Carl von Ossietzky
Universität
Oldenburg

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

Correctness of Data Flows in Asynchronous Distributed Systems
–
Model Checking and Synthesis

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels

Doktor der Naturwissenschaften (Dr. rer. nat.)

angenommene Dissertation

von Manuel Gieseking

Gutachter:	Prof. Dr. Ernst-Rüdiger Olderog Prof. Bernd Finkbeiner, Ph.D.
Tag der Disputation:	21.07.2022

Abstract

Due to the increasing integration of information technology into our daily life, the *correctness* of such systems plays a major role in their development and is crucial, not least in safety-critical situations. Without computer-aided analysis, *asynchronous distributed systems* in particular are hard to implement correctly due to their multitude of independently acting components. *Model checking* and *synthesis* represent two fully automated, push-button approaches for developing correct implementations from mathematical precise and unambiguous formal models and specifications. Whereas model checking verifies whether a given implementation satisfies a correctness specification, the synthesis approach derives a correct-by-construction implementation from a given specification. *Petri nets* allow us to formally model asynchronous distributed systems, whereas *the branching-time temporal logic CTL** allows us to pose correctness requirements on the temporal behavior of the system. For synthesis, *Petri games* consider a *causal memory* model for the individual components of the asynchronous distributed system.

In this thesis, we introduce new modeling and specification formalisms based on Petri nets, respectively Petri games, and CTL* that enable correctness requirements on the unbounded local data flow in asynchronous distributed systems. The new formalisms are tailored to separate the control flow of the system from the local data flow of the individual components. We provide solving algorithms for the corresponding model checking and synthesis problems with a reasonable complexity, despite the unbounded data flow and the components' incomplete knowledge about their environment in causality-based models. In particular, this enables us to introduce for the first time decision procedures for the synthesis of Petri games with specifications beyond safety requirements. The accompanying tool implementations for both approaches deal with the state explosion problem either via a reduction to a hardware model checking problem in order to use state-of-the-art algorithms and toolboxes provided in this setting or by using symbolic BDD-based algorithms.

Zusammenfassung

Die Korrektheit von informationstechnischen Systemen spielt aufgrund deren wachsenden Einbindung in unser alltägliches Leben eine zunehmend wichtige Rolle und ist nicht zuletzt für sicherheitskritische Systeme entscheidend. Insbesondere *asynchrone verteilte Systeme* sind, aufgrund ihrer Vielzahl von unabhängig voneinander agierenden Komponenten, ohne computergestützte Analyseverfahren schwer korrekt zu implementieren. Die *Modellprüfung (Model Checking)* und die *Synthese* stellen dabei zwei etablierte und vollautomatische Ansätze zur Entwicklung von korrekten Implementierungen aus mathematisch präzisen und eindeutigen formalen Modellen und Spezifikationen dar. Bei der Modellprüfung wird eine gegebene Implementierung daraufhin überprüft, ob sie die Korrektheitsspezifikation erfüllt, wohingegen der Syntheseansatz aus einer gegebenen Spezifikation eine korrekte Implementierung erzeugt. Petri-Netze ermöglichen die formale Modellierung asynchroner verteilter Systeme, während die Logik CTL* es ermöglicht Korrektheitsanforderungen an die zeitlichen Abläufe des Systems zu stellen. Für die Synthese verwenden die Petri-Spiele ein *kausales Gedächtnismodell* für die einzelnen Komponenten des asynchronen verteilten Systems.

In dieser Arbeit führen wir, basierend auf Petri-Netzen bzw. Petri-Spielen und CTL*, neue Modellierungs- und Spezifikationsformalismen ein, die es ermöglichen, Anforderungen an den unbeschränkten lokalen Datenfluss in asynchronen verteilten Systemen zu stellen. Die neuen Formalismen sind darauf zugeschnitten, den Kontrollfluss des Systems vom lokalen Datenfluss der einzelnen Komponenten zu trennen. Wir stellen Lösungsalgorithmen für die entsprechenden Modellprüfungs- und Syntheseprobleme zur Verfügung, die trotz des unbeschränkten Datenflusses und des unvollständigen Wissens der Systemkomponenten über die Umgebung des Systems in kausalitätsbasierten Modellen, eine angemessene Komplexität aufweisen. Insbesondere ermöglicht dies erstmals Entscheidungsprozeduren für die Synthese von Petri-Spielen mit Spezifikationen jenseits von Sicherheitsanforderungen. Die im Rahmen dieser Arbeit für beide Ansätze implementierten Tools behandeln das Problem der Zustandsraumexplosion entweder durch eine Reduktion auf ein Hardware-Model-Checking-Problem, oder durch die Verwendung von symbolischen BDD-basierten Algorithmen.

Acknowledgements

I am deeply indebted to so many wonderful people I was privileged to meet during my time as a Ph.D. student. You have all made this journey such a special experience and have been a constant source of support. So thank you so much, even if you are not explicitly listed here by name!

I am very grateful to my supervisor Ernst-Rüdiger Olderog for his continuous support, intensive discussions, and pleasant conversations. You have always made me feel that you have confidence in me – be it in research or in teaching. This trusting environment let me grow even beyond the working environment. Your love of correctness down to the smallest detail has been an example to me. Bernd Finkbeiner, thank you so much for accompanying and guiding me all these years and especially for welcoming me so warmly into your group (not only for the time of my research stay). You are always full of exciting ideas and energy such that working with you is a steady source of joy and motivation. To all you people in Saarbrücken, thank you for taking me in and making the time in Saarbrücken such a delightful experience. Especially, thank you so much, Jesko! Our weekly video conferences were such a tremendous help professionally and personally. I really enjoyed our travels, conferences, and holidays together.

Another big thanks goes out to my extended group in Oldenburg: Andrea, Björn, Christoph, Christopher, Evgeny, Harro, Hein, Hendrik, Ira, Maike, Mark, Martin, Nick, Nico, Nicolai, Patrick, Paul, Stephanie, Sven, Swami, Thomas, Tim, and Uli. Thank you all for the pleasant working environment, discussions, conversations, and the inspiration. Tim, you have paved the way for me to join the group and your dedication to teaching was a never equaled role model for me. Sven, thank you so much for taking me by the hand for my first steps in research and teaching! You were always there for my questions and problems and I could always count on your opinion. Christopher, Hein, Nick, and Paul thank you for always having an open door and for all these fruitful discussions.

Furthermore, I like to thank Martin Fränzle, Astrid Rakow, Eike Best, Annegret Habel, and Oliver Theel for serving on my thesis committee and/or providing valuable feedback in the other official talks about my thesis. Thank you Ann, Lukas, Moritz, Muhammad, Paul, and Valentin for the joyful cooperation. It was a pleasure to be part of your studies.

Many people have participated in this rollercoaster of emotions (not necessarily voluntarily) as friends, roommates, and more: Aline, Caro, Christopher, Daniel, Doro, Ellen, Holger, Lea, Leo, Lobi, Lucia, Nick, Sebastian, Sven B., Sven L., Thomas, and Vero; thank you so much for bearing with me, enduring my emotions, and having my back. Lastly, I would like to thank my family for their unconditional support and for always providing a safe haven. Thank you, Mum and Dad, for simply everything. Love you!

And finally, thank you so much Lana! Even though you were not yet born at the time of writing this theses, you stood at the gates and pushed me to finally submit. Every day you present me with a whole new and wonderful world! Thanks!

Contents

1 Introduction

The society's dependency on Information Technology (IT) has increased rapidly in recent decades. Computer systems have evolved from specialized tools for the few to an indispensable part of our daily lives; sometimes even beyond notice. We use the technologies, for example, quite naturally in our mobile phones or rely on their existence by using high-speed trains as public transport. The *correctness* of such systems is crucial and an extremely challenging task. While a malfunctioning mobile phone is at best only annoying, a fault in the train's emergency braking system can cause real harm. Although the monetary aspect is of no relevance when human tragedies come into play, defective systems can nevertheless cause high expenses. There are numerous popular examples with severe consequences for the manufactures that have made it into global news like the Ariane-5 and the Mars Climate Orbiter to name only two of the most popular and most expensive ones [BK08].

Due to the constant availability of networks and the ever decreasing space requirements of powerful devices, modern systems are increasingly composed of a huge number of networked computers. Even if the system itself appears to be a single coherent unit, the components of such a *distributed system* act autonomously [TS07]. To avoid a constant communication of every single component with a central control, systems are more and more decentralized. This comes with the cost of an incomplete knowledge of the system's components about the system's environment. In *asynchronous* distributed systems, the single independent components do not progress at a common fixed rate as in a *synchronous* setting, but each component progresses at its own individual rate between the synchronizations with others. This makes it particularly cumbersome to implement algorithms for asynchronous distributed systems correctly, because between synchronizations each component does not know exactly the current state of any other component. However, especially in manufacturing there is a rising demand for the development of local controllers and their mutual communication [Mis12; MH08].

The growth of these systems in size and complexity makes it even more challenging for humans to correctly implement sound controllers. Consequently, this has increased the demand for *computer aided verification techniques*. There are several approaches for improving the development process to obtain correct systems, from *testing* [Kin76; JVCS07] specific aspects of the system by carrying out experiments, to proving the system's correctness with semi-automatic *theorem provers* [NPW02; BC04; Kah07]. While *simulating* and *testing* [Mye04] can show the *existence* of some bugs in the system by making experiments, it is rarely possible to use these techniques to show the *absence* of any faulty behavior due to the sheer amount of possibilities [CGP01]. Furthermore,

the theory of computability with Turing's halting problem [Tur37] and Rice's theorem [Ric53] directly shows the limitations of automatically verifying arbitrary systems, when it is already undecidable by any algorithm whether an arbitrary program terminates. Thus, computer aided verification techniques are always in the area of tension between expressiveness and efficiency [CHV18].

The *model-based* verification techniques work on mathematical precise and unambiguous descriptions of the system's behavior and correctness requirements [BK08]. Thus, in a first step, such formal models have to be created from the often only informally available system design. This leads to the *validation problem*, i.e., the problem of judging whether we have correctly constructed the formal model and correctness specifications from the informal design. Consequently, the result of the algorithm for the correctness analysis can just be as good as the provided input [BK08]. Creating such inputs is a major challenge and far from trivial [Roz16]. Therefore, suitable modeling formalisms tailored to the nature of the problems under consideration are of great importance.

This thesis contributes to developing asynchronous distributed systems *correctly* by following two well-established, fully automated, push-button approaches: model checking and synthesis. The basic idea of *model checking* [CE81; QS82] is, given a finite-state system (the implementation) and a specification stating the system's correctness requirements, to exhaustively explore the system's state space and to return a counterexample trace when the property is violated (see Sec. 1.1). For *synthesis* [Chu57; Chu62], this implementation is automatically generated from the given specification. This allows the developer to focus on the behavior of the system at an abstract level by writing specifications, instead of getting lost in the details of how to actually implement the desired features. This results in implementations that are correct-by-construction and avoids the error-prone task of manually coding implementations (see Sec. 1.2).

1.1 Model Checking

Model checking is a framework for verifying finite-state concurrent systems that has been independently developed in the early 1980s by Clarke and Emerson [CE81] and by Queille and Sifakis [QS82]. It is based on a *model* describing how the system *actually behaves* and a *specification* prescribing what the *desired behavior* of the system is. Thus, the practical process first requires the user to define a formal model of the system description which incorporates all possible behavior of the system in an unambiguous way. Second, the requirements for the system under consideration must be formalized in a mathematically precise specification. Given these input parameters, the model checker can then fully automatically determine whether the model satisfies the specification, and, in case of a violation of the requirements, provides a counterexample trace allowing to diagnose the faulty behavior. This process is visualized in Fig. 1.1.

In principle, model checking is a fully automated push-button technique. This means that different to, e.g., the deductive reasoning with theorem provers, no human interaction or expert knowledge is necessary in the process of verifying whether the given system satisfies the given specification. In the classical form, model checking consists of

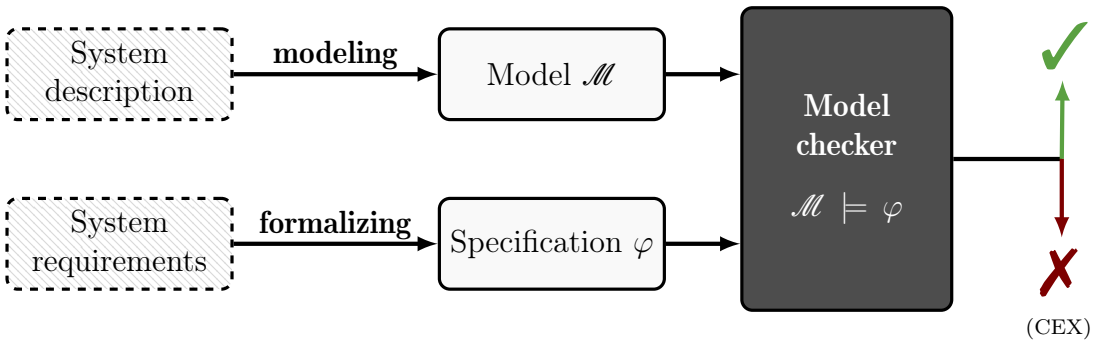


Fig. 1.1: *Schematic overview of the model checking procedure. First, a model is constructed from the actual system and the requirements for the system are formalized in a specification. Triggering the model checker fully automatically determines whether the model satisfies the specification. In the negative case a counterexample (CEX) trace of the model is provided witnessing the violating behavior.*

three basic components [CHV18; CW14]:

Modeling Formalism: Typically, the system under consideration is given in some description language and is not already in the mathematical format expected by the model checking algorithms. Thus, the first task is to create the model $\mathscr{M}$ such that it can be captured by the algorithms. In many cases this step can be done automatically [CGP01]. Classically, the model $\mathscr{M}$ is a *finite-state automaton* or a *Kripke structure* with finitely many *states* and *transitions*. Information like the current state of the system's variables distinguish the states of the model, whereas the evolving of the system from one state to another is determined by its transitions [CHV18; BK08].

Specification Language: Usually, some logic based on *temporal logic* is used as specification language for model checking [CW14]. Temporal logic enriches classical propositional logic with operators referring to the system's behavior over time. As a result we can state properties like "eventually something good happens", "some bad situation never arises", or "under specific circumstances some behavior occurs infinitely often". Probably the most prominent representatives of temporal logics are the *branching-time temporal logic (CTL*)* [EH86] and their syntactic fragments *linear-time temporal logic (LTL)* [Pnu77] and *computation tree logic (CTL)* [CE81]. A main challenge in defining specifications is to cover all the requirements that the system should satisfy [CGP01].

Verification Algorithm: Generally, an exhaustive examination of the entire state space is performed by the algorithms to answer the question whether the finite-state structure $\mathscr{M}$ is a *model* of the formula φ specifying the requirements [CW14]. This is denoted by $\mathscr{M} \models \varphi$ and constitutes the origin of the name model checking [Cla08]. Due to the exponential size of the state space, the *state-explosion problem*, a main challenge for the verification algorithms is to scale to real-life problems [CHV18].

For distributed systems, the model description is typically given implicitly as a product of the individual components that may synchronize on certain points. Especially the different schedules of the asynchronous concurrent components, called *interleavings*, lead in general to an exponential state space when multiplying out the explicit behavior of the system. This *state-explosion* problem constitutes one of the main obstacles to model checking in practice [CGJL+01; DLS06]. To tackle this problem, model checking algorithms are often combined with methods like structural reductions, symbolic algorithms, or abstraction techniques [CHV18].

Instead of explicitly enumerating all the states and transitions of the system, *symbolic algorithms* implicitly represent the state space and the transition relation in a symbolic way through sets. Specialized data structures have been developed for providing compact representations with efficient operations for such set-based approaches. Ordered *binary decision diagrams (BDDs)* are a well-established and effective data structure based on Boolean functions which has already been used for the first symbolic model checkers for finite-state systems [CG18]. For *structural reductions*, the structure of the model description with its implicitly given behavior is exploited. For example, the investigation can be restricted to one of the symmetric behaviors for *symmetry reductions* [ES93; CFJ93; ID93] or it can only be focused on specific interleavings for *partial order reductions* [Pel18]. Such structural reductions preserve all relevant aspects of the system's behavior. Hence, the original system satisfies the specification *if and only if* the reduced one does. For *abstractions* we are loosening this requirement. Though the original system still satisfies the specification *if* the abstraction does, the original system does not necessarily not satisfy the specification *if* the abstraction is not verified. So, for abstractions, the model under consideration is represented by another one that is preferably easier to analyze, but at the same time preserves enough information of the original system to guarantee the above mentioned property. *Counterexample-guided abstraction refinement (CEGAR)* [BS93; Kur14; CGJL+00] iteratively generates such abstractions with increasing precision driven by the abstract counterexamples. Thus, whenever model checking the abstraction yields a counterexample which is no counterexample for the verification of the original system (called *spurious*), this spurious counterexample is used to create the refined abstraction for the next step [DKW08].

Generating counterexamples automatically is one of the key features of model checkers and an especially effective one for convincing system engineers to use formal methods [CV03]. The diagnostic feature of having an actual execution of the model leading to a state of the system violating some property makes counterexamples so beneficial. Since counterexamples often consist of only a small number of states, they are generally more accessible than the huge state space of the system [CV03] and can make a useful contribution to the validation problem by revealing errors in the design of the system and its specifications. *Bounded model checking (BMC)* [BCCZ99; BCCF+99] makes use of counterexamples by restricting the bug finding procedure to counterexamples with length up to a given *bound*. BMC is a very efficient method for finding minimal counterexamples, especially when for example the BDD generation of symbolic algorithms already exceeds the available memory [CV03]. However, due to its bounded nature, BMC can only verify the absence of bugs for systems where iteratively increasing the bound

results in reaching some *completeness threshold* [CKOS04]. This completeness threshold means that when there is a counterexample, then there is already a counterexample of length shorter than the given threshold.

Over the years, a multitude of *model checking tools* have emerged for a variety of application domains and programming languages (e.g., EMC [CES86], SMV [McM93; BCMD+92], SPIN [Hol97; Hol05], JAVA PathFinder [HP00; BHPV00] (uses SPIN), NuSMV [CCGR00], CMC [MPCE+02], CBMC [CKL04], ZING [AQRX04]) and this list is still constantly growing (e.g., GenMC [KV21], Pono [MILY+21], HyperProb [DÁBB21], Intrepid [Bru21]). The increasing tool support has enabled the successful application of model checking to various examples of both theoretical and practical nature [BCMD90; HS96; HJMS03; CFIP+13; CW14; HS99; PB15; BK08].

