goal of this work was the task creation itself, where we defined an optimization metric to iteratively merge function nodes into sets of tasks with the goal to minimize inter-task communication and balance node weights. The latter part was realized by minimizing the standard deviation with respect to a desired node weight. To represent the implications of task creation also in the formal model, a set of operations was defined to merge function nodes. Beside the merging itself we defined operations to eliminate self-activations and local data nodes. Self-activations typically arise when merging two subsequent nodes where the first one activates the second. Under specific conditions such a self-activation may be removed by concatenating the respective transitions. Local data nodes are nodes that are connected to exactly one

function node and can be removed if they do not influence the function node behavior. For all operations we have shown that they preserve the specification semantics in terms of causal dependencies on input and output events. Additionally, we showed that the relevant properties to decide boundedness are preserved and thus boundedness remains decidable also after task creation.

To perform the partitioning of function nodes into a set of tasks, a two-step algorithm was proposed. First, a start solution is produced by an initial algorithm which is improved by an extension of the Kernighan-Lin algorithm. This process is guided by the cohesion metric as optimization goal and a set of user-defined constraints restricting the minimum number of tasks and the maximum task weight. Furthermore, partitioning constraints were defined to claim or forbid the merging of two function nodes. This allows, for example, to determine the granularity of atomic nodes for partitioning. Thus, the whole process of task creation is strongly user-guided and is meant to support the user to find an appropriate system implementation.

We evaluated the task creation approach with the help of the ViDAs case study, which models a driver assistance system in Simulink. After the translation into a function network, we applied the proposed algorithms with different variations of parameters. First, it could be generally observed that the communication density was significantly decreased for each task network, which was one major goal of task creation. Furthermore, weights were balanced and in particular the minimum task weight was significantly increased to avoid tasks with very small weights. Due to the fact that function networks derived from Simulink models usually have a high amount of communication, this aspect dominates the task creation process. To evaluate the scalability of the approach, we also defined artificial benchmarks with up to 500 nodes that were analyzed within acceptable time bounds.

As a future extension, we discussed to relax the condition to eliminate self-activations to make this operation more general applicable. Even if this had effects on other channels, this would still preserve the partial order of Simulink block executions. This extension would facilitate to apply this operation also to function nodes with input ports with more than one incoming channel as it can be often observed in Simulink models. Another interesting point to investigate in future is how to determine the worst case execution time when two transitions are concatenated by the operation to eliminate self-activations. While the current approach of summing up the delays of the single transitions is only an approximation, in practice the worst case execution time might often differ from this sum e.g. because of cache effects and compiler optimizations. Thus, it would be desirable to perform a WCET analysis for each potential merging option, which is however currently not viable due to the needed effort in time and manual configuration. Hence, for the future it would be worth to investigate faster and more automated methods to better predict how execution times change when code segments are concatenated.

As a last step, we presented the integration of task creation into the design space exploration framework. After an overview of the basic concepts of the design space exploration process, we discussed the role of task creation as the initial step to make Simulink models available for the framework. Furthermore, task creation serves as an additional backtracking step to refine the created task network by the knowledge

gained from previous results. To show the impact of different task networks on the resulting allocations and in particular the costs needed to modify the existing system, we evaluated different task networks created for the ViDAs case study. We observed that the number of tasks, their weights and the amount of inter-task communication has significant impacts on the needed modification costs. As we claimed before, neither a very small nor a very high number of tasks leads to costs-minimal results. Instead, intermediate solutions where tasks can be distributed to different ECUs without inducing too much communication on local or global buses led to the best results. In the task network with the highest number of tasks, the process was even not able to allocate all tasks on the system because the global bus did not have enough capacity to transmit all needed signals in time. This shows how sensitive the whole design process is to the granularity and weight distribution of tasks and the communication density. Thus, finding a reasonable task network is a major factor to get cost-minimal results for architecture modifications.

A future extension regarding the interaction of task creation and design space exploration is the support or avoidance of cycles in function networks. Currently, cycles need to be removed before applying design space exploration and they often occur in terms of OR-loops when merging function nodes even if the original model was cycle-free. Cycles are removed by splitting those tasks where a cycle starts or ends. This increases the imbalance of task weights and the communication density between tasks and thus counteracts the actual goal of task creation. To solve this, either cycles need to be avoided by task creation at all, or the design space exploration needs to be extended to be able to handle them. Concerning the first option, we already investigated an approach to check for cycles before performing a merging operation leading to an extreme increase in runtime making it only applicable to very small function networks. Furthermore, it may be desired in general that function networks with cycles can be handled because they may already be existent in the original specification model and thus cannot always be removed.

A. Proofs for Function Networks

A.1. Proofs for Event Patterns

The following lemma is the proof of Lemma 3.1.1 from page 28.

Lemma A.1.1 (Equivalent Representation of Event Patterns) *Let EP be an event pattern with $EP = (\Sigma^{EP}, P^-, P^+, J, O)$. An equivalent representation of this event pattern is the following:*

$$\eta^+(t) = \begin{cases} \left\lfloor \frac{J}{P^-} \right\rfloor + 1 & , if\ t \in [0,\ P^- - (J \bmod P^-)) \\ n+1 & , if\ t \in [n \cdot P^- - J, (n+1) \cdot P^- - J),\ n > \left\lfloor \frac{J}{P^-} \right\rfloor \end{cases}$$

$$\eta^-(t) = \begin{cases} 0 & , if\ t\ \in [0, O + P^+ + J) \\ n & , if\ t\ \in [O + n \cdot P^+ + J,\ O + (n+1) \cdot P^+ + J),\ n > 0. \end{cases}$$

Proof:

1. *Proof for $\eta^+(t) = 1 + \left\lfloor \frac{t+J}{P^-} \right\rfloor$: Let $J = k \cdot P^- + x$ with $x \in [0, P^-)$.*

   a) Let $t \in [0,\ P^- - (J \bmod P^-))$:

$$\begin{aligned} 1 + \left\lfloor \frac{[0,\ P^- - (J \bmod P^-)) + J}{P^-} \right\rfloor &= \left\lfloor \frac{J}{P^-} \right\rfloor + 1 \\ \iff \left\lfloor \frac{[J,\ P^- - (J \bmod P^-) + J)}{P^-} \right\rfloor &= \left\lfloor \frac{J}{P^-} \right\rfloor \\ \iff \left\lfloor \frac{[J,\ P^- - ((k \cdot P^- + x) \bmod P^-) + (k \cdot P^- + x))}{P^-} \right\rfloor &= \left\lfloor \frac{J}{P^-} \right\rfloor \\ \iff \left\lfloor \frac{[J,\ P^- - x + k \cdot P^- + x)}{P^-} \right\rfloor &= \left\lfloor \frac{J}{P^-} \right\rfloor \\ \iff \left\lfloor \frac{[k \cdot P^- + x,\ (k+1) \cdot P^-)}{P^-} \right\rfloor &= \left\lfloor \frac{k \cdot P^- + x}{P^-} \right\rfloor \\ \iff k &= k \end{aligned}$$

b) Let $t \in [n \cdot P^- - J, (n+1) \cdot P^- - J),\ n > \left\lfloor \frac{J}{P^-} \right\rfloor$*:*

$$\begin{aligned} 1 + \left\lfloor \frac{[n \cdot P^- - J, (n+1) \cdot P^- - J) + J}{P^-} \right\rfloor &= 1 + n \\ \iff \left\lfloor \frac{[n \cdot P^-, (n+1) \cdot P^-)}{P^-} \right\rfloor &= n \\ \iff [n, n+1) &= n \\ \iff n &= n \end{aligned}$$

2. Proof for $\eta^-(t) = \max\left(0, \left\lfloor \frac{t-O-J}{P^+} \right\rfloor\right)$*:*

a) Let $t \in [0, O + P^+ + J)$*:*

$$\begin{aligned} \eta^-(t) &= \max\left(0, \left\lfloor \frac{t - O - J}{P^+} \right\rfloor\right) \\ &= \max\left(0, \left\lfloor \frac{[0, O + P^+ + J) - O - J}{P^+} \right\rfloor\right) \\ &= \max\left(0, \left\lfloor \frac{[-O - J, P^+)}{P^+} \right\rfloor\right) \\ &= \max\left(0, \left[\left\lfloor \frac{-O - J}{P^+} \right\rfloor, 1\right)\right) \\ &= 0 \end{aligned}$$

b) Let $t \in [O + n \cdot P^+ + J,\ O + (n+1) \cdot P^+ + J)$ *with* $n > 0$*:*

$$\begin{aligned} \eta^-(t) &= \max\left(0, \left\lfloor \frac{t - O - J}{P^+} \right\rfloor\right) \\ &= \max\left(0, \left\lfloor \frac{[O + n \cdot P^+ + J,\ O + (n+1) \cdot P^+ + J) - O - J}{P^+} \right\rfloor\right) \\ &= \max\left(0, \left\lfloor \frac{[n \cdot P^+,\ (n+1) \cdot P^+)}{P^+} \right\rfloor\right) \\ &= \lfloor [n, n+1) \rfloor \\ &= n \end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.2 from page 29.

Lemma A.1.2 (Event Pattern Language) *Let* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *be an event pattern. Then the language is defined as follows:*

$$\begin{aligned} L(EP) = \{\ & (\sigma_1,t_1)....(\sigma_i,t_i)...(\sigma_{i+m},t_{i+m})... \mid\ \sigma_i \in \Sigma^{EP}, \\ & (1)\ t_i \in [\max(0,(i-1)\cdot P^- - J), O + (i+1)\cdot P^+ + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m\cdot P^- - J), O + (m+2)\cdot P^+ + J) \\ \}\ & \textit{where } i,m \in \mathbb{N}^+ \end{aligned}$$

Proof: From Def. 3.1.1 we know:

$$\begin{aligned} L(EP) = \{\ & (\sigma_1,t_1)....(\sigma_i,t_i)...(\sigma_{i+n},t_{i+n})... \mid\ \sigma_i \in \Sigma^{EP}, \\ & (1)\ i \geq \eta^-(t_i), \\ & \quad\ i \leq \eta^+(t_i), \\ & (2)\ \forall m : \\ & \quad\ m+1 \geq \eta^-(t_{i+m} - t_i) \\ & \quad\ m+1 \leq \eta^+(t_{i+m} - t_i) \\ & \} \end{aligned}$$

First, we proof (1). From the alternative event pattern representation of Lemma 3.1.1 follows that

$$\begin{aligned} & t_i \leq [O + i\cdot P^+ + J, O + (i+1)\cdot P^+ + J) \ \wedge \\ & t_i \geq \begin{cases} [0,\ P^- - (J \bmod P^-)) & \textit{, if } i \leq \left\lfloor \frac{J}{P^-} \right\rfloor + 1 \\ [(i-1)\cdot P^- - J, i\cdot P^- - J) & \textit{, else} \end{cases} \\ \iff\ & t_i < O + (i+1)\cdot P^+ + J \ \wedge \\ & t_i \geq \max(0,(i-1)\cdot P^- - J) \\ \iff\ & t_i \in [\max(0,(i-1)\cdot P^- - J), O + (i+1)\cdot P^+ + J) \end{aligned}$$

The proof for (2) works in the same way as for (1) while i is replaced by $m+1$ leading to

$$\begin{aligned} & (2)\ \forall m :\ m+1 \geq \eta^-(t_{i+m} - t_i) \ \wedge \\ & \qquad\qquad m+1 \leq \eta^+(t_{i+m} - t_i) \\ \iff\ & (2)\ \forall m : t_{i+m} - t_i \in [m\cdot P^- - J, O + (m+2)\cdot P^+ + J] \end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.3 from page 30.

Lemma A.1.3 (Valid Translation to Jersak Model) *Let* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *be a periodic event pattern with* $P^- = P^+$ *and its Eta-curves* η^+ *and* η^-*. The translation from Def. 3.1.3 is a valid abstraction i.e., the resulting* η*-curves* $\eta^-(t)_{P+J}$ *and* $\eta^+(t)_{P+J}$ *contain all streams that the event pattern contains i.e.*

$$\eta^+(t)_{P+J} \geq \eta^+(t) \ \wedge \ \eta^-(t)_{P+J} \leq \eta^-(t)$$

Proof:

1. $\eta^+(t)_{P+J} \geq \eta^+(t)$

   a) *Case:* $t \in [0, \ P - (J \bmod P))$ $t \in [0, P - J)$

$$\begin{aligned} & \eta^+(t)_{P+J} \geq \eta^+(t) \\ \iff & \left\lceil \frac{t + Jitter}{Period} \right\rceil \geq \left\lfloor \frac{J}{P} \right\rfloor + 1 \\ \iff & \left\lceil \frac{[0, (J \bmod P)) + \epsilon + J + O}{P} \right\rceil \geq \left\lfloor \frac{J}{P} \right\rfloor + 1 \end{aligned}$$

   With $J = k \cdot P + x$ *where* $x \in [0, P)$

$$\begin{aligned} & \left\lceil \frac{[0, ((k \cdot P + x) \bmod P)) + \epsilon + k \cdot P + x + O}{P} \right\rceil \geq \left\lfloor \frac{k \cdot P + x}{P} \right\rfloor + 1 \\ \iff & \left\lceil \frac{[0, x + \epsilon + k \cdot P + x + O}{P} \right\rceil \geq k + 1 \\ \implies & k + \left\lceil \frac{x + \epsilon + x + O}{P} \right\rceil \geq k + 1 \\ \implies & \left\lceil \frac{x + \epsilon + x + O}{P} \right\rceil \geq 1 \end{aligned}$$

   b) *Case:* $t = n \cdot P - J + y$ *with* $y \in [0, P)$

$$\begin{aligned} & \eta^+(t)_{P+J} \geq \eta^+(t) \\ \iff & \left\lceil \frac{n \cdot P - J + y + \epsilon + J + O}{P} \right\rceil \geq n + 1 \\ \iff & \left\lceil \frac{n \cdot P + y + \epsilon + O}{P} \right\rceil \geq n + 1 \\ \iff & n + \left\lceil \frac{y + \epsilon + O}{P} \right\rceil \geq n + 1 \\ \iff & \left\lceil \frac{y + \epsilon + O}{P} \right\rceil \geq 1 \end{aligned}$$

2. $\eta^-(t)_{P+J} \leq \eta^-(t)$

a) Case: $t \in [0, O + P + J)$

$$\begin{aligned}
&\max\left(0, \left\lfloor \frac{t - Jitter}{Period} \right\rfloor\right) \leq 0 \\
\iff\ &\max\left(0, \left\lfloor \frac{[0, O + P + J) - Jitter}{P} \right\rfloor\right) \leq 0 \\
\Longleftarrow\ &\max\left(0, \left\lfloor \frac{O + P + J - \epsilon - \epsilon - J - O)}{P} \right\rfloor\right) \leq 0 \\
\iff\ &\max\left(0, \left\lfloor \frac{P - \epsilon - \epsilon)}{P} \right\rfloor\right) \leq 0 \\
\iff\ &\max(0, 0) \leq 0
\end{aligned}$$

b) Case: $t\ \in [O + n \cdot P + J,\ O + (n + 1) \cdot P + J)$
$\implies t = O + n \cdot P + J + y, y \in [0, P)$

$$\begin{aligned}
&\max\left(0, \left\lfloor \frac{O + n \cdot P + J + y - Jitter}{Period} \right\rfloor\right) \leq n \\
\iff\ &\max\left(0, \left\lfloor \frac{O + n \cdot P + J + y - \epsilon - J - O)}{P} \right\rfloor\right) \leq n \\
\iff\ &\max\left(0, \left\lfloor \frac{n \cdot P + y - \epsilon}{P} \right\rfloor\right) \leq n \\
\iff\ &n + \left\lfloor \frac{y - \epsilon}{P} \right\rfloor \leq n \\
&\textit{because } y \leq P - \epsilon
\end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.4 from page 31.

Lemma A.1.4 (Time Distance is Bounded) *Let* η_1^+, η_2^- *be Eta-functions of two period-equivalent event patterns* $EP_1 = (\Sigma_1^{EP}, P_1^-, P_1^+, J_1, O_1)$ *and* $EP_2 = (\Sigma_2^{EP}, P_2^-, P_2^+, J_2, O_2)$ *where* $EP_1 \overset{P}{=} EP_2$*. Then the following holds:*

$$\delta_t(\eta_1^+, \eta_2^-) \leq O_2 + 2 \cdot P + J_2 + J_1$$

Proof: By applying Def. 3.1.6 we need to show the following:

$$\forall t_1, t_2 \in \mathbb{R}_0^+ \mid \eta_1^+(t_1) = \eta_2^-(t_2) :\ |t_2 - t_1| \leq O_2 + 2 \cdot P + J_2 + J_1$$

1. *Case:* $\eta_1^+(t_1) = \eta_2^-(t_2) = \lfloor \frac{J}{P} \rfloor + 1$

$$\begin{aligned}
\eta_1^+(t_1) = \left\lfloor \frac{J_1}{P} \right\rfloor + 1 &\iff t_1 \in [0, P - (J_1 \bmod P)) \\
\wedge\, \eta_2^-(t_2) = \left\lfloor \frac{J_2}{P} \right\rfloor + 1 &\iff t_2 \in [O_2 + P + J_2, O_2 + 2 \cdot P + J_2) \\
\implies \sup(\{|t_2 - t_1|\}) &\leq O_2 + 2 \cdot P + J_2 \\
&\leq O_2 + 2 \cdot P + J_2 + J_1
\end{aligned}$$

□

2. *Case:* $\eta_1^+(t_1) = \eta_2^-(t_2) = 1 + n$ $(n > \lfloor \frac{J}{P} \rfloor)$

$$\begin{aligned}
\eta_1^+(t_1) = 1 + n &\iff t_1 \in [n \cdot P - J_1, (n+1) \cdot P - J_1) \\
\wedge\, \eta_2^-(t_2) = 1 + n &\iff t_2 \in [O_2 + (1+n) \cdot P + J_2, O_2 + (1+n+1) \cdot P + J_2) \\
\implies \sup(\{t_2 - t_1\}) &\leq O_2 + (1+n+1) \cdot P + J_2 - (n \cdot P - J_1) \\
&= O_2 + 2 \cdot P + J_2 + J_1
\end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.6 from page 32.

Lemma A.1.5 (Infinite Distance) *Let η_1^+, η_2^- be Eta-functions of two periodic event patterns $EP_1 = (\Sigma_1^{EP}, P_1, P_1, J_1, O_1)$ and $EP_2 = (\Sigma_2^{EP}, P_2, P_2, J_2, O_2)$ that are not period-equivalent i.e., $P_1 = \frac{q}{r} \cdot P_2$ where $q, r \in \mathbb{R}^+$ and $q \neq r$. The time distance until the same number of events is produced grows with every periodic step and is thus not bounded i.e. $\delta_t(\eta_1^+, \eta_2^-) = \infty$ leading with Def. 3.1.6 to*

$$\lim_{t_1, t_2 \to \infty} \left(\sup_{t_1, t_2 \in \mathbb{R}_0^+} (|t_2 - t_1| \;\; | \; \eta_1^+(t_1) = \eta_2^-(t_2)) \right) = \infty$$

Proof: From $\eta_1^+(t_1) = \eta_2^-(t_2)$ it follows for $t_1, t_2 \to \infty$:

$$\begin{aligned}
&\eta_1^+(t_1) = \eta_2^-(t_2) = 1 + n_1 \\
\implies\; & t_1 \in [n_1 \cdot P_1 - J_1, (n_1 + 1) \cdot P_1 - J_1) \\
\implies\; & t_2 \in [O_2 + (1 + n_1) \cdot P_2 + J_2, O_2 + (1 + n_1 + 1) \cdot P_2 + J_2)
\end{aligned}$$

We first determine the maximum difference as follows:

$$\begin{aligned}
&\sup_{t_1, t_2 \in \mathbb{R}_0^+} (|t_2 - t_1| \mid \eta_1^+(t_1) = \eta_2^-(t_2)) \\
=\;& |O_2 + (2 + n_1) \cdot P_2 + J_2 - n_1 \cdot P_1 + J_1| \\
=\;& |O_2 + J_1 + J_2 + (2 + n_1) \cdot P_2 - n_1 \cdot \frac{q}{r} \cdot P_2| \\
=\;& |O_2 + J_1 + J_2 + (2 + n_1 \cdot (1 - \frac{q}{r})) \cdot P_2|
\end{aligned}$$

Then regarding the limit leads to the statement we want to prove:

$$\lim_{t_1,t_2\to\infty}\left(\sup_{t_1,t_2\in\mathbb{R}_0^+}(|t_2-t_1| \mid \eta_1^+(t_1)=\eta_2^-(t_2))\right)$$
$$=\lim_{n_1\to\infty}\left(\left|O_2+J_1+J_2+(2+n_1\cdot(1-\frac{q}{r}))\cdot P_2\right|\right)=\infty$$

because $\frac{q}{r}\neq 1$ □

The following lemma is the proof of Lemma 3.1.7 from page 33.

Lemma A.1.6 (Synchronization is Correct Abstraction) *Let* $EP_1...EP_n$ *be* n *period-equal event patterns with* $EP_i=(\Sigma_i^{EP},P_i^-,P_i^+,J_i,O_i)$ *and the respective Eta-curves* $\eta_i^{-/+}$ *where* $\forall i,j\in\{1,...,n\}: EP_i\overset{P}{=}EP_j$. *Let* $EP_s=sync(EP_1,...,EP_n)$ *be the synchronization of these event patterns with the Eta-curves* $\eta_s^{-/+}$. *Then the following holds:*

$$\forall i\in\{1,...,n\},t\in\mathbb{R}_0^+:\ \eta_s^-(t)\leq\eta_i^-(t)\ \wedge\ \eta_s^+(t)\geq\eta_i^+(t)$$

Proof:

1. $\forall i\in\{1,...,n\},t\in\mathbb{R}_0^+:\eta_s^-(t)\leq\eta_i^-(t)$

$$\begin{aligned}\max\left(0,\left\lfloor\frac{t-O_s-J_s}{P}\right\rfloor\right)\ &\leq\ \max\left(0,\left\lfloor\frac{t-O_i-J_i}{P}\right\rfloor\right)\\ \iff t-O_s-J_s\ &\leq\ t-O_i-J_i\\ \iff O_i+J_i\ &\leq\ O_s+J_s\\ \iff O_i+J_i\ &\leq\ \max(O_1,...,O_n)+\max(J_1,...,J_n)\end{aligned}$$

2. $\forall i\in\{1,...,n\},t\in\mathbb{R}_0^+:\eta_s^+(t)\geq\eta_i^+(t)$

$$\begin{aligned}1+\left\lfloor\frac{t+J_s}{P}\right\rfloor\ &\geq\ 1+\left\lfloor\frac{t+J_i}{P}\right\rfloor\\ \iff J_s\ &\geq\ J_i\\ \iff \max(J_1,...,J_n)\ &\geq\ J_i\end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.8 from page 34.

Lemma A.1.7 (Superposition is Valid Abstraction) *Let* EP_1,EP_2 *be two event patterns with* $EP_1=(\Sigma_1^{EP},P_1^-,P_1^+,J_1,O_1)$ *and* $EP_2=(\Sigma_2^{EP},P_2^-,P_2^+,J_2,O_2)$ *where* $J_1=J_2=0$ *and the respective lower Eta-curves* η_1^- *and* η_2^-. *Let further be* $EP_s=$

super(EP_1, EP_2) *be the superposition of both event patterns with the lower Eta-curve* η_s^-*. Then the superposition offset is a valid abstraction because the following holds:*

$$\forall t \in \mathbb{R}_0^+ : \eta_s^-(t) \leq \eta_1^-(t) + \eta_2^-(t)$$

Proof: Let $t = O_s + n \cdot P_s^+ + y$*,* $y \in [0, P_s^+]$*,* $O_s = O_1 + x_1 = O_2 + x_2$ *with* $x_1, x_2 > 0$

$$\begin{aligned}
\eta_s^-(O_s + n \cdot P_s^+ + y) &\leq \eta_1^-(O_s + n \cdot P_s^+ + y) + \eta_2^-(O_s + n \cdot P_s^+ + y) \\
\iff\ & n \leq \eta_1^-(\max(O_1, O_2) + n \cdot \frac{1}{\frac{1}{P_1^+} + \frac{1}{P_2^+}} + y) \\
& \quad + \eta_2^-(\max(O_1, O_2) + n \cdot \frac{1}{\frac{1}{P_1^+} + \frac{1}{P_2^+}} + y) \\
\iff\ & n \leq \eta_1^-(\max(O_1, O_2) + n \cdot \frac{1}{\frac{P_1^+ + P_2^+}{P_1^+ \cdot P_2^+}} + y) \\
& \quad + \eta_2^-(\max(O_1, O_2) + n \cdot \frac{1}{\frac{P_1^+ + P_2^+}{P_1^+ \cdot P_2^+}} + y) \\
\iff\ & n \leq \eta_1^-(\max(O_1, O_2) + n \cdot \frac{P_2^+}{P_1^+ + P_2^+} \cdot P_1^+ + y) \\
& \quad + \eta_2^-(\max(O_1, O_2) + n \cdot \frac{P_1^+}{P_1^+ + P_2^+} \cdot P_2^+ + y) \\
\iff\ & n \leq \eta_1^-(O_1 + x_1 + n \cdot \frac{P_2^+}{P_1^+ + P_2^+} \cdot P_1^+ + y) \\
& \quad + \eta_2^-(O_2 + x_2 + n \cdot \frac{P_1^+}{P_1^+ + P_2^+} \cdot P_2^+ + y) \\
\overset{Monotony}{\Longrightarrow}\ & n \leq n \cdot \frac{P_2^+}{P_1^+ + P_2^+} + n \cdot \frac{P_1^+}{P_1^+ + P_2^+} \\
\iff\ & n \leq n \cdot \frac{P_1^+ + P_2^+}{P_1^+ + P_2^+} \\
\iff\ & n \leq n
\end{aligned}$$

□

The following lemma is the proof of Lemma 3.1.9 from page 35.