1.2 Synthesis of Distributed Systems

Reactive synthesis [Chu57; Chu62] addresses the correctness problem from a different angle than verification. Whereas verification approaches *check* the correctness of the given implementation with respect to a given specification, synthesis approaches automatically *derive* the implementation from the given specification or state their non-existence. This results in implementations that are correct-by-construction. Hence, with the synthesis approach, the developer can focus on *what* the features of the system should be, rather than *how* these features are realized by actually implementing them. This shifts the error-prone task of manually writing code that implements the specification of the system to the development of specifications, i.e., going from the imperative to the declarative level [BCJ18].

The question of whether such an implementation exists for a given specification and the corresponding decidability problem are called the *realizability problem*. Originally, this question was posed by Alonzo Church [Chu57] for specifications formalized in the *monadic second-order logic of one successor* (S1S) and for the synthesis of reactive systems. In *reactive systems* [HP84], the system continuously interacts with its environment. In an unpublished technical report [McN65], McNaughton, based on the work of Gale and Stewart [GS53], proposed a game-theoretic treatment of Church's problem [Tho09]. One first solution to this problem was introduced by Büchi and Landweber [BL69] by following this suggestion. Another solution was independently introduced by Rabin [Rab69] following an automata-theoretic approach.

In *game-based* synthesis [BL69], the synthesis problem is formulated as an infinite game over a finite graph between two players. One player, representing the *system*, tries to satisfy the given specification, whereas the other player, representing the *environment*, attempts to provoke a violation. The states of the game are the vertices in the graph, which is called the *game arena*. Each vertex is uniquely associated to either the environment or the system. A play on the arena proceeds in rounds. In each round the player associated to the current vertex chooses the successor vertex with respect to the edges of the graph. The player can make this decision completely informed about the game arena and all turns taken so far. In reactive systems, such plays are generally

of infinite duration. Thus, the games under consideration are infinite in the sense of the plays, but finite in the sense of the state space, i.e., of the game arena on which they are played. A strategy for the system player to win the game against all behavior of the environment constitutes an implementation that is guaranteed to satisfy the specification [Fin16; BCJ18].

Church's original question was posed for systems that can be considered as a single coherent unit. In this scenario the synthesized implementation represents one central controller of the system. A main challenge of this so-called *monolithic synthesis* is the complexity of the algorithms. One way to tackle this problem is to reduce the complexity of the input specification language while remaining sufficiently expressive to formalize interesting problems [Fin16]. A widely used specification language is GR(1) [PPS06; BJPP+12] (for *generalized reactivity of rank 1*). While early algorithms suffered from a nonelementary complexity for S1S specifications [Sto74] and later algorithms from a double-exponential complexity for specifications in linear temporal logic (LTL) [PR89a], the GR(1) fragment of LTL can be solved in polynomial time in the size of the state space (which is usually single-exponential) and allows for a symbolic BDD representation [BJPP+12]. Another way to reduce the complexity is to impose a restriction on the size of the constructed implementations. In *bounded synthesis* [SF07; FS13] the immense search space for a solution to the synthesis problem is traversed in a structured way in order to identify small and thus easier-to-find solutions.

All this has fostered the emergence of a variety of *tools* for the monolithic synthesis (e.g., Anzu [JGWB07], Unbeast [Ehl11], RATSY [BGHK+12], `Acacia+` [BBFJ+12], `slugs` [ER16], `BoSy` [FFT17], `ltlsynt` [MC18], Strix [MSL18]) with an annual synthesis competition [JBCF+19]. Furthermore, the synthesis algorithms and tools have been successfully applied to real-world design problems like the formal specifications for an AMBA AHB Arbiter and the synthesis of its circuit [BGJP+07a; BGJP+07b; GCH13].

Nowadays, most reactive systems are not made up of a single unit, but consist of several independent and distributed components, each of which may have incomplete information about the other components. In this setting, the *synthesis of distributed systems* [PR90; MT01; KV01a] constructs a set of implementations, one for each system component, rather than deriving a single implementation as is the case for the monolithic setting. The implementations must collectively satisfy the given specification of the distributed system. One of the main challenges here is to find out when and how much information the individual processes need to exchange to satisfy their goal [Fin16].

Figure 1.2 provides an overview of the general synthesis process for distributed systems. Given a description of the distributed system and the desired requirements, the input elements for the synthesizer are constructed to obtain mathematical precise objects processable by the tool. Typically, such constructed specifications consist of a single temporal logic formula. However, from an engineering perspective, it may be beneficial to model the general possibilities of the system's architecture separately from the requirements the system should satisfy. This difference is more of a technical nature, since in general the requirements for the architecture can also be expressed within the logical formula [BCJ18]. Given the specification, the synthesizer checks whether an

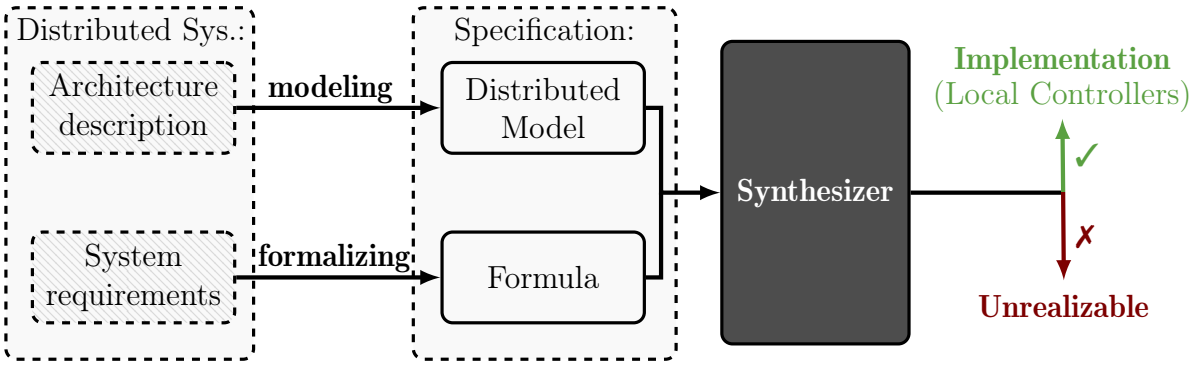


Fig. 1.2: *Schematic overview of the* synthesis of distributed systems. *The description of a distributed system is first formalized in the input language of the synthesizer. For example, such a specification may consist of only a single temporal logic formula. Then, if such an implementation exists, the synthesizer fully automatically derives local controllers for each process of the distributed system that collectively satisfy the given specification.*

implementation exists that satisfies the specification. If this is the case, it fully automatically derives an implementation consisting of local controllers for each component of the distributed system that satisfies the specification.

The generalization of Church's problem for reactive monolithic systems to distributed systems is called the *distributed synthesis problem*. This problem was introduced and shown to be in general undecidable by Pnueli and Rosner [PR90] in a *synchronous* setting. Despite the undecidability of the problem in general, there are also decidability results for restricted architectures. For example, for pipeline architectures [PR90], two-way pipeline and one-way ring architectures [KV01a], the decidability of the distributed synthesis problem is known. Generally, Finkbeiner and Schewe showed the decidability for all architectures without information forks [FS05]. In the *asynchronous* setting, the consideration of a *causal memory* model, i.e., during a synchronization the distributed components exchange their complete causal past, enabled further decidability results [GLZ04; GGMW13; MW14; Gim17].

The synthesis part of our work builds on *Petri games* [FO14; FO17], a *causality-based* multi-player game model for the synthesis of *asynchronous* distributed systems with a local safety objective. Petri games are based on Petri nets [Pet62] in the way that the players of the game are the tokens in the net. In Petri games there are two teams: the environment players and the system players. The common goal of the system players is to collaborate in such a way that each player satisfies their local safety specification against all possible behavior of the environment players. A winning strategy of the system players consists of a correct local controller for each system player. Finding such solutions is non-trivial because at any time in the game the players may have a different level of informedness about the other players and the global state of the system. Furthermore, depending on the players' choices for synchronizations, this level may change dynamically over the time. Despite being undecidable in general [FO14], there exist subclasses of Petri games where the distributed synthesis problem is decidable [FO14], even when the

specification is extended to a global safety condition [FG17; FGHO22]. For example, for Petri games with one environment player, a bounded number of system players, and a local safety objective, the problem is EXPTIME-complete [FO14].

1.3 Contributions

This thesis contributes to the development of asynchronous distributed systems and their correctness analysis in terms of model checking and synthesis. The main contributions are structured in two parts. On the one hand, we introduce a model and a specification language for the local data flow of processes in asynchronous distributed systems and provide model checking algorithms for verifying the correctness of such systems. On the other hand, we extend the previous model and specification language to enable the synthesis of asynchronous distributed systems with causal memory and introduce solving algorithms to automatically generate correct implementations of local controllers for the processes in the system. To show the practical applicability, both parts are accompanied with tool support.

1.3.1 Part I: Model Checking Local Data Flows

We introduce a new framework for model checking asynchronous distributed systems with local data flows. Our model is based on Petri nets [Pet62], a well-established model for asynchronous distributed systems, and on the temporal logic CTL* [EH86] and its fragment LTL [Pnu77], which are commonly used in model checking. The main objective is to have a convenient modeling technique that separates the global configuration of the system and the local data flow of the processes in the system, while providing viable model checking algorithms at an affordable cost.

Petri nets with transits [FGHO19a] allow for this separation by augmenting the flow relation of standard Petri nets with a so-called *transit relation*. The global configuration of the system and its control are specified with a standard Petri net, whereas the local data flow of the processes is specified with the transits. As an example, consider a computer network with a number of switches forwarding data packets according to some routing configuration. The routing process and possible updates to its configuration represent the global control, while the forwarding process of the unbounded number of packets, which may enter the network at any time, represents the data flow.

Flow-CTL* [FGHO20c] allows us to use LTL to reason about the global behavior of the system, e.g., to define maximality and fairness assumptions, and allows us to use CTL* to specify the correctness of the local data flow of the corresponding runs. As a result, a specification is composable of individual requirements for the data flow depending on different control runs. In the networking scenario, for instance, the developer can focus with separate specifications on different concurrent update routines for the routing process. The model checking algorithms then ensure that rolling out these updates does not cause any packet loss or forwarding loops in the network.

We provide dedicated model checking algorithms for the different features of the temporal logic Flow-CTL*. For the linear-time fragment *Flow-LTL* [FGHO19a] the algorithm has a single-exponential time complexity. Many requirements of real-world problems can be expressed with linear-time specifications. For example, the desired requirements of the networking scenario in this thesis are defined using Flow-LTL. The desired requirements for the other application domain introduced in this thesis, access control policies for physical spaces, are expressed in the branching-time fragment *Flow-CTL* [FGHO20c]. Here, the data flow represents the possible paths of people in a building. The corresponding model checking algorithm provided has a double-exponential time complexity. For the complete Flow-CTL* logic, the provided algorithm has a triple-exponential time complexity.

The problem of checking whether a Petri net with transits satisfies a Flow-CTL* formula is reduced to the problem of checking whether a standard Petri net satisfies an LTL formula. For Flow-CTL* and Flow-CTL, this reduction employs a sequence of automata constructions for each individual requirement on the data flow. For Flow-LTL, we provide two different algorithms which both avoid the expensive automata constructions. The first construction constitutes a single-exponential time algorithm which composes suitable subnets in a sequential order [FGHO19a]. The second construction composes the subnets in a parallel manner resulting in a double-exponential time algorithm for specifications with more than one individual requirement on the data flow [FGHO20a]. However, the second exponent is only dependent on the number of individual requirements on the data flow used in the formula. For the examples from the networking scenario with specifications having few individual local data flow requirements, this approach still significantly outperforms the sequential one.

Lastly, we further reduce the model checking problem for safe Petri nets and LTL with places and transitions as atomic propositions to a hardware model checking problem by encoding the Petri net in a circuit [FGHO19a]. Thereby, we can use the state-of-the-art algorithms and toolboxes provided in this setting to solve our initial model checking problem. By implementing the algorithms in the tool AdamMC [FGHO20a; GHY21] we could verify and falsify example specifications for concurrent update routines in network topologies. The tool focuses on the approaches regarding the Flow-LTL specifications, but still provides algorithms for Flow-CTL specifications in an early development state.

1.3.2 Part II: Synthesis of Distributed Systems with Local Conditions

Based on the results of the previous part, this part moves from verification to synthesis. We introduce a new framework for the synthesis of asynchronous distributed systems with causal memory and local data flows. This model, called *Petri games with transits*, combines the features of Petri games [FO14; FO17] and Petri nets with transits [FGHO19a]. In Petri games, the tokens carry the information about their causal past as they flow through the net. This information may be used by the players to decide for their next moves. In Petri games with transits, the data flow defines a second layer of information flow. This layer is used to specify the correctness requirements of the system.

As winning objectives, the new specification language allows for *existential*, i.e., checking whether there *is* a flow in the system satisfying the objective, and *universal* conditions, i.e., checking whether *all* flows in the system satisfy the objective. We define safety, reachability, Büchi, co-Büchi, and parity conditions. Furthermore, we restrict the specification language Flow-LTL [FGHO19a] from the previous part to its local fragment and extend it with the existential view. We thereby provide a specification language for reasoning about temporal constraints on the local data flow of the processes.

We solve the synthesis problem with local specifications for 1-bounded Petri games with transits that have one environment player, a bounded number of system players, and no mixed communication, i.e., a system player must never offer a communication with the environment and at the same time allow to proceed independently with other system players. This enables us for the first time to solve Petri games with winning conditions that go beyond safety requirements. Moreover, Petri games with transits allow for an unbounded number of data flows. While an unbounded number of tokens makes Petri games [FO17] (and therewith Petri games with transits) undecidable, due to the data flow, Petri games with transits provide unbounded features with a decidable synthesis problem.

The synthesis problem for Petri games with transits and local winning objectives is reduced to the synthesis problem of a two-player game over a finite graph with complete information. The reduction method detaches the treatment of the causal memory model in the complete information game from the treatment of the local specifications. This separation and the generality of the proofs facilitates extensibility to further winning conditions. The complexity of the synthesis algorithm depends on the winning condition of the Petri game with transits. For the existential and universal winning conditions, the synthesis problem for Petri games with transits is EXPTIME-complete. For local Flow-LTL, the complexity is single- or double-exponential in the size of the Petri game with transits and double- or triple-exponential in the size of the formula, depending on whether a mixture of existential and universal local flow specifications is used in the formula.

The tool AdamSYNT [FGO15; FGHO17; GHY21] provides BDD-based algorithms for solving the distributed synthesis problem for 1-bounded Petri games with one environment player, a bounded number of system players, no mixed communication, and a local safety objective. Furthermore, we developed algorithms for high-level Petri games [GO21], a succinct representation of sets of Petri games, exploiting the symmetries in the system [GOW20; GW21a; GW20] and provide BDD-based algorithms for subclasses of Petri games with transits in an early development state.

1.4 Structure of the Thesis

The two parts of the thesis are structured analogously. Each part starts with an informal motivation to intuitively introduce the setting of the respective part (Chap. 2 and Chap. 9). Hereafter, Chap. 3 and Chap. 10 introduce established concepts on which we base the approaches of the corresponding parts and provide pointers to the literature

where the concepts are presented and explained in more detail. The reader familiar with these concepts can skip these chapters and use them as a reference in particular for the notations used in this thesis. Chapter 4 and Chap. 11 introduce the new model used in the corresponding parts, whereas the following chapters (Chap. 5, Chap. 6, and Chap. 12) contain the decision procedures for solving the respective problems. To improve readability, the latter chapters each contain a separate section with detailed proofs and further formal definitions. The penultimate chapters (Chap. 7 and Chap. 13) provide insights into the respective tool development, whereas Chap. 8 and Chap. 14 present the related work corresponding to the respective part. Chapter 15 gives a brief overview of the common framework of the previously mentioned tools and introduces the web interface of the unified tool.

1.5 Publications

The present thesis is based on the following peer-reviewed publications:

[FGO15] Bernd Finkbeiner, Manuel Gieseking, and Ernst-Rüdiger Olderog. "Adam: Causality-Based Synthesis of Distributed Systems". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 433–439. DOI: 10.1007/978-3-319-21690-4_25.

[FGHO17] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Symbolic vs. Bounded Synthesis for Petri Games". In: *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. 2017, pp. 23–43. DOI: 10.4204/EPTCS.260.5.

[FGHO19a] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Data Flows in Concurrent Network Updates". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 2019, pp. 515–533. DOI: 10.1007/978-3-030-31784-3_30.

[FGHO20a] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. 2020, pp. 64–76. DOI: 10.1007/978-3-030-53291-8_5.

[FGHO20c] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Branching Properties on Petri Nets with Transits". In: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. 2020, pp. 394–410. DOI: 10.1007/978-3-030-59152-6_22.

[GHY21] Manuel Gieseking, Jesko Hecking-Harbusch, and Ann Yanich. "A Web Interface for Petri Nets with Transits and Petri Games". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. 2021, pp. 381–388. DOI: 10.1007/978-3-030-72013-1_22.

[GO21] Manuel Gieseking and Ernst-Rüdiger Olderog. "High-Level Representation of Benchmark Families for Petri Games". In: *Model Checking, Synthesis, and Learning - Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday*. 2021, pp. 115–137. DOI: 10.1007/978-3-030-91384-7_7.

Parts of the thesis are additionally based on the corresponding full versions of some of the papers mentioned above [FGHO20d; FGHO20b; FGHO19b]. Furthermore, the tool, which was developed accompanying this thesis, is additionally based on the following peer-reviewed publications:

[GOW20] Manuel Gieseking, Ernst-Rüdiger Olderog, and Nick Würdemann. "Solving high-level Petri games". In: *Acta Informatica* 57.3-5 (2020), pp. 591–626. DOI: 10.1007/s00236-020-00368-5.

[GW21a] Manuel Gieseking and Nick Würdemann. "Canonical Representations for Direct Generation of Strategies in High-Level Petri Games". In: *Application and Theory of Petri Nets and Concurrency - 42nd International Conference, PETRI NETS 2021, Virtual Event, June 23-25, 2021, Proceedings*. 2021, pp. 95–117. DOI: 10.1007/978-3-030-76983-3_6.

Again, there is a corresponding more detailed full version [GW21b] of the last paper.

Most of the publications arose in the context of a joint project of Bernd Finkbeiner and Ernst-Rüdiger Olderog on "Petri games", funded by the German Research Foundation (DFG). In particular, this project led to Jesko Hecking-Harbusch's dissertation [Hec21] and to this thesis.