Lemma A.1.8 (Correct Event Pattern for Delayed Language) *Let* Σ^{EP} *be a set of events with an event pattern* $EP(\Sigma^{EP}) = (\Sigma^{EP}, P^-, P^+, J, O)$*. If each event* $e \in \Sigma^{EP}$ *is delayed by a time interval of* $[min, max]$ *with* $min, max \in \mathbb{N}_0 \times \mathbb{N}_0$ *and*

$min \leq max$, then the resulting language $L(\Sigma^{EP})'$ can be abstracted by applying the delay function. Following Def. 3.1.1, the delayed language $L(\Sigma^{EP})'$ is defined as

$$\begin{aligned}
L(\Sigma^{EP})' = \{\ & (\sigma_1, t_1 + [min, max])....(\sigma_i, t_i + [min, max])... \\
& (\sigma_{i+m}, t_{i+m} + [min, max])... \mid\ \sigma_i \in \Sigma^{EP}, \\
& (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\
\} \quad & \textit{where } i, m \in \mathbb{N}^+
\end{aligned}$$

Then it holds

$$L(\Sigma^{EP})' \subseteq L(delay(EP(\Sigma^{EP}), [min, max]))$$

Proof: First, we can transform the language $L(\Sigma^{EP})'$ as follows:

$$\begin{aligned}
L(\Sigma^{EP})' = \{\ & (\sigma_1, t_1 + [min, max])....(\sigma_i, t_i + [min, max])... \\
& (\sigma_{i+m}, t_{i+m} + [min, max])... \mid\ \sigma_i \in \Sigma^{EP}, \\
& (1)\ t_i \in [\max(min, (i-1) \cdot P^- - J + min), O + (i+1) \cdot P^+ + J + max) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J - (max - min)), \\
& \qquad O + (m+2) \cdot P^+ + J + max - min) \\
\} \quad & \textit{where } i, m \in \mathbb{N}^+
\end{aligned}$$

Now, we show language inclusion by determining the language of the event pattern resulting from $delay(EP(\Sigma^{EP}))$. Please remember that the delay function changes the jitter to $J' = J + max - min$ and the offset to $O' = O + min$.

$$\begin{aligned}
L(\Sigma^{EP})' = \{\ & (\sigma_1, t_1)...(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma^{EP}, \\
& (1)\ t_i \in [\max(0, (i-1) \cdot P^- - (J + max - min)), \\
& \qquad O + min + (i+1) \cdot P^+ + J + max - min) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - (J + max - min)), \\
& \qquad O + min + (m+2) \cdot P^+ + J + max - min) \\
\} \quad & \textit{where } i, m \in \mathbb{N}^+ \\
\iff & \\
L(\Sigma^{EP})' = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma^{EP}, \\
& (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J - (max - min)), \\
& \qquad O + (i+1) \cdot P^+ + J + max) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J - (max - min)), \\
& \qquad O + (m+2) \cdot P^+ + J + max) \\
\} \quad & \textit{where } i, m \in \mathbb{N}^+
\end{aligned}$$

This results in the following unequations to be satisfied:

1.

$$\begin{aligned}\max(min, (i-1)\cdot P^- - J + min) &\geq \max(0, (i-1)\cdot P^- - J - (max - min)) \\ \iff\ min \geq 0\ &\wedge\ min \geq -(max - min) = min - max\end{aligned}$$

2.

$$O + i \cdot P + J + max \leq O + i \cdot P + J + max$$

3.

$$\begin{aligned}\max(min, m\cdot P^- - J - (max - min)) &\geq \max(0, m\cdot P^- - J - (max - min)) \\ \iff\ min &\geq 0\end{aligned}$$

4.

$$O + (m+1)\cdot P + J + max \leq O + (m+1)\cdot P + J + max$$

□

A.2. Proofs for Function Network Semantics

The following lemma is the proof of Lemma 3.3.1 from page 55.

Lemma A.2.1 (Transitivity of Causality)

$$\begin{aligned}&(1)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\}[cond_2]\ \wedge \\ &(2)\ \{f_1, ..., f_m\}[cond_2] \xrightarrow{[min_2, max_2]} \{g_1, ..., g_r\}[cond_3] \\ \implies\ &(3)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1+min_2, max_1+max_2]} \{g_1, ..., g_r\}[cond_3]\end{aligned}$$

Proof: What we need to proof is that if the automata of the first two patterns (1) and (2) never reach the 'fail' state, also the automaton of the third pattern (3) never reaches the 'fail' state. (1) and (3) consume simultaneously all input events e_1 to e_n in an arbitrary order because both patterns have the same input events. When all input events have been received, they both reach the state 'check'. Because the condition $cond_1$ is also identical, either both automata proceed to the state 'wait' or return to the state 'init'. Hence, the clock is also reset at the same time with the transition $(check \xrightarrow{[cond_1]/\{c\}} wait)$*. Because we know that (1) holds, we know that between min_1 and max_1 time units all the events $\{f_1, ..., f_m\}$ occur in an arbitrary order and the condition $cond_2$ holds. Then, the automaton of (1) returns to the state 'init'. As soon as all events $\{f_1, ..., f_m\}$ have occurred, the automaton of (2) reaches its 'check' state, and we know from automaton (1) that $cond_2$ holds at the time where the events $\{f_1, ..., f_m\}$ have occurred. Thus, the automaton of (2) proceeds to its 'wait' state.*

From (2) we know, that after an additional time delay between min_2 and max_2 time units, the events $\{g_1, ..., g_r\}$ occur and the condition $cond_3$ holds. To summarize, the events $\{g_1, ..., g_r\}$ and the condition $cond_3$ occur between min_1+min_2 and max_1+max_2 time units after $\{e_1, ..., e_n\}$ have occurred under the condition $cond_1$, which is exactly what (3) states. Thus, (3) never reaches its fail state. □

The following lemma is the proof of Lemma 3.3.2 from page 55.

Lemma A.2.2 (Transitivity of Causality with External Conditions)

$$\begin{aligned}
&(1)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\} \ \wedge \\
&(2)\ \{f_1, ..., f_m\}[cond_2] \xrightarrow{[min_2, max_2]} \{g_1, ..., g_r\} \ \wedge \\
&(3)\ \forall i \in \{1, ..., n\}, j \in \{1, ..., m\} :\ [cond_2] \textit{ holds during } [e_i, f_j] \\
\Longrightarrow\ &(4)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1+min_2, max_1+max_2]} \{g_1, ..., g_r\}
\end{aligned}$$

Proof: As a first step, we use (3) to extend (1) to the statement (1a) as follows:

$$\{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\}[cond_2]\ (1a).$$

We can do this because we know from (3) that $cond_2$ holds as soon as an e_i has occurred and until an f_j has occurred. By replacing (1) by (1a), we can omit (3) and finally get the same statement as already proven in Lemma 3.3.1:

$$\begin{aligned}
&(1a)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\}[cond_2] \ \wedge \\
&(2)\ \{f_1, ..., f_m\}[cond_2] \xrightarrow{[min_2, max_2]} \{g_1, ..., g_r\} \\
\Longrightarrow\ &(4)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1+min_2, max_1+max_2]} \{g_1, ..., g_r\}
\end{aligned}$$

□

The following lemma is the proof of Lemma 3.3.3 from page 56.

Lemma A.2.3 (Combination of Conditions in Causality Pattern)

$$\begin{aligned}
&(1)\ \{e_1, ..., e_n\}[cond_1] \xrightarrow{[min_1, max_1]} \{f_1, ..., f_m\} \ \wedge\ ... \ \wedge \\
&\quad \{e_1, ..., e_n\}[cond_k] \xrightarrow{[max_k, min_k]} \{f_1, ..., f_m\} \\
\Longrightarrow\ &(2)\ \{e_1, ..., e_n\}[cond_1 \vee ... \vee cond_k] \xrightarrow{[min', max']} \{f_1, ..., f_m\} \\
&\quad \textit{where } min' = \min(min_1, ..., min_k),\ max' = \max(max_1, ..., max_k)
\end{aligned}$$

Proof: All the patterns of (1) wait for the events $e_1, ..., e_n$ to occur. Then, for all conditions $cond_1$ to $cond_k$ it holds that the output events $f_1, ..., f_m$ occur after a delay that may vary between $min' = \min(min_1, ..., min_k)$ and $max' = \max(max_1, ..., max_k)$ time units. This means, that it is sufficient that only one of these conditions is true to see the events $f_1, ..., f_m$ within a time interval of $[min', max']$, which is exactly what (2) states. □

The following lemma is the proof of Lemma 3.3.4 from page 59.

Lemma A.2.4 (Event Pattern of Event Source) *Let $\phi = (EP, \mathcal{P}^{out}) \in \Phi$ be an event source where $EP = (\Sigma^{EP}, P^-, P^+, J, O)$. Then the event pattern of each output port $p_j^{out} \in \mathcal{P}^{out}$ is a valid abstraction of the respective language i.e.*

$$\forall p_j^{out} \in \mathcal{P}^{out} : \; L(p_j^{out}) \subseteq L(ren(EP, \{p_j^{out}.\sigma \mid \; \sigma \in \Sigma(p_j^{out})\}))$$

Proof: $L(ren(EP, \{p_j^{out}.\sigma \mid \; \sigma \in \Sigma(p_j^{out})\}))$ is defined as follows:

$$\begin{aligned} L(EP) = \{ \; & (p_j^{out}.\sigma_1, t_1)....(p_j^{out}.\sigma_i, t_i)...(p_j^{out}.\sigma_{i+n}, t_{i+n})... \mid \; \sigma_i \in \Sigma(p_j^{out}), \\ & (1) \; t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\ & (2) \; \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\ & \} \quad \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

1. *We start with property (1) and the case $i = 1$. For $L(p_j^{out})$ we know*

$$\begin{aligned} & O \leq t_1 \leq O + P^+, \\ & 0 \leq \delta_i \leq J \\ \iff & t_1 \in [O, O + P^+], \\ & \delta_1 \in [0, J] \\ \iff & (t_1 + \delta_1) \in [O, O + P^+ + J] \\ \implies & (1) \; (t_1 + \delta_1) \in [\max(0, (1-1) \cdot P^- - J), O + (1+1) \cdot P^+ + J) \\ \iff & (1) \; (t_1 + \delta_1) \in [0, O + 2 \cdot P^+ + J) \end{aligned}$$

2. *Now, we show property (1) for $i \geq 1$. For $L(p_j^{out})$ we know*

$$\begin{aligned} & t_i + P^- \leq t_{i+1} \leq t_i + P^+, \\ & 0 \leq \delta_{i+1} \leq J \\ \implies & t_1 + (i-1) \cdot P^- + P^- \leq t_{i+1} \leq t_1 + (i-1) \cdot P^+ + P^+, \\ & 0 \leq \delta_{i+1} \leq J \\ \iff & t_1 + i \cdot P^- \leq t_{i+1} + \delta_{i+1} \leq t_1 + i \cdot P^+ + J \\ \iff & [O, O + P^+] + i \cdot P^- \leq t_{i+1} + \delta_{i+1} \leq [O, O + P^+] + i \cdot P^+ + J \\ \iff & O + i \cdot P^- \leq t_{i+1} + \delta_{i+1} \leq O + P^+ + i \cdot P^+ + J \\ \iff & (t_{i+1} + \delta_{i+1}) \in [O + i \cdot P^-, \; O + (i+1) \cdot P^+] + J] \end{aligned}$$

From this statement, we can conclude property (1)

$$\begin{aligned} \implies & (t_{i+1} + \delta_{i+1}) \in [\max(0, (i+1-1) \cdot P^- - J), O + (i+1+1) \cdot P^+ + J) \\ \iff & (t_{i+1} + \delta_{i+1}) \in [i \cdot P^- - J), O + (i+2) \cdot P^+ + J) \end{aligned}$$

3. *The next step is to show that property (2) holds:*
   For $L(p_j^{out})$ *we know*

$$\begin{aligned} & \forall m : t_i + m \cdot P^- \leq t_{i+m} \leq t_i + m \cdot P^+, \\ & 0 \leq \delta_i \leq J \\ \implies & \forall m : (t_{i+m} - t_i) \in [m \cdot P^-, m \cdot P^+], \\ & 0 \leq \delta_i \leq J \\ \iff & \forall m : ((t_{i+m} + \delta_{i+m}) - (t_i + \delta_i)) \in [m \cdot P^- - J, m \cdot P^+ + J] \end{aligned}$$

   From this statement, we can conclude property (2) i.e. $\forall m :$

$$\begin{aligned} \implies & ((t_{i+m} + \delta_{i+m}) - (t_i + \delta_i)) \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\ \iff & ((t_{i+m} + \delta_{i+m}) - (t_i + \delta_i)) \in [m \cdot P^- - J, O + (m+2) \cdot P^+ + J) \end{aligned}$$

□

The following lemma is the proof of Corollary 3.3.5 from page 81.