In the papers regarding the model checking approach, i.e., [FGHO19a; FGHO20a; FGHO20c], the main contributions of the author of this thesis refer to the development of the model, the reduction methods, and the implementation of the approaches. These parts do not appear in [Hec21].

The paper [GHY21] introduces the web interface for the model checking and the synthesis approach. Ann Yanich has been an undergraduate student employed on the joint project and supervised by the author of this thesis. For the implementation part of the paper, she developed the web front-end, the infrastructure, and the communication with the web server[1] as part of her contract, while the author of this thesis implemented the necessary interfaces and features in the back-end for turning the command-line tools into libraries usable for interactive tools.

The main contribution of the author of this thesis to the papers regarding the synthesis approach, i.e., [FGO15; FGHO17], is the development and implementation of the BDD algorithms and the benchmark suites for the decision procedure for the synthesis of 1-bounded Petri games with one environment player, a bounded number of system players, and a local safety objective.

For the paper [GO21], the main contribution of the author of this thesis is the development of high-level representations for these benchmark suites. For the implementation exploiting the features of these representations (presented in [GOW20; GW21a]), Lukas Panneke, another undergraduate student employed on the joint project and supervised by the author of this thesis, later implemented parsers and renderers for these so-called *high-level Petri games*. The main contributions of the author of this thesis to the papers [GOW20; GW21a], whose theoretical and practical results are not part of this thesis, but contributed to the accompanying tool, are the experimental results, benchmarks, and the implementation in AdamSYNT.

[1] https://github.com/adamtool/webinterface

Part I

Model Checking Local Data Flows

Contents

2 Motivation

In this chapter we introduce a new framework for model checking the local data flow in asynchronous distributed systems at an intuitive level. The model is called Petri nets with transits [FGHO19a] and the corresponding specification language is called Flow-CTL* [FGHO20c]. For Petri nets with transits, the formal definitions are given in Chap. 4 and for Flow-CTL* in Sec. 5.2. We motivate the formalisms by considering two application domains: *software-defined networking* and *access control policies in physical spaces*.

Petri nets with transits are based on *Petri nets* [Pet62; Rei13], a well-established model for asynchronous distributed systems. In a Petri net, *tokens* represent the distributed processes of the system. They flow independently through the net until synchronizing with others. The synchronizations determine the possible behavior of the global system. Here, we call this flow relation the *control flow*. For example, the update routines of the routing configuration in a software-defined network can be modeled in a distributed manner with the control flow relation. *Petri nets with transits* refine this flow relation to additionally incorporate the *local data flow*. In the networking example, the forwarding process of the unbounded number of packets, which may enter the network at any time, can be represented by the local data flow.

*Flow-CTL** is a temporal logic based on the *branching-time temporal logic CTL** [EH86; CHVB18]. Temporal logics can be used to make statements about the temporal constraints of the system. For example, it can be specified that a formula should eventually hold at some point in time or should globally hold for all points in time. Flow-CTL* is composed of CTL* and its linear-time fragment *linear-time temporal logic (LTL)* [Pnu77; CHVB18]. CTL* is used to reason about the local data flow, whereas LTL is used to reason about the global configuration of the system, i.e., the control flow. This allows us to select specific runs of the system, e.g., those adhering to fair schedulings, and to only check the correctness of the selected runs with respect to the data flow. For instance, a network administrator can focus on different concurrent update routines for the routing process with separate specifications. The model checking algorithm then ensures that rolling out these updates does not cause any packet loss or forwarding loops in the network. Interestingly, the separation of the different aspects of the system yield different timelines for checking the control part of the formula and the local data flow parts. For the control, every step of the global system has to be considered, whereas for each local data flow, only those steps are relevant that involve the own flow and do not belong to some concurrent behavior.

In Sec. 2.1, we consider *software-defined networking (SDN)* [MABP+08; CFG14] as application domain for linear-time specifications, i.e., the fragment *Flow-LTL* of Flow-CTL* examined in Chap. 6. Here, we allow for global, overlapping, and concurrent update routines for the routing process in networks. The correctness is stated on the local flow of the data packets in the system to, e.g., prevent data loss or forwarding loops. For the branching specifications examined in Chap. 5, *access control for physical spaces* [FTFO12; FCS11] is considered as application domain in Sec. 2.2. Here, we allow for concurrent update routines of the door policies. People roaming through a building are considered as the local data flow of the system. The correctness is, for example, stated in the sense that there must always be a way to the emergency exit, or forbidden areas are only permitted for persons belonging to the access-authorized status group.

This chapter is based on the ideas presented in [FGHO19a; FGHO20c].

2.1 Software-Defined Networking

The networking technology *software-defined networking (SDN)* [MABP+08; CFG14] decouples the routing process, i.e., where the packets are sent to, called the *control plane*, from the *data plane*, i.e., the actual packet forwarding process. On the control plane, an update to the routing configuration can be initiated by a central controller and is then implemented in a distributed manner in the network. Correctly implementing concurrent updates is a serious challenge. Even though it is desirable to implement the update as concurrent as possible, some switches have to be updated in a sequential manner to, e.g., prevent the loss of packets or forwarding loops.

Consider for example the networking update problem depicted in Fig. 2.1 based on [FMW16]. The topology with five switches A,C,P,S, and L is given in Fig. 2.1c as circles and their physical connections are depicted by black edges. The initial configuration is given in Fig. 2.1a in a simplified syntax based on [MRFR+13; FHFM+11] and is depicted with solid blue arrows in Fig. 2.1c, whereas the final configuration is given in Fig. 2.1b and is depicted by dashed orange arrows. So all data is entering the network in switch A (`ingress={A}`) and leaves the network in switch C (`egress={C}`). Each forwarding, e.g., in the initial configuration all data is forwarded from switch A to switch S (`A.fwd(S)`), is depicted by arrows in Fig. 2.1c. An update routine has the purpose to restructure the network from the initial configuration to the final configuration. When rolling out the update concurrently in any order, the network could have an intermediate configuration where the packets loop between switches P and L. This routing loop occurs when the update of switch P is implemented before the update of switch L. This problem would eventually be resolved when switch L is finally updated but due to the asynchronicity of the update there are no guarantees on when this will happen. To avoid this undesired behavior, a correct update is given with

$$\texttt{upd(A.fwd(P))} \;||\; \texttt{(upd(L.fwd(C))} \;\gg\; \texttt{upd(P.fwd(L)))}. \tag{2.1}$$

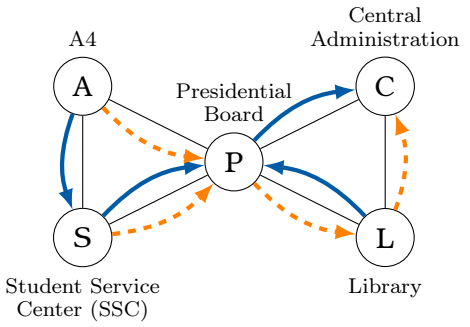
This means that the update of switch A (depicted by `upd(A.fwd(P))`) is done concurrently (depicted by ||) to the sequential update of first updating switch L and then

```
ingress={A}
  A.fwd(S)
  S.fwd(P)
  P.fwd(C)
  L.fwd(P)
 egress={C}
```

(a) *The initial configuration of a network topology.*

```
ingress={A}
  A.fwd(P)
  S.fwd(P)
  P.fwd(L)
  L.fwd(C)
 egress={C}
```

(b) *The final configuration of a network topology.*



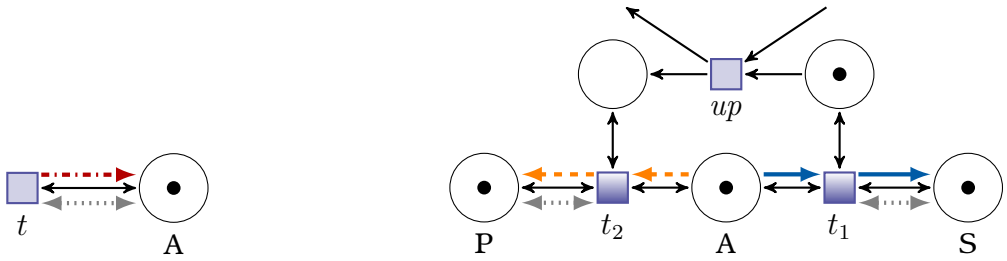
(c) *A network topology with the initial (blue/solid) and final routing (orange/dashed) depicted.*

Fig. 2.1: *A network topology with an initial and a final configuration for routing the data packets through the network.*

switch P (depicted by »). The notation is based on [MRFR+13].

With Petri nets with transits we can model such topologies, configurations, and updates and with Flow-LTL we can express standard networking properties like loop freedom, drop freedom, connectivity, and packet coherence. In Chap. 6 we show generally how to model check Petri nets with transits against Flow-LTL and by that we can also verify the correctness of such concurrent *overlapping* updates in software-defined networking. Rather than only model checking the initial and the final configuration, this routine also checks every configuration inbetween. This means that due to the higher complexity of model checking each configuration, we can spare the implementation costs for ensuring additional properties like consistency. The *consistency* property [RFRS+12] restricts the system such that each packet is only allowed to be either transmitted via the initial or the final configuration. This can be ensured, e.g., by tagging the packets with their configuration identifier and keeping the entries for the old configurations in the routing table after the update. With the overlapping updates we drop this restriction and additionally permit that the packets can be transmitted via any mixture of these configurations.

In Fig. 2.2 we show two patterns how to create a Petri net with transits for software-defined networking. The circles are called *places*, the squares *transitions*, and the dots *tokens*. A transition can fire (or is *enabled*), when all predecessor places connected via the black arrows contain a token. When an enabled transition *fires* it takes all tokens of the predecessor places and puts a token in each successor place connected via the black arrows. This constitutes the *control plane*. The colored arrows describe the *data plane*. Thus, when transition t fires in Fig. 2.2a the token is moved from switch A to switch A due to the black double-headed arrow and also all data packets already in switch A are kept in A due to the gray dotted double-headed arrow. Due to the red dotted and dashed arrow new data is given into the network each time transition t fires. With this pattern we can model the ingress nodes of any network. In Fig. 2.2b two forwarding rules (`A.fwd(P)` and `A.fwd(S)`) are depicted. In the depicted configuration only



(a) *The creation of infinitely many data packets in switch* A*. Whenever transition* t *fires, new data is given into the network.*

(b) *A global configuration of the network routing. Currently, the data can only be transmitted from switch* A *to switch* S *via transition* t_1*. When transition 'up' fires, the packets can only be transmitted to switch* P *via transition* t_2*.*

Fig. 2.2: *Example patterns for modeling software-defined networking with Petri nets with transits.*

the forwarding to switch S is enabled. Thus, transition t_1 can fire but transition t_2 cannot fire due to a missing token in its preset. Each time transition t_1 fires all data in switch A is moved to switch S but also due to the gray dotted doubled-headed arrow no data already in switch S is lost when firing transition t_1. The update transition *up* can move the token so that transition t_1 gets disabled and transition t_2 gets enabled. The additional black arrows indicate that such an update transition normally depends on more tokens to activate this update concurrently with others or in a sequential manner. Using these patterns, we can create the example network and initial configuration depicted in Fig. 2.1, together with the concurrent overlapping update given in Eq. 2.1. The result is depicted in Fig. 2.3.

The five switches A,C,P,S, and L with their corresponding forwarding transitions are depicted by the places in the center of Fig. 2.3. The activation places a_i with $i \in \{\mathrm{A} \to \mathrm{P}, \mathrm{P} \to \mathrm{C}, \mathrm{A} \to \mathrm{S}, \mathrm{L} \to \mathrm{P}, \mathrm{L} \to \mathrm{C}, \mathrm{S} \to \mathrm{P}, \mathrm{P} \to \mathrm{L}\}$ for the forwarding transitions contain a token corresponding to the initial configuration. The update routine, arranged around the center, can update the switches by activating and deactivating the corresponding forwarding transitions. Starting in u_{init} the update creates two tokens residing in $u_{||_j}$ with $j \in \{1, 2\}$, one for each concurrent part of the update. The transition $»_x$ with $x \in \{\mathrm{A}, \mathrm{L}, \mathrm{P}\}$ updates switch x.

With Flow-LTL we can specify standard networking properties like *connectivity*, i.e., *all* data packets ($\mathbb{A}$) which are given into the network in the ingress switch A, should *eventually* ($\Diamond$) reach the egress switch C: $\mathtt{con} = \mathbb{A}(\mathrm{A} \to \Diamond \mathrm{C})$. This formula is not satisfied by the Petri net with transits depicted in Fig. 2.3 because without any fairness assumptions there is for example the run of the system where transition *ingr* fires infinitely often and the data packets never reach switch C. Thus, each forwarding transition t must be weak fair, i.e., when it is at some point infinitely long enabled, it has to fire infinitely often: $\mathtt{wf}(t) = \Diamond\Box\mathtt{enabled}(t) \to \Box\Diamond t$. The symbol $\Box$ stands for *globally* or *always*. Furthermore, we do not want to get stuck within an update. Hence, also the

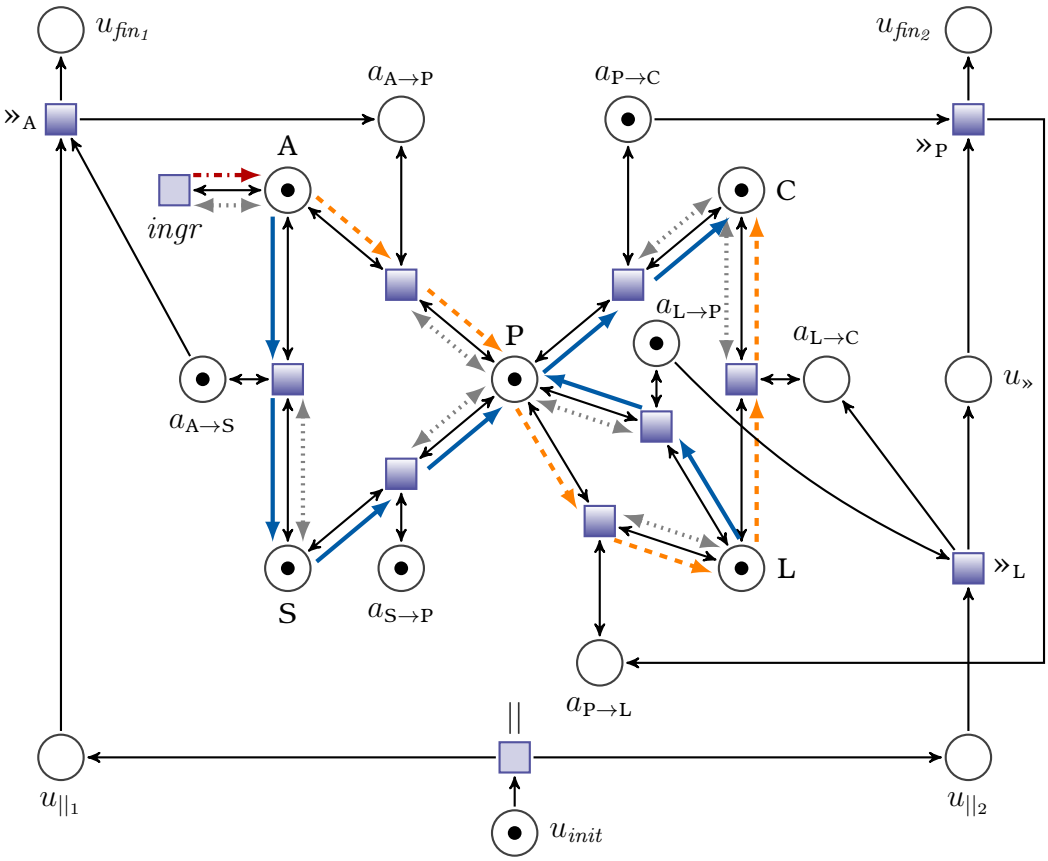


Fig. 2.3: *The Petri net with transits corresponding to the topology and initial configuration depicted in Fig. 2.1 and the concurrent overlapping update given in Eq. 2.1 of the example for software-defined networking. The five switches* ***A,C,P,S,*** *and* ***L*** *and the routing is depicted in the center of the picture, whereas the concurrent update is arranged around it.*

corresponding update transitions have to be weak fair. In Fig. 2.3 the weak fair transitions are represented by shading from white to blue. With Flow-LTL we can select such fair runs and check the data flow of these runs. Hence, the formula $\bigwedge_{t\in\blacksquare} \texttt{wf}(t) \to \texttt{con}$, where $\blacksquare$ is the set of all transitions marked as weak fair, is indeed satisfied by the Petri net with transits depicted in Fig. 2.3.

In the tool AdamMC (cp. Chap. 7) we implemented the routine to automatically generate a Petri net with transits from a topology, initial configuration and concurrent update and to enable model checking standard network properties like loop freedom, drop freedom, connectivity, and packet coherence [FGHO20a].

2.2 Physical Access Control

In the examples for software-defined networking, it is sufficient to consider linear temporal specifications because we are only interested in whether the flow of the data packets

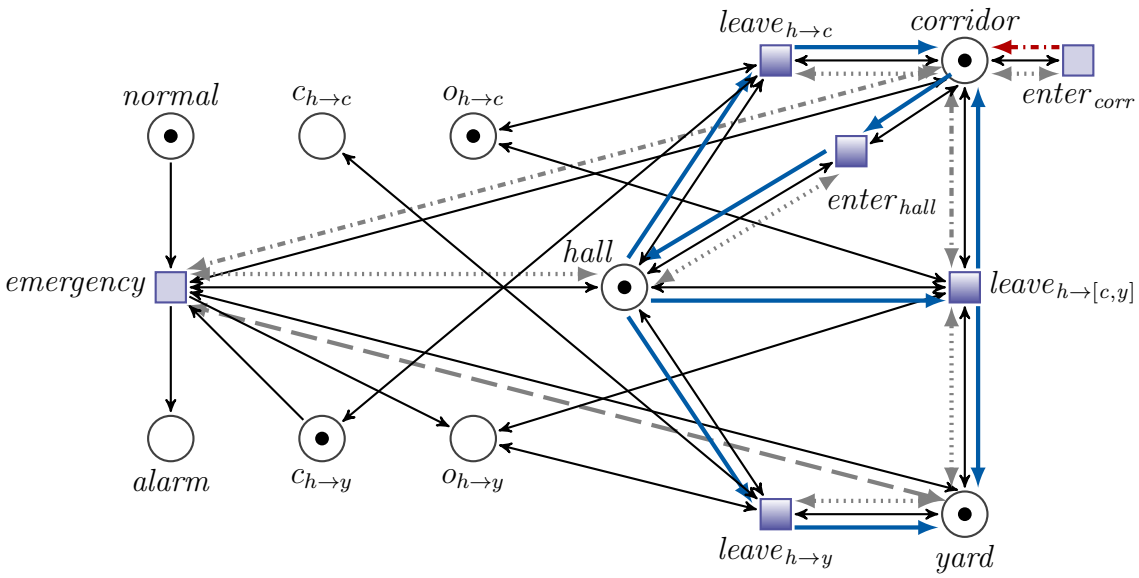
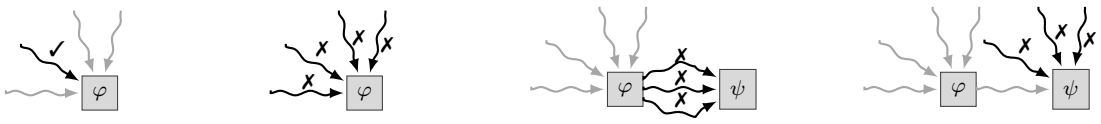


Fig. 2.4: *An example for access control policies for physical spaces modeled with Petri nets with transits. A lecture hall with two exits (the corridor and the yard) is depicted. An emergency introduces an update of the door policies, such that the former closed door to the yard is additional available to the people in case of an emergency. The solid blue single-headed arrows describe the possible paths of people moving through the building.*

behaves correctly and therefore only track the paths of the packets. However, for *access control policies for physical spaces* [FTFO12; FCS11] we consider people roaming through a building as the data flow. By that, interesting specifications like whether there is a way to the emergency exit for each person to every point in time, require branching properties. This can be seen in the example of a lecture hall depicted in Fig. 2.4 as a Petri net with transits.