Corollary A.2.1 (Causality of Finite Source Data Node) *According to the extended function network definition from Def. 3.2.3, a finite source data node* $d = (\{p^{in}\}, \delta, EP, \{p^{out}\}) \in \mathcal{D}_{fsource}$ *with* $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *is translated into a function node* $f_d = (\{p^{in}, p_d^{tr}\}, (S, s_0, T), \{p^{out}, p_d^{\perp}\})$*. This leads to the following causal dependency if the assumption holds that the delay between an output and input event is bounded by* $P^- - J$*:*

$$\begin{aligned} & \Delta(L(\Sigma^{EP}), L(\Sigma(p^{in}))) \leq P^- - J \\ \implies & \forall e \in \Sigma^{EP} : \; p_d^{tr}.e \xrightarrow{\delta} p^{out}.e \end{aligned}$$

Proof: From the definition of the output language of an event source (Def. 3.3.4) follows that the minimum distance between two succeeding events is $\Delta^-(L(\Sigma^{EP})) = P^- - J$ *(while we know that for source nodes* $J < P^-$ *holds). The transition system of* f_d *starts in the 'ready' state, where it produces an event* e *at* p^{out} δ *time units after it has received this event at* p_d^{tr}*, which is the statement to prove. Then it moves to the 'wait' state, where it waits for an input event* $a \in \Sigma(p^{in})$ *to arrive. From the assumption we know that* $\Delta(L(\Sigma^{EP}), L(\Sigma(p^{in}))) \leq P^- - J$ *holds. This means, that we will always see an input event* $a \in \Sigma(p^{in})$ *before the next source event* $e \in \Sigma^{EP}$ *has arrived, and thus the transition system is always in the 'wait' state if an event* e *arrives. This concludes the proof.* □

A.3. Proofs for Boundedness and Event Pattern Propagation

The following lemma is the proof of Corollary 3.4.1 from page 87.

Corollary A.3.1 (Translating from RTC to Event Pattern) *A RTC function*

$$\alpha^u(\Delta) := N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor$$
$$\alpha^l(\Delta) := N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor$$

can be represented by an event pattern $EP = (\Sigma^{EP}, P^-, P^+, J, O)$ *where*

- $J = (N^u - 1) \cdot P^-$
- $O = -N^l \cdot P^+$

Proof:

1.

$$\begin{aligned} & \eta^+(\Delta) \geq \alpha^u(\Delta) \\ \iff & 1 + \left\lfloor \frac{\Delta + J}{P^-} \right\rfloor \geq N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor \\ \iff & 1 + \left\lfloor \frac{\Delta + (N^u - 1) \cdot P^-}{P^-} \right\rfloor \geq N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor \\ \iff & 1 + N^u - 1 + \left\lfloor \frac{\Delta}{P^-} \right\rfloor \geq N^u + \left\lfloor \frac{\Delta}{P^-} \right\rfloor \end{aligned}$$

2.

$$\begin{aligned} & \eta^-(\Delta) \leq \alpha^l(\Delta) \text{ for } \alpha^l(\Delta) \geq 0 \\ \iff & \left\lfloor \frac{\Delta - O - J}{P^+} \right\rfloor \leq N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor \\ \iff & \left\lfloor \frac{\Delta + N^l \cdot P^+ - J}{P^+} \right\rfloor \leq N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor \\ \iff & N^l + \left\lfloor \frac{\Delta - J}{P^+} \right\rfloor \leq N^l + \left\lfloor \frac{\Delta}{P^+} \right\rfloor \end{aligned}$$

because $J \geq 0$

□

The following lemma is the proof of Lemma 3.4.7 from page 92.

Lemma A.3.1 (Deciding Boundedness of Activation Buffers) *Let* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ *be a function node with* $\mathcal{A} = (S, s_0, T)$*. Let further be*

- R *be the set of all partial runs of* $\mathcal{A}$ *where* $r = (s_1 \xrightarrow{t_1} s_2 ... \xrightarrow{t_k} s_{k+1}) \in R$ *is a partial run of the transition system of length* k *with* $s_i \in S$, $t_i = (p_i, E_i, s_i \to \Psi_i, s_{i+1}) \in T$,
- $\delta^{max}(r) = \sum_{i=1}^{k} (\delta_i^{max})$ *denote the maximum delay of a run* r *where* δ_i^{max} *denotes the maximum delay of a transition* t_i *as introduced in Lemma 3.3.12,*
- $act = Sync(I_p, \{start_f\}, c)$ *be the activation buffer of* f*,*
- $EP(I_p) = (\Sigma^{EP}, P, P, J, O)$ *be the periodic event pattern over* I_p*.*

Then it holds that act is bounded if each cyclic partial run of length k *has a length smaller than* $k \cdot P$ *i.e.:*

$$act \text{ is bounded} \iff \forall r = (s_1 \xrightarrow{t_1} s_2 ... \xrightarrow{t_k} s_1) \in R \ : \ \delta(r) \leq k \cdot P$$

Proof: For a synchronization buffer to be bounded, all input streams must have the same period (see Lemma 3.4.5). The activation buffer has two input streams, namely all input events of the function node that are known to arrive with the event pattern $EP(I_p)$ *with period* P*, and start events whose language has been determined in Lemma 3.4.6 to be*

$$\begin{aligned} L(start_f) = \{ \ & (start_f, u_1), ..., (start_f, u_i), ..., (start_f, u_{i+m})\} \\ | \ & u_1 = 0, \\ & u_{i+1} = \max(u_i, t_i) + \delta, \\ & \delta \in [\delta^{min}, \delta^{max}] \end{aligned}$$

Thus, for the period of start events it holds $P_{start} \geq P$ *because it always needs an input event to see the next* $start_f$ *event.*

1. $\Longrightarrow$ *(Proof by contraposition):*
   If we assume that there exists a partial cyclic run r *of length* k *with a delay* $\delta(r) > k \cdot P$*, then this partial run can be concatenated to an infinite run leading to* $P_{start} > P$*. Together with Lemma 3.4.5, it follows that act is unbounded.*
2. $\Longleftarrow$*:*
   If for each partial run of length k *holds* $\delta(r) \leq k \cdot P$*, then it follows for the period of* $start_f$ *events that* $P_{start} \leq P$*. Because we know that* $P_{start} \geq P$ *holds, it follows that* $P_{start} = P$ *and act is bounded.*

□

The following lemma is the proof of Lemma 3.4.9 from page 94.

Lemma A.3.2 (Cyclic Causal Dependency leads to Unboundedness) *Let bfn be a reachable function network with* $bfn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})$ *and* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ *be a state-independent function node with an input port* $p^{in} \in \mathcal{P}^{in}$ *and a output port* $p^{out} \in \mathcal{P}^{out}$ *and a cyclic causal dependency between* p^{in} *and* p^{out}*. Let further be sync the synchronization buffer of* p^{in} *and act the activation buffer of* f*. Then it holds that at least sync is unbounded or act is unbounded i.e.*

$$\begin{aligned} & \exists e_{in} \in \Sigma(p_{in}),\ e_{out} \in \Sigma(p_{out}) : \\ & p^{in}.e_{in} \xrightarrow{[min,max]} p^{out}.e_{out} \ \wedge\ p^{out}.e_{out} \xrightarrow{[min',max']} p^{in}.e_{in} \\ \implies & \textit{act is unbounded} \ \vee\ \textit{sync is unbounded} \end{aligned}$$

Proof:

1. *OR loop: If* p^{in} *has only one incoming channel, this must be the one which causes the cyclic causal dependency, and it must be an OR loop because no synchronization is needed for one incoming channel. For one input channel, the buffer sync is always bounded. Due to the causal dependency from* p^{in} *to* p^{out}*, we know that the event pattern of* p^{out} *depends on* p^{in}*. If we nevertheless try to determine the event pattern of* p^{out} *while ignoring* p^{in}*, and propagate the event pattern to* p^{in}*, then an event pattern* $EP'(p^{out})$ *would be determined as superposition of* $EP(p^{out})$ *with itself leading with Lemma 3.4.2 to*

$$EP'(p^{out}) = super(EP(p^{out}), EP(p^{out})).$$

   This event pattern would again be propagated to p^{in} *and then to* p^{out} *leading to* $EP''(p^{out}) = super(EP'(p^{out}), EP'(p^{out}))$ *and so on. Thus, with each propagation pass, the period of* $EP(p^{out})$ *is halved (see Lemma 3.4.8) leading eventually to a violation of the boundedness condition of the activation buffer* $\delta^{max} \leq P$ *because* $\delta^{max} > 0$*. Thus, the activation buffer act is unbounded.*

2. *AND loop: If there is more than one incoming channel at* p^{in}*, we have a synchronization loop, and there will never occur a synchronization event at* p_{out} *because this needs an event to occur at* p^{in} *before. This would lead to an event pattern for* p_{out} *with an infinite upper period bound. Because such an event pattern cannot be period-equal to any event pattern, this leads with Lemma 3.4.5 to an unbounded synchronization buffer.*

□

B. Proofs for Simulink Translation and Preserving Semantics

B.1. Proofs for Translation

The following lemma is the proof of Lemma 4.2.1 from page 118.

Lemma B.1.1 (Period Multiplier - Output Port Language) *Let* $f^{Mult}(k, off, \{o_1, ..., o_m\}, [\delta^-, \delta^+]) = (\{p^{in}\}, \mathcal{A}, \{p_1^{out}, ..., p_m^{out}, p^\perp\})$ *be a period multiplier function node and* $EP(\Sigma(p^{in})) = (\Sigma(p^{in}), P, P, J, O)$ *be the event pattern of the input port leading with Lemma 3.1.2 to the timed language*

$$\begin{aligned} L(EP(\Sigma(p^{in}))) = \{\ & (p^{in}.\sigma_1, t_1)...(p^{in}.\sigma_i, t_i)...(p^{in}.\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma(p^{in}), \\ & (1)\ t_i \in [\max(0, (i-1)\cdot P - J), O + (i+1)\cdot P + J) \\ & (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m\cdot P - J), O + (m+2)\cdot P + J) \\ \}\ & \textit{where } i, m \in \mathbb{N}^+ \end{aligned}$$

Then the language for each output port p_j^{out} *with* $j \in \{1, ..., m\}$ *is defined as follows:*

$$L(p_{o_j}) = \{(p_{o_j}.o_j, r_1 + [\delta^-, \delta^+])...(p_{o_j}.o_j, r_i + [\delta^-, \delta^+])...\} \quad \textit{where } r_i = t_{1+off+(i-1)\cdot k}$$

Proof: When the first input event arrives, f *is in its initial state* s_{k-off}*. An output event* $p_{o_j}.o_j$ *is only produced in state* s_k *i.e. it takes* off *input events and thus* $off \cdot P$ *time units from the first input event at* t_1 *until we see the first output event at an output port* p_{o_j}*. Additionally, we have to consider the transition delay, which is defined to be* $[\delta^-, \delta^+]$ *leading to* $off \cdot P + [\delta^-, \delta^+]$*. Thus, the first event occurs at* $t_{1+off} + [\delta^-, \delta^+]$*, which we denote as* $r_1 + [\delta^-, \delta^+]$ *with* $r_1 = t_{1+off}$*. After the first output event has been emitted, we are in state* s_1*, and thus it takes* k *input events, which are* $k \cdot P$ *time units, until the next output event is produced. Thus, the* i^{th} *output event (*$i > 0$*) occurs at time* $t_{1+off+(i-1)\cdot k} + [\delta^-, \delta^+]$*, which we denote as* $r_i + [\delta^-, \delta^+]$ *with* $r_i = t_{1+off+(i-1)\cdot k}$*.* □

The following lemma is the proof of Lemma 4.2.2 from page 119.