The control part with the update policy of the doors is depicted at the left-hand side and the data flow at the right-hand side of the figure. The lecture hall indicated by place *hall* has two exits, one to the corridor (indicated by the place *corridor*) and one to the yard (indicated by the place *yard*). Each door can either be open o_i or closed c_i for $i \in \{h \to c, h \to y\}$. In the *normal* mode the access policy of the doors is that only the exit to the corridor is available to the people. In case of an emergency (transition *emergency* fires) an alarm is triggered and the people are informed about the emergency by the gray dashed, dotted, and dash dotted double-headed arrows of the *emergency* transition. With transition $enter_{corr}$ arbitrarily many people can enter the building and can further enter the hall via transition $enter_{hall}$. For simplicity reasons this door is always open. Initially, only transition $leave_{h\to c}$ is enabled, because only the door to the corridor is open. Hence, the only way to leave the hall is via this transition into the corridor. When the update happens, i.e., transition *emergency* fires, a token is additionally put into place $o_{h\to y}$. This means access to the yard is granted and transitions $leave_{h\to y}$ and $leave_{h\to[c,y]}$ are also enabled. Especially the last transition is



$\mathbb{A}\,\mathbf{EF}\varphi$ $\mathbb{A}\,\mathbf{AG}\neg\varphi$ $\mathbb{A}\,\mathbf{AG}(\varphi \to \mathbf{AG}\neg\psi)$ $\mathbb{A}\,\mathbf{A}(\varphi\,\mathbf{R}\,\neg\psi)$

(a) ***Permission:*** *There must be a path to the φ-space. The φ-space can be accessed.*

(b) ***Prohibition:*** *There is no path allowed into the φ-space. The φ-spaces cannot be accessed.*

(c) ***Blocking:*** *There is no path first entering a φ-space and then entering a ψ-space. The ψ-spaces cannot be accessed after accessing the φ-spaces.*

(d) ***Waypointing:*** *There is no direct path into the ψ-space. A φ-space must be accessed before accessing a ψ-space.*

Fig. 2.5: *Some example specifications in Flow-CTL* for access control policies for physical spaces due to [TDB16]. The ✓-symbol indicates a granted access, whereas the ✗-symbol indicates a forbidden access.*

interesting because due to the branching the people are allowed to either leave the hall to the corridor or to the yard. Thus, with Flow-CTL* we can state that the yard is a prohibited area as long as there is no emergency, and when there is an emergency, there exists always a path to the yard:

$$\mathbb{A}(\mathbf{A}(\mathbf{AG}\neg \mathit{yard}\,\mathbf{U}\,\mathit{emergency}) \wedge \mathbf{AG}(\mathit{emergency} \to \mathbf{EF}\,\mathit{yard})).$$

This reads for all data flow trees ($\mathbb{A}$) holds that for all path ($\mathbf{A}$) globally ($\mathbf{G}$) the formula $\neg\mathit{yard}$ is satisfied until ($\mathbf{U}$) there is an emergency, and that on all path globally holds that when there is an *emergency*, then there is a path ($\mathbf{E}$) which finally ($\mathbf{F}$) reaches the *yard*. Note that for simplicity reasons we assume here an emergency has to happen. Since we again evaluate these formulas on specific runs of the system, this formula does not hold for the Petri net with transits depicted in Fig. 2.4 without the depicted weak fairness assumptions on the transitions. This selection of the runs is also the reason why we consider the branching not in the places of the Petri net with transits but in the transitions. Generally, with Flow-CTL* we can express standard properties for access control policies for physical spaces, e.g., introduced in [TDB16], as depicted in Fig. 2.5.

3 Models and Objectives

This chapter serves to introduce established concepts on which we base our procedure for model checking asynchronous distributed systems with local data flows. In model checking we have, on the one hand, the system which should be verified to behave correctly and, on the other hand, a specification language to define what correctness exactly means. For the system, we introduce *Petri nets* [Pet62; Rei13], a well-known model for asynchronous distributed systems, which we extend in Chap. 4 to explicitly incorporate the data flow of the system. The possible behavior of a Petri net is often represented by a state graph which exhibits all interleavings of the system explicitly. *Petri net unfoldings* [Eng91; EH08] with an explicit representation of the concurrency and the causal dependencies of the events in the system also cover all possible behavior of the system but keep the concurrent structure to not explicitly enumerate the interleavings.

As specification language we introduce the commonly used *branching-time temporal logic CTL** [EH86; CHVB18], with its fragment *linear-time temporal logic (LTL)* [Pnu77; CHVB18]. With temporal logics we are able to express properties about computation sequences of the system, like whether a condition holds in the next state, it holds until another condition is satisfied, it holds eventually in the future, or it holds globally. In Sec. 5.2 and Sec. 6.1 these logics are extended to allow for the reasoning about the data flow in the system.

Finally, we introduce different kinds of automata over infinite words and trees [GTW02; PP04; KN01; CDGJ+08]. The reduction method of the model checking problem presented in Chap. 5 is based on a sequence of automata constructions for which these models are used.

3.1 Petri Nets

In this section we recall the syntax and semantics of Petri nets [Pet62; BF88; Rei13]. Furthermore, we recap the definitions corresponding to the unfolding [NPW81; Eng91; KKV03; EH08] to have a formal model for the causal dependencies (and independencies) of the behavior of the distributed system while preserving its concurrent structure.

3.1.1 Definition

A *Petri net*, introduced by Carl Adam Petri in his PhD thesis [Pet62], is a directed bipartite possibly edge-labeled graph with *places* (depicted as circles) storing the current state of the system and *transitions* (depicted as rectangles), also called *events*, executing the state changes. *Tokens* (depicted as black dots) can be considered as the processes

of the system and are moved by the transitions from places to places. As an example consider Fig. 2.3. Ignoring all colored arcs results in a standard Petri net.

A finite and non-empty set of symbols Σ is called *alphabet* and its elements *letters*. The symbols Σ^ω and Σ^* denote the set of all infinite or finite concatenations of letters of Σ, respectively. Such concatenations are called *words*. The *length* of a word w is given by $|w|$ and its letters can be addressed via indices starting with zero. The *empty word* is denoted by ε.

A *multiset* is an extension to a standard set such that it can contain a finite number of the same element rather than a single one. The order still does not matter. We use *multisets* to handle several tokens residing in the same place. Formally, a *multiset* over a set S is a function $M : S \to \mathbb{N}$ mapping a natural number to every element of the set counting its occurrence in M. We identify multisets only mapping to $\{0, 1\}$ with standard sets. Applying a function $f : S_1 \to S_2$ to a multiset $M_1 : S_1 \to \mathbb{N}$ results in the multiset $M_2 : S_2 \to \mathbb{N}$ with $M_2(s_2) = \sum_{s_1 \in S_1 : f(s_1) = s_2} M_1(s_1)$ for all $s_2 \in S_2$.

▶ **Definition 1 (Petri Net).** A (Place/Transition or P/T) *Petri net* is a four-tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ with

- a possibly infinite set of *places* $\mathcal{P}$,
- a possibly infinite set of *transitions* $\mathcal{T}$ (disjoint to $\mathcal{P}$, i.e., $\mathcal{T} \cap \mathcal{P} = \emptyset$),
- a *flow function* $\mathcal{F} : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \to \mathbb{N}$ assigning to each arc connecting places and transitions a non-negative integer *arc weight* (or *arc multiplicity*), and
- an *initial marking* $In : \mathcal{P} \to \mathbb{N}$, defining the number of tokens initially residing on the places.

The elements in $\mathcal{P} \cup \mathcal{T}$ are called *nodes*. We call a Petri net *plain* iff the flow function only maps to the values 0 and 1, i.e., $\mathcal{F}$ can be seen as a set rather than a multiset. ◀

For each place $p \in \mathcal{P}$ of a Petri net $\mathcal{N}$ we define the *pre-* and *postset* of p as all transitions connected to the place with an arc weight at least 1, i.e., $pre(p) = \{t \in \mathcal{T} \mid \mathcal{F}(t, p) > 0\}$ and $post(p) = \{t \in \mathcal{T} \mid \mathcal{F}(p, t) > 0\}$. Similarly, we define the *pre-* and *postset* of a transition $t \in \mathcal{T}$ as multisets over $\mathcal{P}$ such that $pre(t)(p) = \mathcal{F}(p, t)$ and $post(t)(p) = \mathcal{F}(t, p)$ holds for all $p \in \mathcal{P}$. For the transitions we require *finite synchronization* [BF88], i.e., the pre- and postsets of transitions are finite sets. In general, we equip the functions *pre* and *post* with a superscript, e.g., $pre^{\mathcal{N}}$, when we want to stress the dependency on a Petri net $\mathcal{N}$. A *marking* is a multiset $M : \mathcal{P} \to \mathbb{N}$ representing a global state of the distributed system. A transition $t \in \mathcal{T}$ is *enabled* (or *fireable*) in a marking M iff all places in the preset of t contain at least as many tokens as the arc weight deducts, i.e, $\forall p \in \mathcal{P} : \mathcal{F}(p, t) \leq M(p)$. We call a marking M *final* iff no transition $t \in \mathcal{T}$ is enabled in M. An enabled transition $t \in \mathcal{T}$ *firing* in a marking M leads to a successor marking M' by removing the by $\mathcal{F}$ required tokens from the preset of t and putting the by $\mathcal{F}$ specified token numbers in the postset, i.e., $\forall p \in \mathcal{P} : M'(p) = M(p) - \mathcal{F}(p, t) + \mathcal{F}(t, p)$. This is denoted by $M[t\rangle M'$. For a sequence of enabled

transitions $t_0, \ldots, t_n \in \mathcal{T}$ with $n \in \mathbb{N}$, we write $\zeta = M_0[t_0, \ldots, t_n\rangle M_{n+1}$ iff markings $M_0, \ldots, M_{n+1}$ exist such that $M_i[t_i\rangle M_{i+1}$ holds for all $i \in \{0, \ldots, n\}$ and call ζ a *firing sequence*. For a shorter notation, we write $M_0[w\rangle M_n$ for a finite word $w = t_0 \cdots t_n \in \mathcal{T}^*$ iff $M_n = M_0$ in case of the empty word $w = \varepsilon$ and $M_0[t_0, \ldots, t_n\rangle M_n$ otherwise. Firing sequences can also be infinite, written for example as $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$. If M_0 is the initial marking of the net, ζ is called an *initial firing sequence*. Collecting all markings which are transitively reachable from the initial marking by the firing relation yields the set of *reachable markings*. Connecting these markings with respect to the firing relation yields the *reachability graph*. In a reachability graph the complete behavior of the Petri net is exhibited by enumerating all interleavings explicitly.

▶ **Definition 2 (Reachability Graph).** The set of *reachable markings* of a Petri net $\mathcal{N}$ is defined by $R(\mathcal{N}) = \{M : \mathcal{P} \to \mathbb{N} \mid \exists w \in \mathcal{T}^* : In[w\rangle M\}$. Using these markings as states and connecting them with respect to the firing relation yields a labeled transition system called the *reachability graph* $RG(\mathcal{N}) = (S, E, s_0)$ with the set of *states* $S = R(\mathcal{N})$, the set of *edges* $E = \{(M, t, M') \in S \times \mathcal{T} \times S \mid M[t\rangle M'\}$, and the *initial state* $s_0 = In$. ◀

We call a transition $t \in \mathcal{T}$ *dead* iff there is no reachable marking $M \in R(\mathcal{N})$ such that t is enabled in M. A Petri net $\mathcal{N}$ is called *k-bounded*, for $k > 0$, when every place in every reachable marking contains at most $k \in \mathbb{N}$ tokens, i.e., $\forall M \in R(\mathcal{N}), \forall p \in \mathcal{P} : M(p) \leq k$. Note that the set of reachable markings $R(\mathcal{N})$ and the reachability graph $RG(\mathcal{N})$ for k-bounded Petri nets are finite. A 1-bounded Petri net is also called *safe*. In this thesis we mainly focus on 1-bounded Petri nets. This allows for a more lightweight notation such that markings are sets $M \subseteq \mathcal{P}$ rather than multisets, just as the pre- ($pre(t) = \{p \in \mathcal{P} \mid \mathcal{F}(p, t) > 0\}$) and postset ($post(t) = \{p \in \mathcal{P} \mid \mathcal{F}(t, p) > 0\}$) of a transition $t \in \mathcal{T}$, and the flow function $\mathcal{F}$ can be interpreted as a relation $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$.

We extend the model of Petri nets with special arcs, called *inhibitor arcs* [FA73; DR96a], which connect a place p and a transition t of the Petri net and only permit the firing of t when p is empty.

▶ **Definition 3 (Petri Net with Inhibitor Arcs).** A *Petri net with inhibitor arcs* is a five-tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{F}_I, In)$ with $\mathcal{P}$, $\mathcal{T}$, $\mathcal{F}$, and In as for standard P/T Petri nets. An element of the additional set $\mathcal{F}_I \subseteq \mathcal{P} \times \mathcal{T}$ is called *inhibitor arc* and connects a place to a transition.

The notion of a transition $t \in \mathcal{T}$ is *enabled* in a marking M is extended with the constraint $M(p) = 0$ for all $(p, t) \in \mathcal{F}_I$. Nothing changes for the firing rule. ◀

Graphically, inhibitor arcs are depicted as arrows equipped with a circle on their arrow tail (e.g., cp. transition t_5 in Fig. 5.5 on page 67).

3.1.2 Petri Net Unfoldings

The *unfolding* of a Petri net $\mathcal{N}$ [Eng91; EH08] exhibits all possible behavior of the Petri net by explicitly representing the *causal dependencies* (and independencies) while preserving the *concurrent* structure of the system. In an unfolding every loop in $\mathcal{N}$

is unrolled and every backward branching place is expanded by multiplying the place. Hence, every transition in the unfolding stands for the unique occurrence (instance) of a transition of $\mathcal{N}$ during an execution. This is formally defined in Definition 7. For this, we need some introductory terms and notations. Since the solving algorithms of this thesis concern only 1-bounded Petri nets, we restrict ourselves in this section also to unfoldings of 1-bounded Petri nets to keep it simpler.

Let $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ be a Petri net and $x, y \in \mathcal{P} \cup \mathcal{T}$ be two nodes. We call x a *causal predecessor* of y, written $x < y$, iff $x \mathcal{F}^+ y$ holds. We write $x \leq y$ iff $x < y$ or $x = y$. We call x, y *causally related* iff $x \leq y$ or $y \leq x$ holds. We define the *future* of a node x as all nodes *causally depending* on x, i.e., $fut^{\mathcal{N}}(x) = \{y \in \mathcal{P} \cup \mathcal{T} \mid x \leq y\}$. Two nodes $x, y \in \mathcal{P} \cup \mathcal{T}$ are in *conflict*, written $x \sharp y$, iff there is a different place $p \in \mathcal{P} \setminus \{x, y\}$ with two different transitions $t_1, t_2 \in post(p)$ in its postset with $t_1 \neq t_2$, such that $t_1 \leq x$ and $t_2 \leq y$ holds. Two nodes $x, y \in \mathcal{P} \cup \mathcal{T}$ are *concurrent* iff they are neither in conflict nor causally related. A set of places $X \subseteq \mathcal{P}$ is called *concurrent* iff all places are pairwise concurrent. We introduce a special safe Petri net, an *occurrence net*, which represents the occurrences of transitions with their conflicts and causal dependencies.

▶ **Definition 4 (Occurrence Net and Causal Net).** An *occurrence net* is a Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ with the constraints

(i) $\forall t \in \mathcal{T} : pre(t), post(t) \subseteq \mathcal{P}$, i.e, the pre- and postsets of transitions are also sets rather than multisets,

(ii) $\forall p \in \mathcal{P} : |pre(p)| \leq 1$, i.e., each place has only one ingoing arc,

(iii) $\forall t \in \mathcal{T} : \neg(t \sharp t)$, i.e., no transition is in self-conflict,

(iv) $\forall x \in \mathcal{P} \cup \mathcal{T} : \neg(x < x)$, i.e., the flow relation is acyclic,

(v) $\forall x \in \mathcal{P} \cup \mathcal{T} : |\{y \in \mathcal{P} \cup \mathcal{T} \mid y < x\}| < \infty$, i.e., the relation $<$ is well-founded, which means that it does not contain any infinitely decreasing sequence, and

(vi) $In = \{p \in \mathcal{P} \mid pre(p) = \emptyset\}$, i.e., the initial marking are exactly the places which do not have any predecessor.

An occurrence net is called a *causal net*, when further

(vii) $\forall p \in \mathcal{P} : |post(p)| \leq 1$, i.e., each place has also only one outgoing arc,

holds. ◀

If not differently stated, the elements of a superscripted Petri net $\mathcal{N}^X$ are also implicitly superscripted, i.e., $\mathcal{N}^X = (\mathcal{P}^X, \mathcal{T}^X, \mathcal{F}^X, In^X)$, and we abbreviate the pre- and postset functions by pre^X and $post^X$, respectively. For connecting the nodes of two Petri nets with respect to their local connections, the notion of a *homomorphism* on Petri nets is used.

▶ **Definition 5 (Homomorphism).** A *homomorphism* from one Petri net $\mathcal{N}_1 = (\mathscr{P}_1, \mathscr{T}_1, \mathscr{F}_1, In_1)$ to another Petri net $\mathcal{N}_2 = (\mathscr{P}_2, \mathscr{T}_2, \mathscr{F}_2, In_2)$ is a mapping $h : \mathscr{P}_1 \cup \mathscr{T}_1 \to \mathscr{P}_2 \cup \mathscr{T}_2$, which preserves the types of the nodes and the pre- and postconditions of the transitions, i.e.,

(i) $h(\mathscr{P}_1) \subseteq \mathscr{P}_2$ and $h(\mathscr{T}_1) \subseteq \mathscr{T}_2$ and

(ii) $\forall t \in \mathscr{T}_1 : h(pre^{\mathcal{N}_1}(t)) = pre^{\mathcal{N}_2}(h(t))$ and $h(post^{\mathcal{N}_1}(t)) = post^{\mathcal{N}_2}(h(t))$,

where the application of the homomorphism to a set $X \subseteq \mathscr{P}_1 \cup \mathscr{T}_1$ is defined pointwise: $h(X) = \{h(x) \mid x \in X\}$. A homomorphism h is called *initial* iff also

(iii) $h(In_1) = In_2$

holds. ◀

With Definition 4 and Definition 5 we can now introduce two additional types of Petri nets. First, the *branching process* based on an occurrence net and second, the *concurrent run* based on a causal net which formalizes a single concurrent execution of the net.

▶ **Definition 6 (Branching Process and Run).** A *branching process* $\beta = (\mathcal{N}^B, \lambda^B)$ of a Petri net $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In)$ consists of an occurrence net $\mathcal{N}^B$ and a homomorphism $\lambda^B : \mathscr{P}^B \cup \mathscr{T}^B \to \mathscr{P} \cup \mathscr{T}$ such that

$$\forall t_1, t_2 \in \mathscr{T}^B : (pre(t_1) = pre(t_2) \wedge \lambda^B(t_1) = \lambda^B(t_2)) \Rightarrow t_1 = t_2$$

holds. This means λ^B is injective on transitions with the same preset. If λ^B is initial, the branching process β is called *initial*.

A branching process $\beta' = (\mathcal{N}^R, \rho)$ of $\mathcal{N}$ with a causal net $\mathcal{N}^R$ is called *(concurrent) run* of $\mathcal{N}$. If furthermore ρ is an initial homomorphism, β' is called an *initial (concurrent) run*. ◀

As an example of a run see Fig. 4.2 on page 43. We call a Petri net $\mathcal{N}_1 = (\mathscr{P}_1, \mathscr{T}_1, \mathscr{F}_1, In_1)$ a *subnet* of a Petri net $\mathcal{N}_2 = (\mathscr{P}_2, \mathscr{T}_2, \mathscr{F}_2, In_2)$, written $\mathcal{N}_1 \sqsubseteq \mathcal{N}_2$, iff $\mathscr{P}_1 \subseteq \mathscr{P}_2$, $\mathscr{T}_1 \subseteq \mathscr{T}_2$, $\mathscr{F}_1 \subseteq \mathscr{F}_2$, and $In_1 = In_2$ holds. A branching process $\beta_1 = (\mathcal{N}_1, \lambda_1)$ is called a *subprocess* of a branching process $\beta_2 = (\mathcal{N}_2, \lambda_2)$ iff $\mathcal{N}_1$ is a subnet of $\mathcal{N}_2$ and $\lambda_1 = \lambda_{2|\mathscr{P}_1 \cup \mathscr{T}_1}$, where $h_{|X}$ restricts the domain of the function h to the set X.

Now we have everything at hand to formally introduce the *unfolding*.

▶ **Definition 7 (Unfolding).** An *unfolding* of a Petri net $\mathcal{N}$ is an initial branching process $\beta = (\mathcal{N}^U, \lambda^U)$ of $\mathcal{N}$ which satisfies

$$\lambda^U(C) = pre^{\mathcal{N}}(t) \Rightarrow \exists t^U \in \mathscr{T}^U : pre^{\mathcal{N}^U}(t^U) = C \wedge \lambda^U(t^U) = t$$

for all transitions $t \in \mathscr{T}$ and sets of *concurrent* places $C \subseteq \mathscr{P}^U$. ◀

Thus, whenever a transition of the Petri net can occur in the unfolding there is indeed a transition with the same label occurring in the unfolding. Note, an unfolding is unique up to isomorphism [Eng91]. As an example of an unfolding and a concurrent run see Fig. 9.2 on page 142.

▶ **Definition 8 (Covering Firing Sequence).** Consider a run $\beta = (\mathcal{N}^R, \rho)$ of a Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ and a finite or infinite firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ of the run $\mathcal{N}^R$ with $M_0 = In^R$. We say ζ *covers* β iff

$$(\forall p \in \mathcal{P}^R : \exists i \in \mathbb{N} : p \in M_i) \wedge (\forall t \in \mathcal{T}^R : \exists i \in \mathbb{N} : t = t_i),$$

holds. We define the set of *all* covering firing sequences of a run β by $Z(\beta)$. ◀

This means that for a covering firing sequence ζ all places and transitions in the net of the run $\mathcal{N}^R$ do appear in ζ. Note that due to the interleaving of concurrent transitions several firing sequences may cover a single run β.

3.2 Kripke Structures

We consider *Kripke structures* [Kri59] to represent the behavior of dynamic systems. A Kripke structure is a finite directed graph with its vertices representing the states or configurations of the system and its edges representing the state changes. The vertices are labeled with sets of atomic propositions with the meaning that all atomic propositions occuring in the set are satisfied in the corresponding configuration and all not occuring atomic propositions are not satisfied.

▶ **Definition 9 (Kripke Structure).** A *Kripke structure* $\mathcal{K} = (A, S, S_0, \ell, \rightarrow)$ is a five-tuple with

- a set of *atoms* A,
- a set of *states* S,
- a set of *initial states* $S_0 \subseteq S$,
- a *labeling function* $\ell : S \to 2^A$, and
- a *transition relation* $\rightarrow \subseteq S \times S$. ◀

A *(computation) path* $\pi = \pi_0\pi_1 \cdots \in S^\omega$ of a Kripke structure is an infinite sequence of states $\pi_i \in S$ with $(\pi_i, \pi_{i+1}) \in \rightarrow$ for $i \in \mathbb{N}$. We identify the ith *letter* of a path $\pi \in S^\omega$ with either π_i or $\pi(i)$. The *subpath* π^i of a path π is the path starting from $\pi(i)$, i.e., $\pi^i(j) = \pi(i + j)$ for all $j \in \mathbb{N}$. The path π is *initial* iff $\pi_0 \in S_0$. A path describes the dynamic behavior, i.e., the configuration changes of the system. For simplicity reasons a Kripke structure is assumed to have a *total* transition relation, i.e., all states have at least one successor state. We define the *language* of a Kripke structure $\mathcal{K} = (A, S, S_0, \ell, \rightarrow)$ as the prefix closure of all initial computation path, i.e., $\mathcal{L}(\mathcal{K}) = \{w \in S^* \mid w_0 \in S_0 \wedge \forall i \in \{0, \ldots, |w| - 1\} : (w_i, w_{i+1}) \in \rightarrow\}$.

Kripke structures are very commonly used for model checking systems with discrete state changes. Other related models are automata, state machines, and labeled transition systems. For example, the reachability graph of a 1-bounded Petri net (Definition 2) can be seen as a Kripke structure (modulo the totality of the transition relation) where the labels of the states are the places of the corresponding marking.

3.3 Propositional Temporal Logics

For model checking we need in addition to the *model* describing the behavior of the system, e.g., a Kripke structure, a *specification* of the property of interest which should be verified. With temporal logics we can reason about the temporal ordering of events without explicitly introducing time. This makes temporal logics especially useful for specifying concurrent systems [CGP01]. Propositional temporal logics extend classical propositional logics by introducing temporal operators to reason about timing constraints of the system. Whether these timing constraints are considered to be *linear* or *branching* results in two major directions of temporal logics. We introduce the two main representatives for these directions in the two following sections.

3.3.1 Branching-Time Temporal Logic CTL*

With CTL* [EH86] we can describe properties of *computation trees*. In Sec. 3.2 we have introduced the computation path of a Kripke structure as one single execution of the modeled system. Starting in a designated initial state s_0 and considering all computation paths starting in this state yields an infinite computation tree with finite branching. Such a tree exhibits *all* possible executions starting in s_0.

For a set of *atomic propositions* AP we define the set $\texttt{CTL}^*$ of *branching-time temporal logic (CTL*) formulas* by the following *syntax* of *state formulas* over AP

$$\Phi ::= a \mid \neg\,\Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathbf{E}\,\phi$$

where $a \in AP$, Φ, Φ_1, and Φ_2 are state formulas, and ϕ is a *path formula* of the following syntax

$$\phi ::= \Phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1 \mathbf{U}\,\phi_2$$

where Φ is a state formula and ϕ, ϕ_1, and ϕ_2 are path formulas. The *path quantifier* $\mathbf{E}$ states the existence of a path starting in the current state of the system, the *next* operator $\mathbf{X}$ states that a certain formula has to hold in the next step of the system, and the *until* operator $\mathbf{U}$ demands that a formula ψ_1 has to hold until the formula ψ_2 is finally satisfied. The operators $\mathbf{X}$ and $\mathbf{U}$ are called *temporal operators*.

We use the propositional operators $\vee$, $\rightarrow$, $\leftrightarrow$, the formulas $\texttt{true} = p \vee \neg p$, $\texttt{false} = p \wedge \neg p$ for some $p \in AP$, the path quantifier $\mathbf{A}\phi = \neg\mathbf{E}\neg\phi$, and the temporal operators $\mathbf{F}\phi = \texttt{true}\,\mathbf{U}\,\phi$, $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$, $\phi_1\mathbf{W}\phi_2 = \mathbf{G}\phi_1 \vee (\phi_1\mathbf{U}\phi_2)$, and $\phi_1\mathbf{R}\phi_2 = \neg(\neg\phi_1\mathbf{U}\neg\phi_2)$ as abbreviations. We call $\mathbf{F}$ the *finally* (or *eventually*) operator, stating that the formula will eventually hold on the current computation path, $\mathbf{G}$ the *globally* operator (or

always), stating that the formula holds in every state of the current path, **W** the *weak until* operator, stating that ϕ_1 holds until ϕ_2, but ϕ_2 is not required to hold eventually, and **R** the *release* operator, stating that ϕ_2 holds until ϕ_1 (including the first state where ϕ_1 holds) or until forever when ϕ_1 is never satisfied.

The *semantics* of CTL* is interpreted over states and over computation paths of a system. Here, we define the satisfaction relation $\models_{\texttt{CTL}^*}$ for a system modeled with a Kripke structure $\mathscr{K} = (A, S, S_0, \ell, \to)$:

$\mathscr{K}, s \models_{\texttt{CTL}^*} a$	iff	$a \in \ell(s)$
$\mathscr{K}, s \models_{\texttt{CTL}^*} \neg\, \Phi$	iff	not $\mathscr{K}, s \models_{\texttt{CTL}^*} \Phi$
$\mathscr{K}, s \models_{\texttt{CTL}^*} \Phi_1 \wedge \Phi_2$	iff	$\mathscr{K}, s \models_{\texttt{CTL}^*} \Phi_1$ and $\mathscr{K}, s \models_{\texttt{CTL}^*} \Phi_2$
$\mathscr{K}, s \models_{\texttt{CTL}^*} \mathbf{E}\, \phi$	iff	there *exists* a path π starting from s, i.e., $\pi(0) = s$ such that $\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi$

$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \Phi$	iff	$\mathscr{K}, \pi(0) \models_{\texttt{CTL}^*} \Phi$
$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \neg\, \phi$	iff	not $\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi$
$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi_1 \wedge \phi_2$	iff	$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi_1$ and $\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi_2$
$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \mathbf{X}\, \phi$	iff	$\mathscr{K}, \pi^1 \models_{\texttt{CTL}^*} \phi$
$\mathscr{K}, \pi \models_{\texttt{CTL}^*} \phi_1 \mathbf{U}\, \phi_2$	iff	there exists some $j \geq 0$ with $\mathscr{K}, \pi^j \models_{\texttt{CTL}^*} \phi_2$ and for all $0 \leq i < j$ the following holds: $\mathscr{K}, \pi^i \models_{\texttt{CTL}^*} \phi_1$

with states $s \in S$, paths π, atomic propositions $a \in A$, state formulas Φ, Φ_1, and Φ_2, and path formulas ϕ, ϕ_1, and ϕ_2. A Kripke structure $\mathscr{K}$ *satisfies* a CTL* formula Φ (denoted by $\mathscr{K} \models_{\texttt{CTL}^*} \Phi$) iff Φ is satisfied in every initial state $s_0 \in S_0$.

Computation tree logic (CTL) [CE81] is a syntactic fragment of CTL*. In CTL path quantifiers and temporal operators can never occur individually. Each path quantifier must be immediately followed by a temporal operator and each temporal operator must be directly preceded by a path quantifier. As some examples see the properties presented in Fig. 2.5 on page 23 without the operator $\mathbb{A}$.

3.3.2 Linear-Time Temporal Logic LTL

Linear-time temporal logic (LTL) [Pnu77] is a syntactic fragment of CTL*. In LTL there are no path quantifiers apart from a single leading **A** selecting all paths of the system. This universal selection of paths is directly handled in the semantics of LTL and can be intuitively seen as implicitly preceded to every LTL formula. We use different symbols for the operators in LTL than in CTL* to simplify the recognition of formulas of the different logics.

The set LTL of *linear-time temporal logic* (LTL) formulas over a set of atomic propositions AP is given by the following syntax

$$\psi ::= a \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \bigcirc \psi \mid \psi_1 \,\mathscr{U}\, \psi_2,$$

with $a \in AP$ and ψ, ψ_1 and ψ_2 LTL formulas. The *semantics* of the next ($\bigcirc$) and the until $\mathscr{U}$ operator are the same as for the corresponding CTL* operators. And also for

the abbreviations we use the operators $\Diamond$ (*eventually*), $\Box$ (*always*), $\mathscr{W}$ (*weak until*), and $\mathscr{R}$ (*release*) for the corresponding CTL* operators. For some properties specified in LTL see Example 1 on page 47.

3.4 Automata on Infinite Words and Trees

In this section we recall several automata models from standard nondeterministic automata to alternating tree automata. More details and elaborate explanations can be found for example in the following books, surveys, and articles [GTW02; PP04; KN01; CDGJ+08; Tho90; Tho97; KVW00; VW08; Kup18].

For a possibly infinite set X of elements subject to a total order $\widetilde{<}$, we define with $\langle X \rangle = \{(i, x) \in \mathbb{N} \times X \mid i = |\{x' \in X \mid x' \widetilde{<} x\}|\}$ the set sorted according to this total order. Note that if X is infinite, $\langle X \rangle$ is a total function over $\mathbb{N}$. We abbreviate $\langle X \rangle_i = \langle X \rangle(i)$ for $i \in \mathbb{N}$ (or $i \in \{0, \ldots, |X| - 1\}$ if X is finite). We assume the disjoint sets of places $\mathscr{P}$ and transitions $\mathscr{T}$ of a Petri net to have a total order (e.g., a lexicographical one).

▶ **Definition 10 (Nondeterministic Automaton).** A *nondeterministic automaton over infinite words* is a five-tuple $A = (\Sigma, Q, I, \delta, Acc)$ with

- a finite *alphabet* Σ,
- a finite set of *states* Q,
- a set of *initial states* $I \subseteq Q$,
- a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and
- an *acceptance condition* $Acc \subseteq Q^\omega$.

A *run* of A on an infinite word $w = w_0 w_1 \cdots \in \Sigma^\omega$ is an infinite sequences of states $r = r_0 r_1 \cdots \in Q^\omega$ such that

- $r_0 \in I$ and
- $(r_i, w_i, r_{i+1}) \in \delta$ for all $i \in \mathbb{N}$

holds. A run r is *accepting* iff $r \in Acc$ holds.

A nondeterministic automaton over infinite words A *accepts* an infinite input word w iff there exists an accepting run r of A on w. The set of infinite words accepted by A, i.e., $\mathscr{L}(A) = \{w \in \Sigma^\omega \mid A \text{ accepts } w\}$, is called the *language* recognized by A. ◀

For nondeterministic automata it suffices that for a nondeterministic situation in a state $q \in Q$, i.e., with at least two different edges $(q, a, q'), (q, a, q'') \in \delta$ for states $q', q'' \in Q$ with $q' \neq q''$, *some* choice leads to the acceptance of the input word. If we require that *any* choice has to lead to the acceptance of the input word we obtain a *universal automaton*. For *alternating automata* these types of choices are combined by allowing positive Boolean combinations over successor states such that the disjunction expresses the nondeterministic choice and the conjunction the universal choice.

▶ **Definition 11 (Positive Boolean Formulas).** Given a set X, the set $\mathbb{B}^+(X)$ are all *positive Boolean formulas* over the set X, i.e., `true`, `false`, and all combination of elements in X using the conjunction $\wedge$ and the disjunction $\vee$.

A subset $Y \subseteq X$ *satisfies* a positive Boolean formula $\theta \in \mathbb{B}^+(X)$, denoted by $Y \models \theta$, iff assigning `true` to all elements in Y and `false` to all elements in $X \setminus Y$ results in the satisfaction of θ. ◀

Before introducing alternating word automata, we define the structure of trees because a run of an alternating automaton is a tree rather than a word as for nondeterministic automata.

▶ **Definition 12 (Tree, Σ-labeled Tree, D-tree).** A *tree* is a prefix-closed set $T \subseteq \mathbb{N}^*$ such that

$$\forall n \in \mathbb{N}^* \, \forall c \in \mathbb{N} : n \cdot c \in T \implies n \in T \wedge \forall 0 \leq c' < c : n \cdot c' \in T$$

holds. The elements $n \in T$ are called *nodes* and the empty word $\varepsilon \in T$ is called the *root* of T. For a node $n \in T$ we call the nodes $n \cdot c \in T$ for $c \in \mathbb{N}$ the *successors* or *children* of n and collect them in $\texttt{children}(n)$. For a node $n \in T$ the number of children is called the *degree* of n and is denoted by $d(n)$. A node without children is called a *leaf*. An *infinite tree* is a tree without leafs. A *path* $\pi \subseteq T$ of a tree contains the root, i.e., $\varepsilon \in \pi$, and for all nodes $n \in \pi$ it either is a leaf or has a unique child $n \cdot c \in \pi$ for $c \in \mathbb{N}$.