Lemma B.1.2 (Period Multiplier - Output Event Pattern) *Let* $f^{Mult}(k, off, \{o_1, ..., o_m\}, [\delta^-, \delta^+]) = (\{p^{in}\}, \mathcal{A}, \{p_{o_1}, ..., p_{o_m}, p^\perp\})$ *be a period multiplier function node and* $EP = (\Sigma^{EP}, P, P, J, O)$ *be the event pattern of the input port. Then the following holds:*

$$\forall j \in \{1, ..., m\} : L(p_{o_j}) \subseteq L(delay((\{p_{o_j}.o_j\}, P \cdot k, P \cdot k, J, O + off \cdot P),\ [\delta^-, \delta^+]))$$

Proof: Given

$$
\begin{aligned}
L(EP(\Sigma(p^{in}))) = \{\ & (p^{in}.\sigma_1, t_1)...(p^{in}.\sigma_i, t_i)...(p^{in}.\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in \Sigma(p^{in}), \\
& (1)\ t_i \in [\max(0, (i-1)\cdot P - J), O + (i+1)\cdot P + J) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m\cdot P - J), O + (m+2)\cdot P + J) \\
& \}\ \ \textit{where } i, m \in \mathbb{N}^+
\end{aligned}
$$

we can conclude from Lemma 4.2.1 that

$$
\begin{aligned}
& L(p_{o_j}) = \{(p_{o_j}.o_j, r_1 + [\delta^-, \delta^+])...(p_{o_j}.o_j, r_i + [\delta^-, \delta^+])...\} \\
& \textit{where } r_i = t_{1+off+(i-1)\cdot k} \\
\iff & \\
& L(p_{o_j}) = \{\ (p_{o_j}.o_j, r_1 + [\delta^-, \delta^+])...(p_{o_j}.o_j, r_i + [\delta^-, \delta^+])... \mid \\
& (1)\ r_i \in [\max(0, (1 + off + (i-1)\cdot k - 1)\cdot P - J), \\
& O + (1 + off + (i-1)\cdot k + 1)\cdot P + J) \\
& (2)\ \forall m : r_{i+m} - r_i \in [\max(0, m\cdot k\cdot P - J), O + (m\cdot k + 2)\cdot P + J)\ \}
\end{aligned}
$$

Together with Lemma 3.1.9, where the correctness of the delay function for event pattern has been shown, the language of $EP(p_{o_j})$ is determined as follows:

$$
\begin{aligned}
L(EP(p_{o_j})) = \{\ & (p_{o_j}.o_j, r_1 + [\delta^-, \delta^+])....(p_{o_j}.o_j, r_i + [\delta^-, \delta^+])... \mid \\
& (1)\ r_i \in [\max(0, (i-1)\cdot (P\cdot k) - J), \\
& \quad O + off\cdot P + (i+1)\cdot (P\cdot k) + J) \\
& (2)\ \forall m : r_{i+m} - r_i \in [\max(0, m\cdot (P\cdot k) - J), \\
& \quad O + off\cdot P + (m+2)\cdot (P\cdot k) + J)\ \}
\end{aligned}
$$

To show that $L(p_{o_j}) \subseteq L(EP(p_{o_j}))$, it remains to prove that all interval bounds induced by (1) and (2) of $L(EP(p_{o_j}))$ are valid abstractions of those from $L(p_{o_j})$ leading to the following cases:

1. $\max(0, (1 + off + (i-1)\cdot k - 1)\cdot P - J \geq \max(0, (i-1)\cdot (P\cdot k) - J)$
$\implies (off + (i-1)\cdot k)\cdot P - J \geq (i-1)\cdot (P\cdot k) - J$
$\iff off + i - 1 \geq i - 1$
$\iff off \geq 0$

2. $O + (1 + off + (i-1)\cdot k + 1)\cdot P + J \leq O + off\cdot P + (i+1)\cdot P\cdot k + J$
$\iff (1 + off + (i-1)\cdot k + 1)\cdot P \leq off\cdot P + (i+1)\cdot P\cdot k$
$\iff (2 + off)\cdot P + (i-1)\cdot k\cdot P \leq off\cdot P + (i+1)\cdot P\cdot k$
$\iff (2 + off)\cdot P \leq off\cdot P + 2\cdot P\cdot k$
$\iff (2 + off)\cdot P \leq (2\cdot k + off)\cdot P$
$\iff 1 \leq k$

3. $\max(0, m \cdot k \cdot P - J) \geq \max(0, m \cdot P \cdot k - J)$
$\iff 1 \geq 1$

4. $O + (m \cdot k + 2) \cdot P + J \leq O + off \cdot P + (m + 2) \cdot P \cdot k + J$
$\iff (m \cdot k + 2) \cdot P \leq off \cdot P + (m + 2) \cdot P \cdot k$
$\iff (m \cdot k + 2) \cdot P \leq (off + (m + 2) \cdot k) \cdot P$
$\iff m \cdot k + 2 \leq off + m \cdot k + 2 \cdot k$
$\iff 2 \leq off + 2 \cdot k$
because $k \geq 1,\ off \geq 0$

□

B.2. Proofs for Preserving Semantics

The following lemma is the proof of Lemma 4.3.1 from page 128.

Lemma B.2.1 (No Wait Delay for Start Event) *Let* $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out})$ *be a function node and let the output language of the activation buffer* $L(EP(I_p))$ *be defined as*

$$
\begin{aligned}
L(EP(I_p)) = \{\ & (\sigma_1, t_1)....(\sigma_i, t_i)...(\sigma_{i+m}, t_{i+m})... \mid\ \sigma_i \in I_p, \\
& (1)\ t_i \in [\max(0, (i-1) \cdot P^- - J), O + (i+1) \cdot P^+ + J) \\
& (2)\ \forall m : t_{i+m} - t_i \in [\max(0, m \cdot P^- - J), O + (m+2) \cdot P^+ + J) \\
\} \quad & \textit{where } i, m \in \mathbb{N}^+
\end{aligned}
$$

According to Lemma 3.4.6 the language of start events is determined as

$$
\begin{aligned}
L(start_f) = \{\ & (start_f, u_1)...(start_f, u_i)...(start_f, u_{i+n})...\} \\
\mid\ & u_1 = 0, \\
& u_{i+1} = \max(u_i, t_i) + [\delta^{min}, \delta^{max}]
\end{aligned}
$$

Let further be δ^{max} *be the maximum delay of any transition* $t \in T$*. Then it holds:*

$$
\begin{aligned}
\Delta^-(L(I_p)) > \delta^{max} \implies\ & \forall i \in \mathbb{N}^+ :\ u_i \leq t_i\ \wedge \\
& wait_{start} = \Delta(L(I_p), L(start_f)) = 0
\end{aligned}
$$

Proof: Proof by complete induction

1. *Base Case (i=1):*

$$u_1 \leq t_1$$

From Lemma 3.3.12, we know that $u_1 = 0 \leq t_1$. □

2. *Induction Step:*

$$u_i \leq t_i \Longrightarrow u_{i+1} \leq t_{i+1}$$

From Lemma 3.4.6, we know that for the language of the start event it holds

$$u_{i+1} = \max(u_i, t_i) + [\delta^{min}, \delta^{max}]$$
$$\overset{u_i \leq t_i}{\Longrightarrow} \; u_{i+1} = t_i + [\delta^{min}, \delta^{max}]$$

From $\Delta^-(L(I_p)) > \delta^{max}$*, we know that*

$$t_{i+1} > t_i + \delta^{max} > u_{i+1}.$$

From $u_i \leq t_i$*, it follows that* $\forall i \in \mathbb{N}^+ : \; t_i - u_i \leq 0$ *holds, and thus it also holds that*

$$wait_{start} = \Delta(L(I_p), L(start_f)) = 0$$

□

The following lemma is the proof of Lemma 4.3.2 from page 129.

Lemma B.2.2 (No Wait Delay for TBD Function Nodes) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$ *and a block* $b \in B$ *with n input signals. Let* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be the function network translation of tbd with a respective function node* $f_b \in \mathcal{F}$ *with one input port with* $n > 1$ *incoming activation channels, a maximum transition delay* $\delta^{max} = wcet(b)$*, and the following causal dependencies on input events* $in_1, ..., in_n$ *and* in_j *with* $j \in \{1, ..., n\}$ *and trigger events* $tr_{bp} \in TR_{bp}$*:*

(1) $\{p_{in_1}.in_1, ..., p_{in_n}.in_n\} \xrightarrow{[wait^-_{start}, wait^+_{start}]} ((p_{in_1}.in_1, ..., p_{in_n}.in_n), start_f) \; \wedge$

(2) $p_{in_j}.in_j \xrightarrow{[wait^-_{in}+wait^-_{start}, wait^+_{in}+wait^+_{start}]} ((..., p_{in_j}.in_j, ...), start_f)$

(3) $\forall j \in \{1, ..., n\}, \; \forall tr_{bp} \mid tr_{bp}[cond_{bp}] \xrightarrow{[\delta_j, \delta_j]} p_{in_j}.in_j : \; \delta_j < bp$

Then it holds:

$$wait^+_{start} = 0, \; wait^-_{start} = 0, \; wait^+_{in_j} < bp - \delta^{max}$$

Proof:

1.

$$wait^+_{start} = 0, \; wait^-_{start} = 0$$

From the source node ϕ_{bp}*, we know that it holds*

$$\forall tr_{bp} \in TR_{bp} : \; \Delta^-(L(tr_{bp})) = \Delta^+(L(tr_{bp})) = bp$$

because it produces a stream with period bp with $J = 0$ *and* $O = 0$ *leading with Def. 3.3.4 to* $tr_{i+1} - tr_i = bp$*. Because TBD is feasible, we know that* $bp > \delta^{max}$*, which leads with assumption (3) to*

$$\forall j \in \{1, ..., n\} : \; \Delta^-(L(p_{in_j}.in_j)) \geq bp \geq \delta^{max}.$$

This leads with Lemma 4.3.1 to $wait^+_{start} = wait^-_{start} = 0$*.*

2.

$$wait_{in_j} < bp - \delta^{max}$$

From (3) and because TBD is feasible, we know that it holds:

$$tr_{bp}[cond_{bp}] \xrightarrow{[\delta_j,\delta_j]} p_{in_j}.in_j,\ \delta_j < bp - \delta^{max}.$$

Thus, it holds for all input events that $\forall i,j \in \{1,...,n\} : \delta_i - \delta_j < bp - \delta^{max}$ *leading immediately to*

$$wait^{+}_{in_j} = \max_{i \neq j}(\Delta^{+}(L(p_{in_j}.in_j), L(p_{in_i}.in_i))) < bp - \delta^{max}.$$

□

The following lemma is the proof of Lemma 4.3.3 from page 131.

Lemma B.2.3 (Preserve Partial Order - Period Multiplier Nodes) *Let tbd be a feasible TBD with* $tbd = (B, type, S, E, \mathcal{FTS}, tr)$, $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *its function network translation and t be simulation step. Let* $s_1,\ s_2 \in S$ *be signals with* $(b_1, s_1, b_2) \in E$, $(b_2, s_2, b_2') \in E$, $(s_1, s_2) \in PO_S(tbd, t)$ *and* $\nexists (s, s_1) \in PO_S(tbd, t)$. *Let furthermore* $fts = tr(b_1) = (per, init)$ *be the firing time specification of block* b_1, *and* tr_{b_1} *be a trigger event of* b_1 *produced by the period multiplier block* $f_{fts} = f^{Mult}(\frac{per}{bp}, \frac{init}{bp}, \{..., tr_{b_1}, ...\})$ *at port* $p_{tr_{b_1}}$. *Then the following holds:*

$$\exists tr_{bp} \in TR_{bp} : (tr_{bp}, p_{tr_{b_1}}.tr_{b_1}) \in PO_{\Sigma}(fn, t)$$

Proof: Due to Def. 4.3.4 the statement holds if all the following holds:

1.

$$active(EP(p_{tr_{b_1}}.tr_{b_1}), t) \overset{Def.\ 4.3.3}{\Longleftrightarrow} 0 \leq O + J - (t\ mod\ P) < bp \wedge r_1 \geq O$$

From $(s_1, s_2) \in PO_S(tbd, t)$ *we know that* $rdy(b_1, t)$ *holds i.e.*

$$t = init + k \cdot per,\ k \in \mathbb{N}_0$$

Following Lemma 4.2.2, the event pattern of output event $p_{tr_{b_1}}.tr_{b_1}$ *is defined as*

$$EP(p_{tr_{b_1}}.tr_{b_1}) = (\{p_{tr_{b_1}}.tr_{b_1}\}, P * \frac{per}{bp}, P * \frac{per}{bp}, J, O + \frac{init}{bp} \cdot P + \epsilon).$$

Because $f^{Mult}_{b_1,fts}$ *has an activation channel from* ϕ_{bp}, *we can determine* $P = bp$, $O = 0$ *and* $J = 0$ *which leads to*

$$\begin{aligned} EP(p_{tr_{b_1}}.tr_{b_1}) &= (\{p_{tr_{b_1}}.tr_{b_1}\}, bp * \frac{per}{bp}, bp * \frac{per}{bp}, 0, 0 + \frac{init}{bp} \cdot bp + \epsilon) \\ &= (\{p_{tr_{b_1}}.tr_{b_1}\}, per, per, 0, init + \epsilon) \end{aligned}$$