For a given alphabet Σ a Σ-*labeled tree* is a pair (T, v) containing a tree T and a labeling function $v : T \to \Sigma$ labeling each node of the tree with a letter from the alphabet Σ.

Given a finite set $D \subset \mathbb{N}$ of *degrees*, a D-*tree* (T, v) is a Σ-labeled tree where all nodes $n \in T$ have a degree in D. ◀

An alternating word automaton still takes as input an infinite word, but can branch into several successor states simultaneously during its execution. Intuitively, we can think of new instances of the automata are created and simultaneously executed. Thus, a run of an alternating word automaton on an infinite word is a tree labeled with the current states of these copies. An example of an alternating word automaton is presented in Fig. 5.4 on page 61.

▶ **Definition 13 (Alternating Word Automaton).** An *alternating automaton over infinite words*, or short *alternating word automaton (AWA)*, is a five-tuple $\mathcal{A} = (\Sigma, Q, I, \delta, Acc)$ with

- a finite *alphabet* Σ,
- a finite set of *states* Q,
- a set of *initial states* $I \subseteq Q$,
- a *transition function* $\delta : Q \times \Sigma \to \mathbb{B}^+(Q)$, and
- an *acceptance condition* $Acc \subseteq Q^\omega$.

A *run* of $\mathscr{A}$ on an infinite word $w = w_0 w_1 \cdots \in \Sigma^\omega$ is a Q-labeled tree $r = (T, v)$ such that

- $v(\varepsilon) \in I$ and
- for all $n \in T$ with $v(n) = q$, the set of the children's labels

$$Y = \{v(n') \in Q \mid n' \in \texttt{children}(n)\}$$

satisfies the successor formula $\delta(q, w_{|n|}) = \theta$, i.e., $Y \models \theta$,

holds. A run $r = (T, v)$ of $\mathscr{A}$ is *accepting* iff for *every infinite* path $\pi \subseteq T$ of T the infinite word of its labels is accepted, i.e., $v(\pi_0)v(\pi_1)\cdots \in Acc$.

An alternating word automaton $\mathscr{A}$ *accepts* an infinite input word w iff there exists an accepting run $r = (T, v)$ of $\mathscr{A}$ on w. The set of infinite words accepted by $\mathscr{A}$, i.e., $\mathscr{L}(\mathscr{A}) = \{w \in \Sigma^\omega \mid \mathscr{A} \text{ accepts } w\}$, is called the *language* recognized by $\mathscr{A}$. ◀

Even though the run of an alternating word automaton is a tree, the input is still an infinite word. This changes for *tree automata*. A nondeterministic automaton over infinite trees takes as input a Σ-labeled tree that has no leaves.

▶ **Definition 14 (Tree Automaton).** For a given finite set of possible degrees $D \subset \mathbb{N}$ a *nondeterministic automaton over infinite trees*, or short *tree automaton (TA)*, is a five-tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$ with

- a finite *alphabet* Σ,
- a finite set of *states* Q,
- an *initial state* $q_0 \in Q$,
- a *transition relation* $\delta \subseteq Q \times \Sigma \times D \times \bigcup_{k \in D} Q^k$ with $k = m$ for all $(q, \sigma, k, q_1, \ldots, q_m) \in \delta$,
- an *acceptance condition* $Acc \subseteq Q^\omega$.

A *run* of $\mathcal{A}$ on an infinite Σ-labeled D-tree (T, v) is a Q-labeled D-tree $r = (T_r, v_r)$ such that

- $v_r(\varepsilon) = q_0$, and
- for all $n \in T_r$ with $v_r(n) = q$, there exists a successor $(q_0, \ldots, q_{d(n)-1}) \in \delta(q, v(n), d(n))$ such that for all $0 \leq i < d(n)$ children in T we have $n \cdot i \in T_r$ and $v_r(n \cdot i) = q_i$,

holds. Intuitively, we label the input tree with states of the automaton according to the transition relation. In particular, this means $T = T_r$. A run $r = (T_r, v_r)$ of $\mathcal{A}$ is *accepting* iff for *every* path $\pi \subseteq T_r$ of T_r the infinite word of its labels is accepted, i.e., $v_r(\pi_0)v_r(\pi_1)\cdots \in Acc$. Note that all paths are infinite due to the infinite input tree.

A tree automaton $\mathcal{A}$ *accepts* an infinite Σ-labeled D-tree (T, v) iff there exists an accepting run $r = (T_r, v_r)$ of $\mathcal{A}$ on (T, v). The set of infinite Σ-labeled D-trees accepted by $\mathcal{A}$, i.e., $\mathscr{L}(\mathcal{A}) = \{(T, v)\ \Sigma\text{-labeled } D\text{-tree} \mid \mathcal{A} \text{ accepts } (T, v)\}$, is called the *language* recognized by $\mathcal{A}$. ◀

Alternating tree automata again generalize the transition relation to allow for positive Boolean combinations of successor states.

▶ **Definition 15 (Alternating Tree Automaton).** For a given finite set of possible degrees $D \subset \mathbb{N}$ an *alternating automaton over infinite trees*, or short *alternating tree automaton (ATA)*, is a five-tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$ with

- a finite *alphabet* Σ,
- a finite set of *states* Q,
- an *initial state* $q_0 \in Q$,
- a *transition relation* $\delta : Q \times \Sigma \times D \to \mathbb{B}^+(\mathbb{N} \times Q)$ with $\delta(q, \sigma, k) \in \mathbb{B}^+(\{0, \dots, k-1\} \times Q)$ for every $k \in D$, and
- an *acceptance condition* $Acc \subseteq Q^\omega$.

A *run* of $\mathcal{A}$ on an infinite Σ-labeled D-tree (T, v) is a $\mathbb{N}^* \times Q$-labeled D-tree $r = (T_r, v_r)$ such that

- $v_r(\varepsilon) = (\varepsilon, q_0)$, and
- for all $n \in T_r$ with $v_r(n) = (x, q)$ and $\delta(q, v(x), d(x)) = \theta$, there is a set $Y = \{(c_0, q_0), \dots, (c_m, q_m)\} \subseteq \{0, \dots, d(x) - 1\} \times Q$ such that $Y \models \theta$ and for all $0 \leq i \leq m$ we have $n \cdot i \in T_r$ and $v_r(n \cdot i) = (x \cdot c_i, q_i)$,

holds. Thus, each node $n \in T_r$ with $v_r(n) = (x, q)$ corresponds to the node $x \in T$ and represents a copy of the automaton reading x and visiting state q. Note that several nodes of T_r can correspond to a node $x \in T$. A run $r = (T_r, v_r)$ of $\mathcal{A}$ is *accepting* iff for *every infinite* path $\pi \subseteq T_r$ of T_r the infinite word of its labels is accepted, i.e., $v_r(\pi_0)v_r(\pi_1) \cdots \in Acc$.

A tree automaton $\mathcal{A}$ *accepts* an infinite Σ-labeled D-tree (T, v) iff there exists an accepting run $r = (T_r, v_r)$ of $\mathcal{A}$ on (T, v). The set of infinite Σ-labeled D-trees accepted by $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \{(T, v)\ \Sigma\text{-labeled } D\text{-tree} \mid \mathcal{A} \text{ accepts } (T, v)\}$, is called the *language* recognized by $\mathcal{A}$. ◀

Intuitively, we can see the runs of the given automata with increasing complexity as follows: A run of a standard nondeterministic automaton on an infinite word is just the word of the corresponding states of the automaton. A run of the alternating automaton on an infinite word may branch into several states for each read letter of the input word. Thus, we obtain a tree labeled with the corresponding states of the automaton. The input for a nondeterministic tree automaton is already an infinite tree. Thus, the run on this tree is the same tree but now labeled with the corresponding states of the automaton. A run of an alternating tree automaton is a tree, but in this case again due to creating copies of the automaton for the alternation. Thus, given an input letter, the automaton may branch into several instances. Each node of the run is not only labeled with the current state of the instance of the automaton, but also with the corresponding

node of the input tree. Thus, we may be in several states for one node of the input tree. Example 3 on page 58 contains an example for an alternating tree automaton.

So far we defined the acceptance conditions of the automata only abstractly as sets of infinite sequences of states. We now state some examples of *acceptance conditions* for infinite words that are finitely represented.

▶ **Definition 16 (Acceptance Conditions).** The function $\texttt{Inf} : Q^\omega \to 2^Q$ with $\texttt{Inf}(w) = \{a \in Q \mid \forall n \in \mathbb{N} : \exists m \geq n : w_m = a\}$ collects all letters which *occur infinitely often* in an infinite word w over the alphabet Q. For a finite set $A \subseteq Q$ we define the

Büchi condition: $\texttt{BUCHI}(A) = \{w \in Q^\omega \mid \texttt{Inf}(w) \cap A \neq \emptyset\}$ collecting all infinite words that visit some state of A infinitely often, and the

Co-Büchi condition: $\texttt{COBUCHI}(A) = \{w \in Q^\omega \mid \texttt{Inf}(w) \cap A = \emptyset\}$ collecting all infinite words that visit none of the states of A infinitely often.

For a finite set of pairs $A_R = \{(I_0, F_0), \ldots, (I_n, F_n)\} \subseteq 2^Q \times 2^Q$ we define the

Rabin condition: $\texttt{RABIN}(A_R) = \{w \in Q^\omega \mid \exists i \in \{0, \ldots, n\} : \texttt{Inf}(w) \cap I_i \neq \emptyset \wedge \texttt{Inf}(w) \cap F_i = \emptyset\}$ collecting all infinite words that visit some state of I_i infinitely often *and* none of the states of F_i infinitely often for some pair $(I_i, F_i) \in A_R$.

We call n the *index* of the automaton with a Rabin condition. ◀

We often use abbreviations like NBA for a nondeterministic Büchi automaton, ABA for an alternating Büchi word automaton (or ABWA, when we want to stress that it is a word rather than a tree automaton), ATA for an alternating tree automaton when we do not want to specify the acceptance condition, or any other combination of these abbreviations.

4 Petri Nets with Transits

In this chapter we introduce the new model for model checking asynchronous distributed systems with data flows. We call this model *Petri nets with transits*, as it refines the flow relation of a Petri net with a so-called *transit relation* to distinguish between the *control* flow of the system and how the *data* flows through the system.

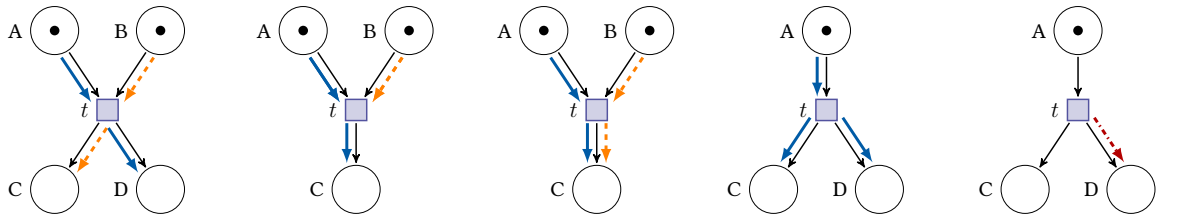
In Chap. 2 we have introduced two application areas of Petri nets with transits. First, a concurrent overlapping update in *software-defined networking* is modeled. The concurrent update (with possibly sequential sub-updates) is modeled via tokens moving through the Petri net, whereas the data flow is modeled by the transits. Second, door policies for the *access control in physical spaces* are modeled. Updating the door policies is again modeled via the tokens whereas the people roaming through the building are modeled at the data flow level. In this chapter we formally define this model in Sec. 4.1 and introduce *data flow chains* and *data flow trees* in Sec. 4.2 which are used to check the correctness of the data flow in Chap. 5 and Chap. 6. This chapter is based on the publications [FGHO19a; FGHO20c].

4.1 The Model

A *Petri net with transits* [FGHO19a] extends a Petri net $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, In)$ by refining the (*control*) flow relation $\mathscr{F}$ with a *transit relation* Υ for modeling the *data* flow. See Fig. 2.3 on page 21 and Fig. 2.4 on page 22 for some examples.

▶ **Definition 17 (Petri Net with Transits).** A *Petri net with transits* is a five-tuple $\mathcal{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ with

- a possibly infinite set of *places* $\mathscr{P}$,
- a possibly infinite set of *transitions* $\mathscr{T}$ (disjoint to $\mathscr{P}$, i.e., $\mathscr{T} \cap \mathscr{P} = \emptyset$),
- a *flow function* $\mathscr{F} : (\mathscr{P} \times \mathscr{T}) \cup (\mathscr{T} \times \mathscr{P}) \to \mathbb{N}$ assigning each arc connecting places and transitions a non-negative integer *arc weight* (or *arc multiplicity*),
- a *transit relation* $\Upsilon : \mathscr{T} \to 2^{(\mathscr{P} \cup \{\triangleright\}) \times \mathscr{P}}$ which defines for each transition $t \in \mathscr{T}$ a relation $\Upsilon(t)$ of type $\Upsilon(t) \subseteq (pre(t) \cup \{\triangleright\}) \times post(t)$, where the symbol $\triangleright$ denotes the *start* of a new data flow, and
- an *initial marking* $In : \mathscr{P} \to \mathbb{N}$, defining the number of tokens initially residing on the places.



(a) *Refinement* **(b)** *Termination* **(c)** *Merging* **(d)** *Branching* **(e)** *New Creation*

Fig. 4.1: *Example patterns how the transit relation can be used in Petri nets with transits to refine the control token flow.*

Thus, the elements $\mathscr{P}$, $\mathscr{T}$, $\mathscr{F}$, and In are as for standard Petri nets. Properties like k-boundedness, plain, etc. can be directly transferred from Petri nets to Petri nets with transits. The *postset regarding* Υ of a place $p \in \mathscr{P}$ and a transition $t \in post(p)$ is defined by $post^{\Upsilon}(p,t) = \{p' \in \mathscr{P} \mid (p,p') \in \Upsilon(t)\}$. ◀

We use a *color coding* for the *visual representation* of the transits $\Upsilon(t)$ of a transition $t \in \mathscr{T}$. The *black arrows* represent, as already in standard Petri nets, the usual *control flow*. The *colored arrows* are locally bound to a transition and represent the transits of the *data flow*. For example in Fig. 4.1a the transit $(\mathtt{A},\mathtt{D}) \in \Upsilon(t)$ is depicted as solid blue arrows and the transit $(\mathtt{B},\mathtt{C}) \in \Upsilon(t)$ as orange dashed arrows. The creation of new data is depicted as a colored outgoing arrow without a correspondingly colored ingoing arrow (cp. Fig. 4.1e for the transits $\Upsilon(t) = \{(\triangleright,\mathtt{D})\}$). The termination of a data flow can either be depicted by not having any colored outgoing arrow of the place, or by having a colored arrow but without using the same color for an outgoing arrow of the corresponding transition. For example in Fig. 4.1b the transits of t are $\Upsilon(t) = \{(\mathtt{A},\mathtt{C})\}$. The orange dashed arrow from place B to transition t depicted in Fig. 4.1b is only to stress the termination of the data flow in B when firing t and could also be omitted. To have a clearer overview, all ingoing arrows of a transition must be uniquely colored and each ingoing control arc can only correspond to at most one colored arrow. However, the outgoing control arcs of a transition can be related to several colored arrows (cp. Fig. 4.1c with $\Upsilon(t) = \{(\mathtt{A},\mathtt{C}),(\mathtt{B},\mathtt{C})\}$) and also several outgoing arrows can have the same color (cp. Fig. 4.1d with $\Upsilon(t) = \{(\mathtt{A},\mathtt{C}),(\mathtt{A},\mathtt{D})\}$).

In Chap. 2 our motivating examples always move the data flow along *persistent* control tokens. This means the control tokens flicker for a moment when a transition is fired but are then placed back into their previously occupied places and only the data flow is extended (cp. for example Fig. 2.3). The examples depicted in Fig. 4.1 show another approach how to use Petri nets with transits. This kind of modeling is especially interesting for the extension of Petri nets with transits and the synthesis approach presented in Part II. Here the data flow is led along *moving* control tokens. By that we can consider processes together with their local data flow and different level of knowledge about the data flow of the others:

Refinement: Depicted in Fig. 4.1a. The two processes p_1 and p_2, initially residing in place A and place B, respectively, have their own local data. Assume that the

data of process p_1 is already corrupted, the data of process p_2 is still accurate, and the data of any process reaching place D gets corrupted. Without the refinement of the flow relation we could never know whether there is still a well-functioning process after firing transition t.

Termination Depicted in Fig. 4.1b. Initially there are two processes p_1 and p_2 residing in place A and place B, respectively. Without the refinement of the flow relation we do not know whether process p_1 (in case $(\mathsf{A},\mathsf{C}) \notin \Upsilon(t)$) or process p_2 (in case $(\mathsf{B},\mathsf{C}) \notin \Upsilon(t)$) with its local data flow terminates when firing transition t. Even both could have terminated (in case $(\mathsf{A},\mathsf{C}),(\mathsf{B},\mathsf{C}) \notin \Upsilon(t)$) and a complete new process, possibly with a fresh data flow ($(\triangleright,\mathsf{C}) \in \Upsilon(t)$), could have been created. Depending on this choice, the process in place C is either corrupted or not, in case one of the processes p_1 or p_2 is already corrupted before firing transition t.

Merging: Depicted in Fig. 4.1c. In this scenario we can consider that the firing of transition t creates a new process based on the data flow of both of the two processes p_1 and p_2, initially residing in place A and place B, respectively. Thus, is already one process corrupted, also the process residing in place C is corrupted.

Branching: Depicted in Fig. 4.1d. When transition t fires we can consider that a copy of process p_1 (initially residing in place A) is created containing the same information as p_1. After the firing of transition t the data flow of these processes are independent. But when the process in p_1 is already corrupted, only corrupted processes exists after firing t.

New Creation: Depicted in Fig. 4.1e. A new process with also fresh data is created when firing transition t. This pattern can be used for failure recovery. No matter whether the process initially residing in place A is corrupted, after firing t the process residing in place D is unaffected of this corruption.

Intuitively, $(\triangleright, q) \in \Upsilon(t)$ defines the start of a new data flow in place $q \in \mathscr{P}$ via transition $t \in \mathscr{T}$. Hence, whenever t fires new data is given into the system at place q. With $(p, q) \in \Upsilon(t)$ we define that whenever transition $t \in \mathscr{T}$ fires all data in place $p \in \mathscr{P}$ *transits* via t to place q. Transits allow us to specify where the data flow is moved forward, split, and merged, where it ends, and where data is newly created. The data flow can be of infinite length and at any point in time (possibly restricted by the control) new data can enter the system at different locations. The data flow is a local property of each distributed component but can possibly be shared via joint transitions to other components. We make these concepts formal in the next section by defining data flow chains and data flow trees.