Thus, we can show the first part of $active(EP(p_{tr_{b_1}}.tr_{b_1}), t)$ *as follows*

$$\begin{aligned} & 0 \leq init + \epsilon - ((init + k \cdot per) \bmod per) < bp \\ \iff & 0 \leq init + \epsilon - init < bp \\ \iff & 0 \leq \epsilon < bp \end{aligned}$$

The second part $r_1 \geq O = init + \epsilon$ *follows directly from Lemma 4.2.1, where the output language is determined such that the first event occurs at* $t_{1+off} + \epsilon$*, where here* $off = \frac{init}{bp}$*. Because the input language has the period bp it follows*

$$r_1 = t_{1+off} + \epsilon \geq \frac{init}{bp} \cdot bp + \epsilon = init + \epsilon.$$

2.

$$\exists tr_{bp} \in TR_{bp} : \ tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon,\epsilon]} p_{tr_{b_1}}.tr_{b_1}, \ \epsilon < bp$$

From Def. 4.2.2 of the period multiplier node, it follows

$$p^{in}_{fts}.tr[state = s_k] \xrightarrow{[\epsilon,\epsilon]} p_{tr_{b_1}}.tr_{b_1}.$$

From Def. 4.2.6, we also know that there exists an activation channel $c = (p_{tr_{b_1}}, [0,0], p^{in}_{fts})$ *from the source node* ϕ_{bp}*. Furthermore, we know that the condition* $[state = s_k]$ *holds during* $[tr_{bp}, p^{in}_{fts}.tr]$ *because there cannot be any preceding activation that could change the state due to the feasibility of tbd. Thus, each previous activation must be finished before the next simulation step starts. Together with Corollary 3.3.1 and Lemma 4.3.2 it follows:*

$$\exists tr_{bp} \in TR_{bp} : \ tr_{bp}[state = s_k] \xrightarrow{[\epsilon,\epsilon]} p_{tr_{b_1}}.tr_{b_1}$$

and it holds $\epsilon < bp$*.* □

Lemma B.2.4 (Active Event Pattern with Delay) *Let tbd be a feasible TBD with* $tbd=(B, type, S, E, \mathcal{FTS}, tr)$*,* $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *its function network translation and t be a simulation step. Let* $(s, s') \in PO_S(tbd, t)$ *with* $s' \in upd(b)$*. Then it holds*

$$\begin{aligned} & active(EP(M(s)), t) \ \wedge \\ & EP(M(s')) = ren(delay(EP(M(s)), [delay, delay]), M(s')) \\ & where \ delay = wait_{in} + wcet(b) \\ \implies & active(EP(M(s')), t) \end{aligned}$$

Proof: Let $EP(M(s)) = (\{M(s)\}, P, P, J, O)$*. We know that* $J = 0$ *because delays only occur in function nodes, where we only consider WCETs and no BCETs leading to fixed delays. We further know from Lemma 4.3.3 that* $O = init + \epsilon + wcets$*, where*

wcets is the sum of WCETs of predecessor blocks on a path to b producing finally the signal s. Independently from the concrete path, we know that wcets < bp because TBD is feasible. This leads to

$$EP(M(s')) = (M(s'), P, P, 0, O + wait_{in} + wcet(b))$$

If active(EP(M(s), t) holds, we know that $r_1 \geq 0$ must hold because $M(s')$ cannot occur earlier than $M(s)$. It remains to show that

$$\begin{aligned} & 0 \;\leq\; O + wait_{in} + wcet(b) + 0 - (t \bmod P) \;<\; bp \\ \iff\; & 0 \;\leq\; init + \epsilon + wcets + wait_{in} + wcet(b) - (t \bmod P) \;<\; bp \end{aligned}$$

From $s <_t s'$ follows that rdy(b) holds and thus $t = init + k \cdot per$ leading to

$$\begin{aligned} 0 \;\leq\; & init + wcets + wait_{in} + wcet(b) - (init + k \cdot per \bmod P) \;<\; bp \\ \iff & 0 \;\leq\; wcets + wait_{in} + wcet(b) \;<\; bp \end{aligned}$$

This holds because $wcets + wait_{in} + wcet(b)$ is a sum of WCETs of blocks in a partially ordered sequence, which is known to be smaller than bp for feasible TBDs. □

The following lemma is the proof of Lemma 4.3.4 from page 131.

Lemma B.2.5 (Preserve Partial Order: Source Blocks) *Let tbd be a feasible TBD with tbd=$(B, type, S, E, \mathcal{FTS}, tr)$, fn = $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ its function network translation and t be a simulation step. Let $s_1 \in S$ be a signal with $(b_1, s_1, b') \in E$ and $\nexists (s, s_1) \in PO_S(tbd, t)$. Then it holds:*

$$\begin{aligned} & (1)\ \exists tr_{bp} \in TR_{bp} : (tr_{bp}, p_{tr_{b_1}}.tr_{b_1}) \in PO_\Sigma(fn, t) \\ \implies & (2)\ (p_{tr_{b_1}}.tr_{b_1}, M(s_1)) \in PO_\Sigma(fn, t) \end{aligned}$$

Proof: b_1 is either a block without any input signals (except from moore-sequential blocks) or a rate transition block from block a to block b, where $per_a > per_b$. For $per_a < per_b$ it never happens that b runs without a and thus b_1 can never be the first block to be executed. To show (2), we need to show all the following:

1. $$p_{tr_{b_1}}.tr_{b_1}[cond_{tr_{b_1}}] \xrightarrow{[wcet(b), wcet(b)]} M(s_1)$$

   a) *If b_1 is a block without any input signals, it is translated to a function node with a transition triggered by $p^{in}_{b_1}.tr_{b_1}$ that produces $M(s_1)$ in any state leading with Corollary 3.3.1 and Lemma 4.3.2 to*

   $$p^{in}_{b_1}.tr_{b_1} \xrightarrow{[wait_{in}+wcet(b), wait_{in}+wcet(b)]} M(s_1).$$

   $wait_{in} = 0$ because we only have one input signal here and thus do not have to wait for any other signal. Because there exists a channel $c = (p_{tr_{b_1}}, [0, 0], p^{in}_{b_1})$, we can conclude with Theorem 3.3.3 that

   $$p_{tr_{b_1}}.tr_{b_1} \xrightarrow{[wcet(b), wcet(b)]} M(s_1).$$

b) If b_1 is a rate transition block with $per_a > per_b$, there exists a transition triggered by $p^{in}_{b_1}.tr_{b_1}$ that produces $M(s_1)$ when in state s_k. Furthermore, we know that $[s_k]$ holds during $[p^{in}_{b_1}.tr_{b_1}, start_{f_{b_1}}]$ because there cannot be another activation that could change the state due to the feasibility of tbd. Together with Corollary 3.3.1 and Lemma 4.3.2 this leads to

$$p^{in}_{b_1}.tr_{b_1}[state = s_k] \xrightarrow{[wait_{in}+wcet(b), wait_{in}+wcet(b)]} M(s_1),$$

where $wait_{in} = 0$ because we have again only one input signal. Because there exists a channel $c = (p_{tr_{b_1}}, [0,0], p^{in}_{b_1})$, we can conclude with Theorem 3.3.3 that

$$p_{tr_{b_1}}.tr_{b_1}[state = s_k] \xrightarrow{[wcet(b), wcet(b)]} M(s_1)$$

2.

$$active(EP(M(s_1)), t)$$

a) If b_1 is a block without any input signals, we know from (1) that

$$active(EP(p_{tr_{b_1}}.tr_{b_1}), t).$$

From 1.a) follows that $EP(M(s_1))$ can be determined as follows:

$$EP(M(s_1)) = ren(delay(EP(p_{tr_{b_1}}.tr_{b_1}), [wcet(b_1), wcet(b_1)]), s_1).$$

With Lemma B.2.4, it follows that $active(EP(M(s_1)), t)$ holds.

b) If b_1 is a rate transition block with $per_a > per_b$, we know that $per_a = per_b \cdot n$ with $n \in \mathbb{N}^+$, $n > 1$, and there exists a period multiplier node for each $fts \in ex(fts_b, per_a) \setminus fts_a$, where

$$ex(fts_b, per_a) = \{(per_a, (init_b + i \cdot per_b) \bmod per_a) \mid 0 \leq i \leq n-1\}.$$

From Lemma 4.3.3, we know that

$$EP(p_{tr_{b_1}}.tr_{b_1}) = (\{p_{tr_{b_1}}.tr_{b_1}\}, per_a, per_a, 0, (init_b + i \cdot per_b) \bmod per_a + \epsilon).$$

With 1.(b) and Lemma 3.4.1, the event pattern of $M(s_1)$ is defined as

$$EP(M(s_1) = super(..., EP_i(M(s_1)), ...), \text{ where}$$

$$\begin{aligned} &EP_i(M(s_1)) = ren(delay(EP(p_{tr_{b_1}}.tr_{b_1}), [wcet(b), wcet(b)]), M(s_1)) \\ &= (\{M(s_1)\}, per_a, per_a, 0, (init_b + i \cdot per_b) \bmod per_a + \epsilon + wcet(b)). \end{aligned}$$

From $rdy(b_1, t)$ we know that $t = init_b + k \cdot per_b,\ k \in \mathbb{N}_0$.
To show $active(EP_i(M(s_1)), t)$, with Def. 4.3.3, we need to show

i.

$$\begin{aligned}
& 0 \leq O + J - (t \bmod P) < bp \\
\iff\ & 0 \leq (init_b + i \cdot per_b) \bmod per_a + \epsilon + wcet(b) \\
& \quad - (init_b + k \cdot per_b) \bmod\ per_a < bp \\
\iff\ & 0 \leq (i \cdot per_b) \bmod per_a + \epsilon + wcet(b) - (k \cdot per_b) \bmod\ per_a < bp \\
\iff\ & 0 \leq (i \cdot per_b) \bmod (per_b \cdot n) + \epsilon + wcet(b) \\
& \quad - (k \cdot per_b) \bmod\ (per_b \cdot n) < bp \\
\iff\ & 0 \leq i \bmod n + \epsilon + wcet(b) - k \bmod n < bp
\end{aligned}$$

Because for a superposition of event patterns to be active it is sufficient that one of the superposed event pattern is active, we choose the fts with $i = k \bmod n$ and get the following:

$$\begin{aligned}
\iff\ & 0 \leq (k \bmod n) \bmod n + \epsilon + wcet(b) - k \bmod n < bp \\
\iff\ & 0 \leq k \bmod n + \epsilon + wcet(b) - k \bmod n < bp \\
\iff\ & 0 \leq \epsilon + wcet(b) < bp
\end{aligned}$$

ii. *$r_1 \geq O = init + \epsilon$ follows directly from Lemma 4.3.3, where $r_1 \geq init$ has been shown for period multiplier nodes. Because a period multiplier (with an execution time ϵ) is a predecessor of a function node that models a source block, the first event of this node may not occur earlier leading to $r_1 \geq init + \epsilon$.*

3.

$$\exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcet(b),\epsilon+wcet(b)]} M(s_1),\ wcet(b) < bp$$

From (1) and Lemma 4.3.3, we know $tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon,\epsilon]} p_{tr_{b_1}}.tr_{b_1}$ with $\epsilon < bp$. Together with 2. and the feasibility of tbd it follows that $wcet(b) < bp$.

□

The following lemma is the proof of Lemma 4.3.5 from page 132.