4.2 Data Flow Chains and Data Flow Trees

Formally, the semantics of the transits is defined via the runs of the Petri net with transits. A *branching process of a Petri net with transits* $\mathscr{N}_T$ is the branching process $\beta^B = (\mathscr{N}_T^B, \lambda^B)$ of the corresponding Petri net $\mathscr{N}$ where the transit relation Υ is

lifted to $\mathcal{N}_T^B$. Hence, $\Upsilon^B : \mathscr{T}^B \to 2^{(\mathscr{P}^B \cup \{\rhd\}) \times \mathscr{P}^B}$ with $\Upsilon^B(t) \subseteq (pre^B(t) \cup \{\rhd\}) \times post^B(t)$ for all $t \in \mathscr{T}^B$ is defined by $(\rhd, q) \in \Upsilon^B(t) \iff (\rhd, \lambda^B(q)) \in \Upsilon(\lambda^B(t))$ for all $q \in \mathscr{P}^B$, and $(p, q) \in \Upsilon^B(t) \iff (\lambda^B(p), \lambda^B(q)) \in \Upsilon(\lambda^B(t))$ for all $p, q \in \mathscr{P}^B$. This means, whenever there is a transit in $\mathcal{N}_T$, there is also a transit in Υ^B for the corresponding elements. By that we obtain notions of *runs* and *unfoldings* for Petri nets with transits as for Petri nets described in Sec. 3.1.2.

We define *flow chains* and *flow trees* by following the transits of a given run to specify the flow of the data in a system modeled as Petri net with transits.

▶ **Definition 18 (Flow Chain).** A (*data*) *flow chain* of a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$ is a *maximal* sequence $\xi = t_0, p_0, t_1, p_1, t_2 \ldots$ (or $\xi = t_0, p_0, t_1, \ldots, t_n, p_n$ if it is finite) of connected places and transitions of $\mathcal{N}_T^R$, i.e., $(t_i, p_i), (p_i, t_{i+1}) \in \mathscr{F}^R$ for all $i \in \mathbb{N}$ (or $i \in \{1, \ldots n\}$ if ξ is finite), with

(I) $(\rhd, p_0) \in \Upsilon^R(t_0)$, i.e., the first transition starts the chain in the first place,

(con) $(p_{i-1}, p_i) \in \Upsilon^R(t_i)$, i.e., all elements are connected via transits, and

(max) if ξ is finite, there is no transition $t \in \mathscr{T}^R$ and place $q \in \mathscr{P}^R$ such that $(p_n, q) \in \Upsilon^R(t)$.

The set of all flow chains of a run β is denoted by $\Xi(\beta)$.

A *flow chain suffix* $\xi' = t_0, p_0, t_1, p_1, t_2 \ldots$ of a run β requires constraints **(con)**, **(max)**, and in addition to the constraint in **(I)** permits that the chain has already started, i.e., **(I')** $(\rhd, p_0) \in \Upsilon^R(t_0) \vee \exists p \in \mathscr{P}^R : (p, p_0) \in \Upsilon^R(t_0)$. Thus, each flow chain is also a flow chain suffix. ◀

Figure 4.2 shows a finite run $\beta = (\mathcal{N}_T^R, \rho)$ of the Petri net with transits $\mathcal{N}_T$ depicted in Fig. 2.4 on page 22. The names of the run's nodes are abbreviations of the corresponding nodes in the Petri net with transits. For example $\rho(e) = emergency$ and $\rho(e_h^i) = enter_{hall}$ for $i \in \{0, 1\}$. Remember that data flow in this example models the movement of people. This run begins by moving all people who are currently in the corridor into the hall (transition e_h^0). Since no data flow has been generated yet, there are currently no people in the corridor. Therefore, no "data" is transmitted. After that, people enter the corridor (transition e_c^0 with $\rho(e_c^0) = enter_{corr}$) and all people currently in the hall are moved into the corridor ($l_{h\to c}^0$ with $\rho(l_{h\to c}^0) = leave_{h\to c}$). Again, since no people are currently in the hall, nothing is transited from the hall, but people already in the corridor, stay in the corridor. Then, all people in the corridor are moved into the hall (transition e_h^1) and new people enter the corridor (transition e_c^1). Now, the door policy update happens and all people in the rooms are informed of the emergency (transition e). Finally, the people can leave the hall either to the corridor or into the yard (transition $l_{h\to[c,y]}^0$ with $\rho(l_{h\to[c,y]}^0) = leave_{h\to[c,y]}$). This runs has three different data flow chains. The first chain starts when transition e_c^0 fires and follows the gray shaded path. In the last transition, transition $l_{h\to[c,y]}^0$, the data flow splits into two branches, yielding the first and the second data flow chain: $\xi_1 = e_c^0, corr^2, l_{h\to c}^0, corr^3, e_h^1, hall^3, e, hall^4, l_{h\to[c,y]}^0, corr^7$

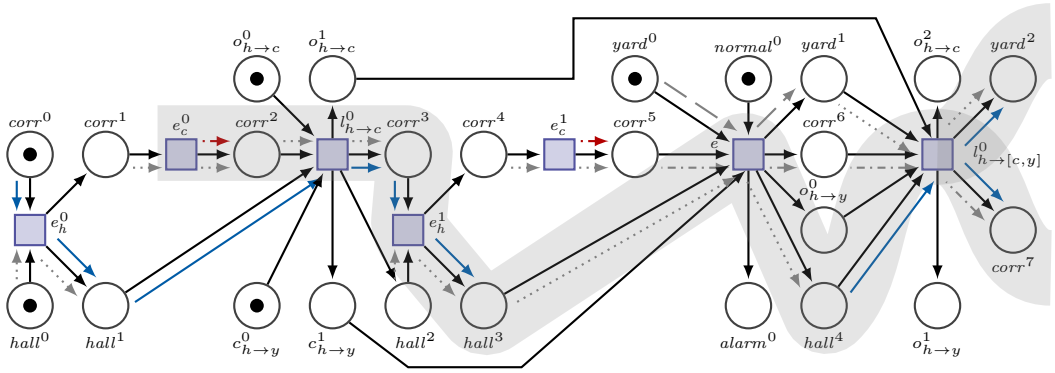


Fig. 4.2: *A finite run of the Petri net with transits from Fig. 2.4 on page 22 with two data flow trees is depicted. The first one is indicated by the gray shaded area. The second one starts from the firing of transition e^1_c and contains no branching.*

and $\xi_2 = e^0_c, corr^2, l^0_{h\to c}, corr^3, e^1_h, hall^3, e, hall^4, l^0_{h\to[c,y]}, yard^2$. Both flow chains represent the people entering the building in the corridor and then moving to the lecture hall. When the alarm is triggered the people leave the hall to the corridor (flow chain ξ_1) or to the yard (flow chain ξ_2). The third data flow chain is created when transition e^1_c fires. For this run, these people stay the whole time inside the corridor: $\xi_3 = e^1_c, corr^5, e, corr^6, l^0_{h\to[c,y]}, corr^7$.

As seen for transition $l^0_{h\to[c,y]}$ with $\rho(l^0_{h\to[c,y]}) = leave_{h\to[c,y]}$, a transition can split the data flow resulting in a branching behavior. This means for one place $p \in \mathscr{P}^R$ there can be two different places $q, q' \in \mathscr{P}^R$ such that $(p,q),(p,q') \in \Upsilon^R(t)$ holds. We can collect these data flows also in a tree structure. The *data flow trees* of a run represents all branching behavior in the transitions of the run with respect to its transits.

▶ **Definition 19 (Flow Tree).** For a given run $\beta = (\mathscr{N}^R_T, \rho)$ of a Petri net with transits $\mathscr{N}_T$ there is for each $t_0 \in \mathscr{T}^R$ and place $p_0 \in \mathscr{P}^R$ with $(\triangleright, p_0) \in \Upsilon^R(t_0)$, a $\mathscr{T}^R \times \mathscr{P}^R$-labeled (*data*) *flow tree* $\tau = (T, v)$ over degrees $\mathscr{D} = \{|post^{\Upsilon^R}(p,t)| \mid p \in \mathscr{P}^R \wedge t \in post^R(p)\}$ with

1. $v(\epsilon) = (t_0, p_0)$ for the root ϵ, and
2. if $n \in T$ with $v(n) = (t,p)$ then for the unique transition $t' \in post^R(p)$ (if existent) we have for all $0 \leq i < |post^{\Upsilon^R}(p,t')|$ that $n \cdot i \in T$ with $v(n \cdot i) = (t', q)$ for $q = \langle post^{\Upsilon^R}(p,t')\rangle_i$

where $\langle post^{\Upsilon^R}(p,t')\rangle_i$ is the i-th value of the ordered list $\langle post^{\Upsilon^R}(p,t')\rangle$. ◀

In Fig. 4.2 one of the two data flow trees of the depicted run is indicated by the gray shaded area. The other one is only a non-branching tree corresponding to the data flow chain ξ_3. Figure 4.3 gives a schematic overview over the firing of the transitions in the Petri net with transits and the data flow trees of the corresponding run depicted in Fig. 4.2. This shows that in case no data flow is created, the firing of transitions does not have any effect on the data flow trees, even if they would transit the data

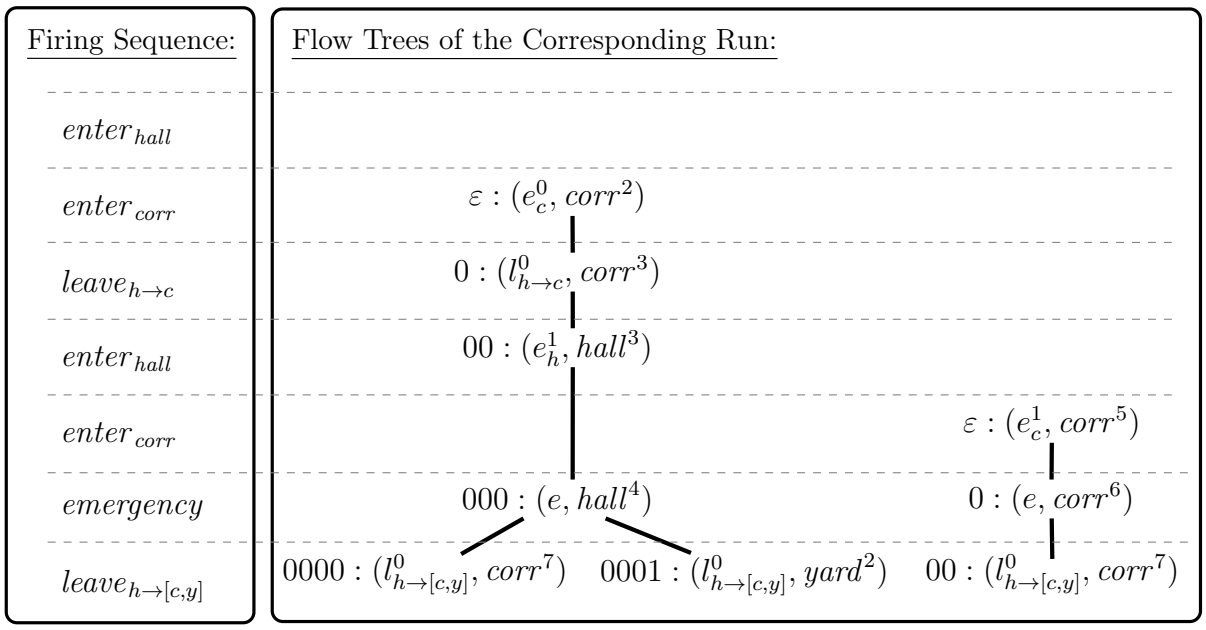


Fig. 4.3: *A schematic overview of the correlation between the firing sequence of the Petri net with transits depicted in Fig. 2.4 on page 22 and the data flow trees of the corresponding run depicted in Fig. 4.2. The nodes of the trees are depicted on the left of the colon, where the corresponding labels are depicted on the right of the colon.*

flow in a different run (cp. the first occurring of transition $enter_{hall}$). Furthermore, concurrent transitions not transiting the data flow of the current place of a specific chain (or tree) does not occur in the data flow chain (or tree). The second occurring of the transition $enter_{corr}$ in the firing sequence does not effect the first tree. So data flow chains (and trees) start at different points in time regarding the firing sequence and more crucially, they proceed in their own timeline omitting concurrent transition not involving the own data flow. In the next chapter we elaborate more on this topic about different timelines for the global system and each local data flow.

5 Model Checking Petri Nets with Transits against Flow-CTL*

In this chapter we introduce a model checking routine for Petri nets with transits and Flow-CTL*. In classical model checking there are three components [CHVB18]. First, there is the *modeling* part. We need to model our system which we want to analyze in an adequate formalism. For distributed asynchronous systems with local data flows we introduced Petri nets with transits in Chap. 4.

Second, there is the *specification* language to describe the system's correctness properties. For that we introduce in this chapter the temporal logic Flow-CTL*. In Sec. 2.2 we already introduced some example specifications formalized in Flow-CTL* (e.g., in Fig. 2.5). Flow-CTL* is composed of two parts. First, there is the LTL part to reason about the global configuration of the system. Here, we consider the runs of the Petri net with transits and check the formula for all firing sequences covering this run. This semantics is formally introduced in Sec. 5.1. Second, there is the CTL* part to reason about the local data flow of the runs. Here, we consider the data flow chains or the data flow chain suffixes to check each of the local data flow formulas on all data flow trees of the run. In the example depicted in Fig. 4.3 we can already see that the local data flow of the processes is independent of the firing of concurrent transitions. Thus, each data flow chain can be considered having its own timeline. As a schematic overview of possible timelines see Fig. 5.1. With τ we depict the global timeline corresponding to the firing sequence of the run. For each data flow chain ξ_i of the possible infinitely many, the steps where a transit extends the data flow are depicted as filled time steps. The CTL* formula is evaluated only on the filled time steps of a data flow chain leading to a separated timeline for each chain. Flow-CTL* is formally introduced in Sec. 5.2.

The third component for model checking are the *algorithms* constituting the decision procedure used to verify the correctness of the system, i.e., checking whether the Petri net with transits satisfies the Flow-CTL* formula, and if not, providing a counterexample. In Sec. 5.3, we reduce the model checking problem of a 1-bounded Petri nets with transits against Flow-CTL* via automata constructions to the model checking problem of Petri nets against LTL. The reduction consists of three steps: First, each data flow subformula of the given Flow-CTL* formula is represented by a finite Petri net via an alternating tree automaton, an alternating word automaton, and a nondeterministic Büchi automaton, to guess and then to verify a counterexample tree (Sec. 5.3.1). Second, the original net for the control subformula of the Flow-CTL* formula and the nets for the data flow subformulas are connected in sequence. A fully formal construction of the Petri net can be found in Sec. 5.4 and, to gain an easier access to the construction, a more

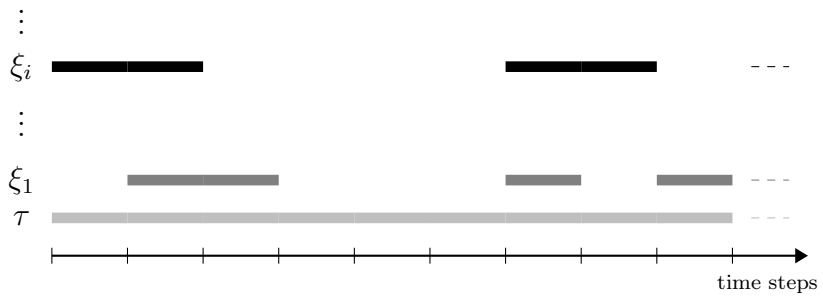


Fig. 5.1: *A possible sequence of the global timeline τ and the timelines of the possible infinite number of flow chains ξ_i. A filled time step for a timeline of a flow chain indicates that the fired transition has a transit which extends this flow chain.*

textual version is presented in Sec. 5.3.2. Third, an LTL formula encodes the control subformula, the acceptance conditions of the automata in terms of the corresponding subnets for the data flow subformulas, and the correct skipping of the subnets in the sequential order to comply with each timeline (cp. Fig. 5.1). This step is presented in Sec. 5.3.3. Together, this results in a model checking problem of 1-bounded Petri nets and LTL formulas. Due to the automata construction for the CTL* formulas, this results in a triple-exponential time algorithm for model checking a 1-bounded Petri net with transits $\mathcal{N}_T$ against a Flow-CTL* formula φ, or a double-exponential time algorithm for the fragment of Flow-CTL formulas, in the size of $\mathcal{N}_T$ and φ. In Chap. 6 we present a single-exponential time algorithm for the fragment of Flow-LTL formulas.

This chapter is based on [FGHO20c] and the corresponding full version [FGHO20d].

5.1 LTL on Petri Net Unfoldings

In Flow-CTL* we reason about the two different layers of a Petri net with transits, i.e., the control layer and the data flow layer. We use LTL to reason about the global configuration of the system (the control layer) and use CTL* to reason about the data flows of the individual processes. The flow chains and trees of the data flow layer are defined on the runs of the Petri net with transits. Hence, we define the semantics of CTL* for Petri nets with transits on the runs rather than the reachability graph.

Usually, in Petri net model checking against temporal logics (see, e.g., the annual model checking contest [KBGH+21]), the formulas are restricted to places. This means that the actual firing of a transition cannot be stated. In this thesis we are also interested in stating fairness and maximality assumptions about the firing of transitions (cp. Sec. 2.1) and do not want to hard-code these assumptions into the algorithms. Thus, we consider $AP = \mathcal{P} \cup \mathcal{T}$ as atomic propositions. By that we have to decide whether the firing of a transition t belong to the state the transition t led to or to the state where t fired. In this chapter we consider the *ingoing semantics*, i.e., a state of the system is the current marking together with the transition used to *enter* the marking. Furthermore, we *stutter* in the last marking of a finite firing sequence to obtain an infinite semantics.

A *trace* is a mapping $\sigma : \mathbb{N} \to X$ to some set X. The i-th suffix $\sigma^i : \mathbb{N} \to X$ is a trace defined by $\sigma^i(j) = \sigma(j + i)$ for all $j \in \mathbb{N}$.