Lemma B.2.6 (Preserve Partial Order - Rate Transition Blocks) *Let tbd be a feasible TBD with $tbd = (B, type, S, E, \mathcal{FTS}, tr)$, $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ its function network translation and t be a simulation step. Let $(s, s') \in PO_S(tbd, t)$ and b be a block with $s' \in upd(b)$ and $type(b)$ = 'rate-transition' connecting the blocks a and b with the sample times $fts_a = tr(a) = (per_a, init_a)$ and $fts_b = tr(b) = (per_b, init_b)$. Let further the following hold:*

$$\begin{aligned}
& active(EP(M(s), t) \wedge \\
& tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets,\epsilon+wcets]} M(s),\ wcets < bp
\end{aligned}$$

Then it holds:

$$\begin{aligned}&(1)\ M(s)[cond_{M(s)}] \xrightarrow{[wcet(b),wcet(b)]} M(s') \wedge\\ &(2)\ active(EP(s'),t) \wedge\\ &(3)\ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s'),\ wcets' < bp\end{aligned}$$

Proof:

1. $M(s)[cond_{M(s)}] \xrightarrow{[wcet(b),wcet(b)]} M(s')$

   a) *If* $per_a > per_b$, *there exists a transition triggered by* $p_b^{in}.s$ *that produces* $M(s')$ *in each state leading with Lemma 4.3.2 to*

   $$p_b^{in}.s \xrightarrow{[wait_{in}+wcet(b),wait_{in}+wcet(b)]} M(s')$$

   where $wait_{in} = 0$ *because there is no other input signal than* s. *From Def. 4.2.6, we know that there exists a channel* $c = (p_s, [0,0], p_b^{in})$ *and that* $M(s) = p_s.s$, *which leads with Lemma 4.3.2 to*

   $$M(s) \xrightarrow{[wcet(b),wcet(b)]} M(s').$$

   b) *If* $per_a < per_b$, b *is translated to a period multiplier block, where there exists a transition triggered by* $p_b^{in}.s$ *that produces* $M(s')$ *in state* s_k *leading with Corollary 3.3.1 and Lemma 4.3.2 to*

   $$p_b^{in}.s[state = s_k] \xrightarrow{[wait_{in}+wcet(b),wait_{in}+wcet(b)]} M(s')$$

   where $wait_{in} = 0$ *because we only have one input signal here. From Def. 4.2.6, we know that there exists a channel* $c = (p_s, [0,0], p_b^{in})$ *and that* $M(s) = p_s.s$ *which leads to*

   $$M(s)[state = s_k] \xrightarrow{[wcet(b),wcet(b)]} M(s').$$

2. $active(EP(M(s')),t) \overset{Def.\ 4.3.3}{\Longleftrightarrow} 0 \leq O + J - (t \mod P) < bp \wedge r_1 \geq O$

   a) *If* $per_a > per_b$, *we know from 1a) that* $M(s) \xrightarrow{[wcet(b),wcet(b)]} M(s')$ *holds. With Lemma 3.4.1, the event pattern* $EP(M(s'))$ *is determined to*

   $$\begin{aligned}&EP(M(s')) = super(..., EP_i(M(s')), ...),\ where\\ &EP_i(M(s')) = ren(delay(EP(M(s), [wcet(b), wcet(b)], M(s')).\end{aligned}$$

   With Lemma B.2.4, we can conclude that $active(EP_i(M(s'),t)$ *holds. Thus, according to Def. 4.3.3, also* $active(EP(M(s')))$ *holds.*

b) If $per_a < per_b$, the rate transition block is translated to a period multiplier block $f^{Mult}(k, off, \{M(s')\}, [wcet(b), wcet(b)])$ with $k = \frac{per_b}{per_a}$ and $off = 0$ leading to the event pattern

$$EP(M(s')) = (M(s'), P * k, P * k, J, O + off \cdot P + wcet(b)).$$

With $P = per_a$ this leads to

$$\begin{aligned} EP(M(s')) =& (M(s'), per_a * \frac{per_b}{per_a}, per_a * \frac{per_b}{per_a}, J, O + 0 \cdot per_a + wcet(b)) \\ =& (M(s'), per_b, per_b, J, O + wcet(b)). \end{aligned}$$

From $s <_t s'$, we know that $rdy(b', t)$ holds, where $s' \in upd(b')$ i.e. $\exists k \in \mathbb{N}_0 : init_b + k \cdot per_b = t$. Thus, we get for the first part of $active(EP(s'), t)$

$$\begin{aligned} & 0 \leq O + wcet(b) - ((init_b + k \cdot per_b) \bmod per_b) < bp \\ \iff & 0 \leq O + wcet(b) - init_b < bp. \end{aligned}$$

From Def. 4.2.6, we know that $init_a = init_b$ and we know $O = init_a + wcets$, where $wcets < bp$ is the sum of all executions delays of preceding blocks i.e.

$$\begin{aligned} & 0 \leq init_a + wcets + wcet(b) - init_a < bp \\ \iff & 0 \leq wcets + wcet(b) < bp. \end{aligned}$$

The second part $r_1 \geq O$ results from the output language of the period multiplier node described in Lemma 4.2.1, where the first event occurs at $t_{1+off} + \epsilon$ with $off = 0$. And from $active(EP(s), t)$ it follows that $t_1 \geq O$.

3. $tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets', \epsilon+wcets']} M(s'),\ wcets' < bp$

We know that $tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets, \epsilon+wcets]} M(s)$ with $wcets < bp$ holds. Together with 1., it follows that $tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets', \epsilon+wcets']} M(s')$ holds, where $wcets' = wcets + wcet(b)$, and $wcets$ is the sum of all blocks that have been executed successively before due to the partial order. Because we know that tbd is feasible, we also know that $wcets' < bp$.

□

The following lemma is the proof of Lemma 4.3.6 from page 133.

Lemma B.2.7 (Preserve Partial Order - Ordinary Blocks) *Let tbd be a feasible TBD with $tbd = (B, type, S, E, \mathcal{FTS}, tr)$, $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ its function network translation and t be a simulation step. Let $(s, s') \in PO_S(tbd, t)$ and b be a block with $s' \in upd(b)$, n input signals $in_1, ..., in_n$, where $\exists j : in_j = s$ and $type(b) \in$ {'sequential', 'Moore-sequential', 'combinational'}.*

Let $f_b = (\{p_b^{in}\}, (\{s_0\}, s_0, \{t\}), \{p_{out_1}, ..., p_{out_m}\}) \in \mathcal{F}$ *be the translation of* b *and* $M(s) = p_s.s$ *as defined in Def. 4.2.6. Let further the following hold:*

$$(A)\ active(EP(M(s), t)) \wedge$$

$$(B)\ \exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets, \epsilon+wcets]} M(s),\ 0 \leq wcets < bp$$

Then it holds:

$$(1)\ M(s) \xrightarrow{[wait_{in}+wcet(b), wait_{in}+wcet(b)]} M(s') \wedge$$

$$(2)\ active(EP(M(s')), t) \wedge$$

$$(3)\ \exists tr_{bp} \in TR_{bp} :\ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets', \epsilon+wcets']} M(s'),\ wcets' < bp$$

Proof:

1.

$$M(s) \xrightarrow{[wait_{in}+wcet(b), wait_{in}+wcet(b)]} M(s')$$

After Def. 4.2.6, f_b *has one state and one transition*

$$(p_b^{in}, E, s_0 \rightarrow \{..., (p_{s'}, s', \delta), ...\}, s_0),$$

where $E = \Sigma^{act}(p_b^{in}) = \{in_1, ..., in_n\}$ *and* $s = in_j$*. With Corollary 3.3.1 and Lemma 4.3.2 this leads to*

$$p_b^{in}.(in_1, ..., in_n) \xrightarrow{[wcet(b), wcet(b)]} M(s').$$

Furthermore, Def. 4.2.6 says that for each signal in_j*, there exists a channel* $c = (p_{in_j}, [0, 0], p_b^{in})$*. With Theorem 3.3.3 this leads to*

$$\forall j \in \{1, ..., n\} :\ p_{in_j}.in_j \xrightarrow{[wait_{in}^- + wcet(b), wait_{in}^+ + wcet(b)]} M(s').$$

Because for each delay the bounds are equal, we know that $wait_{in} = wait_{in}^- = wait_{in}^+$*. With assumption (B) and* $M(s) = M(in_j) = p_{in_j}.in_j$*, we get*

$$M(s) \xrightarrow{[wait_{in}+wcet(b), wait_{in}+wcet(b)]} M(s'),$$

where $wait_{in} < bp - wcet(b)$*.*

2.

$$active(EP(s'), t)$$

Following Lemma 3.4.1, we know from the causal dependency shown at 1. and the fact that f_b *has only one input port, that* $EP(M(s'))$ *can be determined as*

$$EP(M(s')) = ren(delay(EP(M(s)), [wcets', wcets']), M(s')),$$

where $wcets' = wait_{in} + wcet(b)$*. Then it follows with Lemma B.2.4 that*

$$active(EP(s'), t).$$

3.

$$\exists tr_{bp} \in TR_{bp} : \ tr_{bp}[cond'_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s'), \ wcets' < bp$$

From assumption (B) and 1. follows that

$$\exists tr_{bp} \in TR_{bp} : \ tr_{bp}[cond_{bp}] \xrightarrow{[\epsilon+wcets',\epsilon+wcets']} M(s'),$$

where $wcets' = wcets + wait_{in} + wcet(b)$*. We know that* $wcets + wait_{in}$ *is the sum of all blocks that have been executed successively before b due to the partial order. Because tbd is feasible, we know that* $wcets' < bp$*.*

□

C. Proofs for Task Creation

C.1. Proofs for Formal Composition Operations

The following lemma is the proof of Lemma 5.2.1 from page 164.

Lemma C.1.1 (Causality of Signals and Channels) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further be $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ be the function network after the self loop has been eliminated. Then it holds:*

$$
\begin{aligned}
&\exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \wedge \\
&\exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \wedge \\
&\exists c_a = (p_{d'}, \delta_a, p_a),\ p_{d'} \in \mathcal{P}_d^{out} \\
\implies &\forall e \in \Sigma(p_w) :\ p_w.e \xrightarrow{\delta_{write}+\delta_{signal}+\delta_{act}} p_a.e \\
\textit{where } &\delta_{write} = wait_{start_w} + \delta_w, \\
&\delta_{signal} = wait_{start_d} + \delta_d, \\
&\delta_{act} = wait_{start_a} + \delta_a
\end{aligned}
$$

Proof:

1. *From Theorem 3.3.3 and Corollary 3.3.2 follows*

$$
\begin{aligned}
&\exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \\
\implies &\forall p_d \in \mathcal{P}_d^{in}, p'_d \in \mathcal{P}_d^{out}, e \in \Sigma(p_d) :\ p_d.e \xrightarrow{wait_{start_d}+\delta_d} p'_d.e \\
\textit{where } &wait_{start_d} = [wait^-_{start_d}, wait^+_{start_d}]
\end{aligned}
$$

2. *From Theorem 3.3.3 and Corollary 3.3.6 follows*

$$
\begin{aligned}
&\exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \\
\implies &\forall e \in \Sigma(p_w) : p_w.e \xrightarrow{wait_{start_w}+\delta_w} p_d.e \\
\textit{where } &wait_{start_w} = [wait^-_{start_w}, wait^+_{start_w}]
\end{aligned}
$$

3. From Theorem 3.3.3 and Corollary 3.3.6 follows

$$\begin{aligned} & \exists c_a = (p'_d, [\delta_a^-, \delta_a^+], p_a),\ p'_d \in \mathcal{P}_d^{out} \\ \implies & \forall e \in \Sigma(p_{d'}) : p_{d'}.e \xrightarrow{wait_{start_a} + \delta_a} p_a.e \\ & \textit{where } wait_{start_a} = [wait^-_{start_a}, wait^+_{start_a}] \end{aligned}$$

From (a),(b) and (c) follows by transitivity the statement to prove. □

The following lemma is the proof of Lemma 5.2.2 from page 165.

Lemma C.1.2 (Causality of Self Loop) *Let $fn = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network with a function node $f = (\mathcal{P}^{in}, \mathcal{A}, \mathcal{P}^{out}) \in \mathcal{F}$ with $\mathcal{A} = (S, s_0, T)$ that has a self loop from its output port $p_w \in \mathcal{P}^{out}$ sending an event w to its input port $p_a \in \mathcal{P}^{in}$ receiving an event a. Let further be $fn' = elim_a(fn, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ the function network after the self loop has been eliminated. Then the following holds:*

$$\begin{aligned} & \exists t_1 = (p_1, in, s \to \Psi_1, s') \in T \mid \exists \psi_1 = (p_w, e, \delta_1) \in \Psi_1 \ \wedge \\ & \exists t_2 = (p_a, e, s' \to \Psi_2, s'') \in T \mid \exists \psi_2 = (p_o, o, \delta_2) \in \Psi_2 \ \wedge \\ & (\\ & \exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \ \wedge \\ & \exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \ \wedge \\ & \exists c_a = (p_{d'}, \delta_a, p_a),\ p_{d'} \in \mathcal{P}_d^{out} \textit{(signal self loop)} \\ & \vee \\ & \exists c_c = (p_w, \delta_c, p_a) \textit{ (channel self loop)} \\ &) \\ \implies & (p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1 + \delta_{loop} + \delta_{start_f} + \delta_2} p_o.o[state_f = s''] \end{aligned}$$

Proof:

1. From Theorem 3.3.4 follows that

$$\begin{aligned} & \exists t_1 = (p_1, in, s \to \Psi_1, s') \in T \mid \exists \psi_1 = (p_w, e, \delta_1) \in \Psi_1 \textit{(red transition)} \\ \implies & (p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1} p_w.e[state_f = s'] \end{aligned}$$

2. From 1. and Corollary 3.3.6 follows that

$$\begin{aligned} & \exists d = (\mathcal{P}_d^{in}, \delta_d, d_{sig}, \mathcal{P}_d^{out}) \in \mathcal{D} \ \wedge \\ & \exists c_w = (p_w, \delta_w, p_d),\ p_d \in \mathcal{P}_d^{in} \ \wedge \\ & \exists c_a = (p_{d'}, \delta_a, p_a),\ p_{d'} \in \mathcal{P}_d^{out} \textit{(signal self loop)} \\ & \vee \\ & \exists c_c = (p_w, \delta_c, p_a) \textit{ (channel self loop)} \\ \implies & \forall e \in \Sigma(p_w) : p_w.e \xrightarrow{\delta_{loop}} p_a.e \end{aligned}$$

where $\delta_{loop} = \delta_{write} + \delta_{signal} + \delta_{act}$*, if the loop involves a signal data node (as shown under 1.), or* $\delta_{loop} = wait_{start_c} + \delta_c$ *if it is a channel self loop (shown in Corollary 3.3.6).*

3. *From Assumption A2 of Def. 5.2.6 follows that*

$$[state_f = s'] \text{ holds during } [p_w.e, p_a.e].$$

4. *With Theorem 3.3.4 and 3. follows that*

$$\begin{aligned} & \exists t_2 = (p_a, e, s' \to \Psi_2, s'') \in T \mid \exists \psi_2 = (p_o, o, \delta_2) \in \Psi_2 \\ \implies\ & p_a.e[state_f = s'] \xrightarrow{\delta_{start_f} + \delta_2} p_o.o[state_f = s''] \end{aligned}$$

5. *All the previous statements lead to*

$$\begin{aligned} & (p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1} p_w.e[state_f = s'] \wedge \\ & p_w.e \xrightarrow{\delta_{loop}} p_a.e \wedge \\ & [state_f = s'] \text{ holds during } [p_w.e, p_a.e] \wedge \\ & p_a.e[state_f = s'] \xrightarrow{\delta_{start_f} + \delta_2} p_o.o[state_f = s''] \\ \overset{Lemma\ 3.3.2}{\implies}\ & (p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1} p_w.e[state_f = s'] \wedge \\ & p_w.e[state_f = s'] \xrightarrow{\delta_{loop}} p_a.e[state_f = s'] \wedge \\ & p_a.e[state_f = s'] \xrightarrow{\delta_{start_f} + \delta_2} p_o.o[state_f = s''] \\ \overset{Lemma\ 3.3.1}{\implies}\ & (p_1.(in), start_f)[state_f = s] \xrightarrow{\delta_1 + \delta_{loop} + \delta_{start_f} + \delta_2} p_o.o[state_f = s''] \end{aligned}$$

□

Index

List of Figures

List of Tables

Bibliography

[1] S. Abdi. *Functional verification of system level model refinements.* PhD thesis, California State University at Long Beach, Long Beach, CA, USA, 2005.

[2] E. Althaus, R. Naujoks, and E. Thaden. A column generation approach to scheduling of periodic tasks. In *Int'l Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 340–351. Springer, 2011.

[3] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235, April 1994.

[4] J. Bao, P. Battram, A. Enkelmann, A. Gabel, J. Heyen, T. Koepke, C. Läsche, and S. Sieverding. Projektgruppe ViDAs – Endbericht. Technical report, Carl von Ossietzky Universität Oldenburg, 2010.

[5] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, pages 200–236. Springer, 2004.

[6] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Int'l Conf. on Concurrency Theory (CONCUR)*, pages 162–177. Springer, 1999.

[7] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation*, PLDI '03, pages 196–207, 2003.

[9] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng (eds.). Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/ERL M05/21, University of California, Berkeley, 2005.

[10] M. Büker, W. Damm, G. Ehmen, A. Metzner, I. Stierand, and E. Thaden. Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too. Reports of SFB/TR 14 AVACS 69, SFB/TR 14 AVACS, 2011. ISSN: 1860-9821, http://www.avacs.org.

[11] M. Büker, W. Damm, G. Ehmen, A. Metzner, I. Stierand, and E. Thaden. Automating the design flow for distributed embedded automotive applications: Keeping your time promises, and optimizing costs, too. In *Proc. International Symposium on Industrial Embedded Systems (SIES'11)*, pages 156–165, 2011.

[12] M. Büker, W. Damm, G. Ehmen, and I. Stierand. An automated semantic-based approach for creating tasks from matlab simulink models. In G. Salaün and B. Schätz, editors, *Proc. of 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Lecture Notes in Computer Science, pages 149–164. Springer Verlag, 2011.

[13] M. Büker, T. Gezgin, and I. Stierand. On the implementability of complex real-time systems. Technical report, SFB/TR 14 AVACS, 2011. AVACS Technical Report No.68.

[14] M. Büker, K. Grüttner, P. A. Hartmann, and I. Stierand. Mapping of concurrent object–oriented models to extended real–time task networks. In *Forum on Specification and Design Languages (FDL)*, 09 2010.

[15] M. Büker, K. Grüttner, P. A. Hartmann, and I. Stierand. Mapping of concurrent object–oriented models to extended real–time task networks. In *System Specification and Design Languages – Selected Contributions from FDL 2010*, pages 37–54. Springer, January 2012.

[16] M. Büker, A. Metzner, and I. Stierand. Testing real-time task networks with functional extensions using model-checking. In *14th International Conference on Emerging Technologies and Factory Automation*, pages 1 – 10, 2009.

[17] L. P. Carloni, S. Member, K. L. Mcmillan, and A. L. Sangiovanni-vincentelli. Theory of latency-insensitive design. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 20, pages 1059–1076, 2001.

[18] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proc. ACM SIGPLAN 2003*, LCTES '03, 2003.

[19] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, 1996.

[20] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 343–354. Association for Computing Machinery (ACM), 1992.

[21] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. In *DVCon 2007*, February 2007.

[22] M. Di Natale. Optimizing the multitask implementation of multirate simulink models. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 335–346, Washington, DC, USA, 2006. IEEE Computer Society.

[23] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of multitask implementations of simulink models with minimum delays. *IEEE Transactions on Industrial Informatics*, 2010.

[24] H. Dierks, A. Metzner, and I. Stierand. Efficient model-checking for real-time task networks. In *6th International Conference on Embedded Software and Systems*, pages 11–18, May 2009.

[25] S. Dutt. New faster kernighan-lin-type graph-partitioning algorithms. In *Proc. Intl. Conf. on Computer-aided design (ICCAD)*, 1993.

[26] U. Eisemann. *Modeling Guidelines for Function Development and Production Code Generation.* dSPACE GmbH, 2006.

[27] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2(1), 1996.

[28] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, Oct. 1993.

[29] J. Esparza and M. Nielsen. Decibility issues for Petri nets - a survey. *Journal of Informatik Processing and Cybernetics*, 30(3):143–160, 1994.

[30] C. Ferdinand. Worst-case execution time prediction by static program analysis. In *Proc. IPDPS*, 2004.

[31] E. Fersman and W. Yi. A Generic Approach to Schedulability Analysis of Real Time Tasks. *Nordic Journal of Computing*, 11, 2004.

[32] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. DAC'82*, pages 175–181, 1982.

[33] T. Gezgin, S. Henkler, A. Rettberg, and I. Stierand. Contract-based compositional scheduling analysis for evolving systems. In *Proceedings of International Embedded Systems Symposium (IESS)*, 2013. to appear.

[34] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods for Computer-Aided Design (FMCAD)*, pages 68–75, 2006.

[35] G. Hamon. A denotational semantics for stateflow. In *ACM Int'l Conf. on Embedded software (EMSOFT)*, 2005.

[36] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. In *IEEE Proceedings Computers and Digital Techniques*, 2005.

[37] T. Henzinger and S. Matic. An interface algebra for real-time components. In *Proceedings of RTAS 2006*, pages 253–263, April 2006.

[38] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9(4):301–334, 2002. appeared March 2003.

[39] K. Huang and L. Thiele. Performance analysis of multimedia applications using correlated streams. In *Conf. on Design, Automation and Test (DATE'07)*, pages 912–917, 2007.

[40] M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications.* PhD thesis, Technical University of Braunschweig, Germany, 2005.

[41] M. Jersak, K. Richter, and R. Ernst. Performance Analysis for Complex Embedded Applications. *Int'l Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[42] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[43] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1), 1970.

[44] S. Kirkpatrick, C. D. J. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[45] B. Knerr, M. Holzer, and M. Rupp. Hw/sw partitioning using high level metrics. In *Proceedings of the International Conference on Computing, Communications and Control Technologies, Vol.7*, pages 33–38, 2004.

[46] S. Kugele and W. Haberl. Mapping data-flow dependencies onto distributed embedded systems. In *Proc. of SERP 2008*, 2008.

[47] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems. In *EMSOFT '09: Proc. of the seventh ACM international conference on Embedded software*, pages 107–116, 2009.

[48] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[49] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997.

[50] C. Y. Liong, R. I. Wan, O. Khairuddin, and Z. Mourad. Vehicle routing problem: models and solutions. In *Journal of Quality Measurement and Analysis, 2008*, pages 205–218, 2009.

[51] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[52] R. Lublinerman and S. Tripakis. Modular code generation from triggered and timed block diagrams. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:147–158, 2008.

[53] R. Lublinerman and S. Tripakis. Translating data flow to synchronous block diagrams. In *Embedded Systems for Real-Time Multimedia*, 2008.

[54] T. P. Management. *Modeling Guidelines for MATLAB/Simulink/Stateflow and TargetLink*. dSPACE GmbH, 2006.

[55] N. Marian and Y. Ma. Translation of simulink models to component-based software models. In *Int'l Workshop on Research and Educationin Mechatronics (REM)*, July 2007.

[56] A. Metzner. Scheduling analysis of distributed real-time systems under functional constraints. In *Proc. Emerging Technologies and Factory Automation, ETFA*, pages 591–599, Sept. 2008.

[57] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. *Parallel and Distributed Processing Symposium, International*, 0:1–13, 2008.

[58] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel fem simulations by algebraic multigrid. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 57–57, Washington, DC, USA, 2006. IEEE Computer Society.

[59] A. Morton. *Hardware/Software Partitioning and Scheduling of Embedded Systems*. PhD thesis, Electrical and Computer Engineering, University of Waterloo, 2005.

[60] P. Mosterman and J. Ciolfi. Using interleaved execution to resolve cyclic dependencies in time-based block diagrams. In *43rd IEEE Conf. on Decision and Control (CDC04)*, 2004.

[61] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 266–284, London, UK, UK, 1979. Springer-Verlag.

[62] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[63] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *Proc. of EMSOFT 2009*, 2009.

[64] G. K. Rand. The life and times of the savings method for vehicle routing problems. In *ORiON: The Journal of ORSSA. 2009*, pages 125–145, 2009.

[65] P. Reinkemeier, I. Stierand, P. Rehkop, and S. Henkler. A pattern–based requirement specification language: Mapping automotive specific timing requirements. In *Software Engineering 2011 Workshopband*, Lecture Notes in Informatics (LNI), pages 99–108. Köllen Druck + Verlag GmbH, 05 2011.

[66] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models.* PhD thesis, Technical University of Braunschweig, Germany, 2005.

[67] A. Roscoe. *On Theory and Practice of concurrency.* Pearson, 2005.

[68] J. Rox and R. Ernst. Construction and Deconstruction of Hierarchical Event Streams with Multiple Hierarchical Layers. In *Proc. ECRTS*, 2008.

[69] J. Rox and R. Ernst. Modeling event stream hierarchies with hierarchical event models. In *Proc. Conf. on Design, Automation and Test*, 2008.

[70] P. Sasikumar, A. Haq, and P. Baskar. A hybrid algorithm for the vehicle routing problem to third party reverse logistics provider. In *8th International Conference on Supply Chain Management and Information Systems (SCMIS), 2010*, pages 1–8, 2010.

[71] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, pages 119–126, Washington, DC, USA, 2004. IEEE Computer Society.

[72] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *Int'l Conf. on Embedded software (EMSOFT)*, 2004.

[73] S. Schliecker and R. Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proc. Conf. on Hardware Software Codesign and System Synthesis*, 2009.

[74] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 38–48, 1998.

[75] J. Srba. Comparing the expressiveness of timed automata and timed extensions of petri nets. In *FORMATS '08: Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, pages 15–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[76] M. Stasch. Automatisierte Partitionierung von MATLAB-Modellen mittels Semantik erhaltender Modelltransformation, 2012. Bachelor thesis.

[77] E. Thaden. *Semi-automatic optimization of hardware architectures in embedded systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2013.

[78] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *The 27th Annual International Symposium on Computer Architecture(ISCA)*, volume 4, pages 101 –104 vol.4, 2000.

[79] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *Real-Time Systems*, 4:145–165, 1992.

[80] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.*, 57:1300–1314, October 2008.

[81] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4), 2005.

[82] W. Vogler. Fairness and partial order semantics. *Inf. Process. Lett.*, 55(1):33–39, July 1995.

[83] E. Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2006.

[84] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. *Real-Time Systems*, 29(2-3):205–225, Mar. 2005.

[85] B. Westphal, I. Stierand, T. Gezgin, and H. Dierks. The power of uppaal - a language-based characterisation of verification complexity. Reports of SFB/TR 14 AVACS 72, SFB/TR 14 AVACS, 2011. ISSN: 1860-9821, http://www.avacs.org.

[86] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In *Computer-Aided Verification (CAV)*, LNCS, pages 22–36. Springer, 2008.