▶ **Definition 20 (Ingoing Semantics – Firing Sequences).** Given a run $\beta = (\mathcal{N}^R, \rho)$ of a Petri net or Petri net with transits $\mathcal{N}$. To a (finite or infinite) *covering firing sequence* $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ of β, we associate a *trace* $\sigma_R(\zeta) : \mathbb{N} \to 2^{AP}$ with

$$\sigma_R(\zeta)(i) = \begin{cases} \rho(M_0) & \text{for } i = 0 \\ \{\rho(t_{i-1})\} \cup \rho(M_i) & \text{otherwise} \end{cases}$$

if ζ is infinite and

$$\sigma_R(\zeta)(i) = \begin{cases} \rho(M_0) & \text{for } i = 0 \\ \{\rho(t_{i-1})\} \cup \rho(M_i) & \text{for } 0 < i \leq n \\ \rho(M_n) & \text{otherwise} \end{cases}$$

if $\zeta = M_0[t_0\rangle \cdots [t_{n-1}\rangle M_n$ is finite. ◀

Hence, a trace of a covering firing sequence is an infinite sequence of states collecting the current marking and ingoing transition of $\mathcal{N}$, which stutters on the last marking for finite sequences.

We evaluate an LTL formula $\psi \in \texttt{LTL}$ on a Petri net or Petri net with transits $\mathcal{N}$ over the traces of the covering firing sequences of its runs.

▶ **Definition 21 (LTL on Petri Net Runs).** Given a Petri net or Petri net with transits $\mathcal{N}$ and an LTL formula $\psi \in \texttt{LTL}$. The *semantics* of LTL on runs is defined as follows:

$$\begin{array}{rll}
\mathcal{N} \models_{\texttt{LTL}} \psi & \text{iff} & \text{for all runs } \beta \text{ of } \mathcal{N} : \ \beta \models_{\texttt{LTL}} \psi \\
\beta \models_{\texttt{LTL}} \psi & \text{iff} & \text{for all firing sequences } \zeta \text{ covering } \beta : \ \sigma_R(\zeta) \models_{\texttt{LTL}} \psi \\
\sigma_R(\zeta) \models_{\texttt{LTL}} a & \text{iff} & a \in \sigma_R(\zeta)(0) \\
\sigma_R(\zeta) \models_{\texttt{LTL}} \neg\psi & \text{iff} & \text{not } \sigma_R(\zeta) \models_{\texttt{LTL}} \psi \\
\sigma_R(\zeta) \models_{\texttt{LTL}} \psi_1 \wedge \psi_2 & \text{iff} & \sigma_R(\zeta) \models_{\texttt{LTL}} \psi_1 \text{ and } \sigma_R(\zeta) \models_{\texttt{LTL}} \psi_2 \\
\sigma_R(\zeta) \models_{\texttt{LTL}} \bigcirc\psi & \text{iff} & \sigma_R(\zeta)^1 \models_{\texttt{LTL}} \psi \\
\sigma_R(\zeta) \models_{\texttt{LTL}} \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \text{there exists a } j \geq 0 \text{ with } \sigma_R(\zeta)^j \models_{\texttt{LTL}} \psi_2 \text{ and} \\
 & & \text{for all } 0 \leq i < j \text{ the following holds: } \sigma_R(\zeta)^i \models_{\texttt{LTL}} \psi_1
\end{array}$$

for atomic propositions $a \in AP$ and LTL formulas ψ, ψ_1, and ψ_2. We say a Petri net or Petri net with transits $\mathcal{N}$ *satisfies* an LTL formula ψ iff $\mathcal{N} \models_{\texttt{LTL}} \psi$ holds. ◀

As an example for specifying objectives in LTL on runs of a Petri net, we consider fairness and maximality assumptions for runs.

▶ **Example 1.** Neither the definition of a run (Definition 6) nor the semantics of LTL (Definition 21) impose any constraints on the length of the run. However, for a reachability objective, where we want to reach a specific place, we are typically interested in

some sort of maximal runs. A Petri net only satisfies a formula, when all runs satisfy the formula, but for each Petri net there is the run which only consists of the initial marking. Clearly, we do not want to already reject a formula because the initial marking does not contain the place we want to reach. To avoid such cases we introduce two kinds of maximality assumptions.

On the one hand, we consider a run to be *interleaving-maximal* iff whenever there is a possible extension of the run, we must extend the run somewhere. This fits to the common semantics on the reachability graph for example used in the LTL model checking contest [KBGH+21].

Interleaving-Maximality. A run β is *interleaving-maximal* iff, whenever some transition is enabled, some transition will be taken:

$$\beta \models \Box(\bigvee_{t\in\mathcal{T}} \texttt{pre}(t) \rightarrow \bigvee_{t\in\mathcal{T}} \bigcirc t) \qquad \text{for } \texttt{pre}(t) = \bigwedge_{p\in pre(t)} p.$$

On the other hand, we consider a maximality assumption which focuses more on the concurrent structure of the run. This means that we do not want some process in the system to cut off the progressing of another *concurrent* process. Thus, for a run to be *concurrency-maximal* we require that whenever some process can progress the process has to progress.

Concurrency-Maximality. A run β is *concurrency-maximal* iff, when a transition t is from some moment on always enabled, a transition t' (including t itself) sharing a precondition with t is taken infinitely often:

$$\beta \models \bigwedge_{t\in\mathcal{T}} (\Diamond\Box\, \texttt{pre}(t) \rightarrow \Box\Diamond \bigvee_{p\in pre(t), t'\in post(p)} t').$$

This notion is strongly related to the notion of *progress* for transitions in [Rei98]. We can also mark some transition to be *fair* (in a weak or a strong sense), i.e., we do not want the run to completely cut off this transition in certain situations. As an example consider the Petri net with transits in Fig. 2.3 on page 21 modeling a software-defined networking problem. We do not want to reject a formula already because there is the run which only creates infinitely many data packets in switch `A`. Thus, we want the transitions forwarding the data packet to the next switch to eventually fire.

Fairness. A run β is *weakly fair* w.r.t. a transition t iff, whenever t is always enabled after some point, t is taken infinitely often:

$$\beta \models \Diamond\Box\, \texttt{pre}(t) \rightarrow \Box\Diamond t.$$

A run β is *strongly fair* w.r.t. a transition t iff, whenever t is enabled infinitely often, t is taken infinitely often:

$$\beta \models \Box\Diamond\, \texttt{pre}(t) \rightarrow \Box\Diamond t. \qquad \blacktriangleleft$$

In Flow-CTL* we typically select with such formulas specific runs of the system and check the data flow along these runs.

5.2 Flow-CTL*

We use Petri nets with transits to enable reasoning about two separate timelines. Properties defined on the run of the system concern the *global* timeline and allow us to reason about the global behavior of the system like its general control or fairness and maximality assumptions. Additionally, we can express requirements about the individual data flow like the access possibilities of people in buildings. These requirements concern the *local* timeline of the specific data flow. In Flow-CTL*, we can reason about these two parts with LTL in the *run* and with CTL* in the *flow* part of the formula.

In this chapter we also use the ingoing semantics for the flow part of the system.

▶ **Definition 22 (Ingoing Semantics – Flow Chain Suffixes).** Given a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$. To a (finite or infinite) *flow chain suffix* $\xi = t_0, p_0, t_1, p_1, t_2, \ldots$ of β we associate a *trace* $\sigma_F(\xi) : \mathbb{N} \to \{\{t, p\}, \{p\} \mid p \in \mathscr{P}^R \wedge t \in \mathscr{T}^R\}$ with

$$\sigma_F(\xi)(i) = \{t_i, p_i\} \text{ for all } i \in \mathbb{N}$$

if ξ is infinite and

$$\sigma_F(\xi)(i) = \begin{cases} \{t_i, p_i\} & \text{for } i \leq n \\ \{p_n\} & \text{otherwise} \end{cases}$$

if $\xi = t_0, p_0, t_1, p_1, \ldots, t_n, p_n$ is finite.

Additionally, we define the trace $\sigma_{\mathfrak{s}}(\{p\})(i) = \{p\}$ for all $i \in \mathbb{N}$ for a place $p \in \mathscr{P}^R$. This technicality is necessary to allow for the stuttering at the last place of a finite flow chain suffix in Definition 23, although every flow chain suffix must start with a transition. ◀

Hence, a trace of a flow chain suffix is an infinite sequence of states collecting the current place and ingoing transition of the flow chain suffix, which stutters on the last place p for finite flow chain suffixes.

We evaluate a CTL* formula $\Phi \in \texttt{CTL}^*$ on a Petri net with transits $\mathcal{N}_T$ over the traces of the flow chains of its runs.

▶ **Definition 23 (CTL* on Flow Chain Suffixes).** Given a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$, a CTL* formula ϕ, and a state $s \in \{\{t, p\}, \{p\} \mid p \in \mathscr{P}^R \wedge t \in \mathscr{T}^R\}$ of a trace $\sigma_F(\xi)$ of a flow chain suffix ξ. The *semantics* of CTL* on flow chain suffixes

is defined as follows:

$\beta, s \models_{\text{CTL}^*} a$	iff	$a \in \rho(s)$
$\beta, s \models_{\text{CTL}^*} \neg \Phi$	iff	not $\beta, s \models_{\text{CTL}^*} \Phi$
$\beta, s \models_{\text{CTL}^*} \Phi_1 \wedge \Phi_2$	iff	$\beta, s \models_{\text{CTL}^*} \Phi_1$ and $\beta, s \models_{\text{CTL}^*} \Phi_2$
$\beta, s \models_{\text{CTL}^*} \mathbf{E}\,\phi$	iff	Case $s \subseteq \mathscr{P}^R : \beta, \sigma_s(s) \models_{\text{CTL}^*} \phi$ holds. Otherwise: there *exists* some flow chain suffix $\xi = t_0, p_0, \ldots$ of β with $\{t_0, p_0\} = s$ such that $\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi$ holds.
$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \Phi$	iff	$\beta, \sigma_F(\xi)(0) \models_{\text{CTL}^*} \Phi$
$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \neg \phi$	iff	not $\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi$
$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi_1 \wedge \phi_2$	iff	$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi_1$ and $\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi_2$
$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \mathbf{X}\,\phi$	iff	$\beta, \sigma_F(\xi)^1 \models_{\text{CTL}^*} \phi$
$\beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi_1 \mathbf{U}\,\phi_2$	iff	there exists some $j \geq 0$ with $\beta, \sigma_F(\xi)^j \models_{\text{CTL}^*} \phi_2$ and f. a. $0 \leq i < j$ the following holds: $\beta, \sigma_F(\xi)^i \models_{\text{CTL}^*} \phi_1$

for atomic propositions $a \in AP$, state formulas Φ, Φ_1, and Φ_2, and path formulas ϕ, ϕ_1, and ϕ_2. ◀

Note that since the formulas are evaluated on the runs of $\mathscr{N}_T$, the branching is in the transitions and not in the places of $\mathscr{N}_T$.

Based on Definition 21 used to define the global run part of the formula as LTL formulas and Definition 23 used to define the local flow part of the formula as CTL* formulas, we define the new specification language Flow-CTL*.

▶ **Definition 24 (Flow-CTL*).** We define the *syntax* of Flow-CTL* with:

$$\varphi ::= \psi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \psi \rightarrow \varphi \mid \mathbb{A}\,\phi$$

where φ, φ_1, φ_2 are Flow-CTL* formulas, ψ is an LTL formula, and ϕ is a CTL* formula. We call $\varphi_{\mathbb{A}} = \mathbb{A}\,\phi$ *flow formulas* and all other subformulas which are not under the $\mathbb{A}$ operator *run formulas*.

The *semantics* of a Petri net with transits $\mathscr{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ satisfying a Flow-CTL* formula φ is defined over the covering firing sequences of its runs:

$\mathscr{N}_T \models \varphi$	iff	for all runs β of $\mathscr{N}_T : \beta \models \varphi$
$\beta \models \varphi$	iff	for all firing sequences ζ covering β : $\beta, \sigma_R(\zeta) \models \varphi$
$\beta, \sigma_R(\zeta) \models \psi$	iff	$\sigma_R(\zeta) \models_{\text{LTL}} \psi$
$\beta, \sigma_R(\zeta) \models \varphi_1 \wedge \varphi_2$	iff	$\beta, \sigma_R(\zeta) \models \varphi_1$ and $\beta, \sigma_R(\zeta) \models \varphi_2$
$\beta, \sigma_R(\zeta) \models \varphi_1 \vee \varphi_2$	iff	$\beta, \sigma_R(\zeta) \models \varphi_1$ or $\beta, \sigma_R(\zeta) \models \varphi_2$
$\beta, \sigma_R(\zeta) \models \psi \rightarrow \varphi$	iff	$\beta, \sigma_R(\zeta) \models \psi$ implies $\beta, \sigma_R(\zeta) \models \varphi$
$\beta, \sigma_R(\zeta) \models \mathbb{A}\,\phi$	iff	for all flow chains ξ of $\beta : \beta, \sigma_F(\xi) \models_{\text{CTL}^*} \phi$

with LTL formulas ψ, CTL* formulas ϕ, and Flow-CTL* formulas φ, φ_1 and φ_2. We say a Petri net with transits $\mathcal{N}_T$ *satisfies* a Flow-CTL* formula φ iff $\mathcal{N}_T \models \varphi$ holds. ◀

Due to the covering of the firing sequences and the maximality constraint of the flow chain suffixes, every behavior of the run and its complete data flow is incorporated. The operator $\mathbb{A}$ chooses flow chains rather than flow trees as our definition is based on the common semantics of CTL* over paths. But we can give a direct correspondence because a tree satisfies a formula iff all paths of the tree satisfy the formula.

The *trace tree* $\sigma_T(\tau)$ of a flow tree $\tau = (T, v)$ is a $\{\{t, p\}, \{p\} \mid p \in \mathscr{P}^R \wedge t \in \mathscr{T}^R\}$-labeled infinite tree $\sigma_T(\tau) = (T_\sigma, v_\sigma)$ such that i) for all $n \in T$ also $n \in T_\sigma$ and $v_\sigma(n) = \{t, p\}$ for $v(n) = (t, p)$ holds and ii) for all leaves $n' \in T$ we have nodes $n_i = n' \cdot 0^{i+1} \in T_\sigma$ for all $i \in \mathbb{N}$ with labels $v_\sigma(n_i) = \{p\}$ for $v(n') = (t, p)$. This means a trace tree has the structure of the flow tree, but stutters on finite branches. Thus, for a flow tree (T, v) the trace tree $\sigma_T(T, v)$ satisfies a Flow-CTL* formula ϕ iff all paths of the tree satisfy ϕ. The paths are exactly all traces $\sigma_F(\xi)$ of flow chains ξ starting in the root of the tree.

Note that it suffices to find for each flow formula one of the possibly infinitely many flow trees of a run to invalidate the subformula. Even though finding one tree suffices, checking the data flow while the control changes the system complicates the direct expression of the model checking problem within a finite model.

5.3 Reduction to Model Checking Petri Nets against LTL

We solve the model checking problem for a given Flow-CTL* formula φ and a safe Petri net with transits $\mathcal{N}_T$ in four steps:

1. For each flow subformula $\mathbb{A}\,\phi_i$ of φ, a subnet $\mathcal{N}_i^>$ is created via a sequence of automata constructions (substeps (i)-(iv) in Sec. 5.3.1) that allows us to guess a counterexample, i.e., a flow tree not satisfying ϕ_i, and to check its correctness.

2. A Petri net $\mathcal{N}^>$ is created by composing the subnets $\mathcal{N}_i^>$ with a copy of $\mathcal{N}_T$ such that every firing of a transition subsequently triggers each subnet.

3. A formula $\psi^>$ is created such that the subnets $\mathcal{N}_i^>$ are adequately skipped for the run part of φ, and the flow parts are replaced by LTL formulas corresponding to checking the acceptance of a run of the respective automaton.

4. $\mathcal{N}^> \models_{\mathtt{LTL}} \psi^>$ is checked to answer $\mathcal{N}_T \models \varphi$.

The construction from a given safe Petri net with transits $\mathcal{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and a Flow-CTL* formula φ with $n \in \mathbb{N}$ flow subformulas $\varphi_{\mathbb{A}i} = \mathbb{A}\,\phi_i$ with atomic propositions AP_i to a Petri net $\mathcal{N}^> = (\mathscr{P}^>, \mathscr{T}^>, \mathscr{F}^>, \mathscr{F}_I^>, In^>)$ with inhibitor arcs and an LTL formula $\psi^>$ is defined in the following sections. Figure 5.2 gives a schematic overview of the procedure. Step 1. consists of the substeps (i)-(v), and Step 2. is depicted in the gray box at the bottom of the figure.

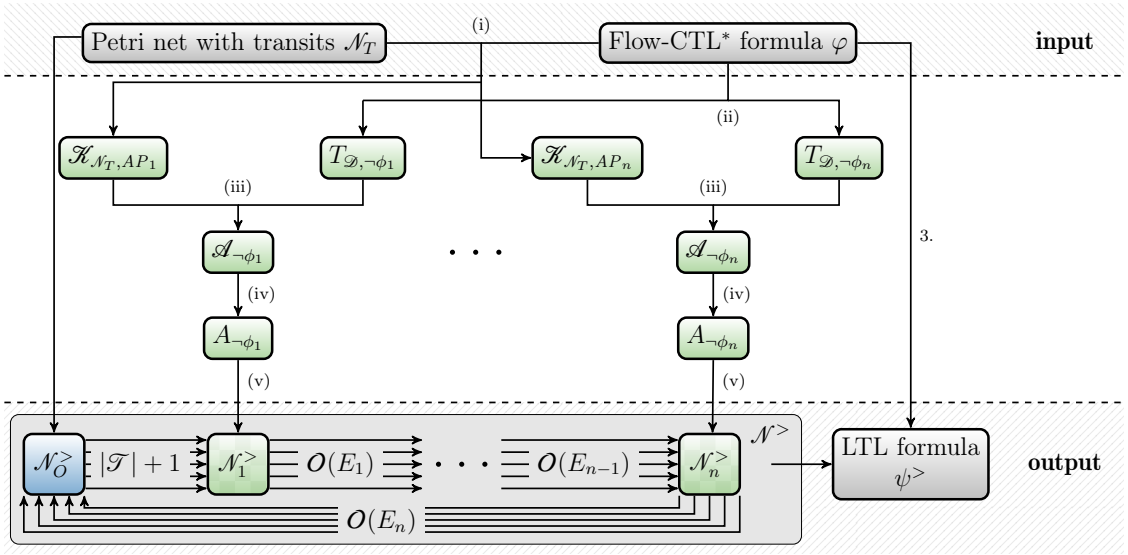


Fig. 5.2: *Overview of the model checking procedure: For a given safe Petri net with transits $\mathcal{N}_T$ and a Flow-CTL* formula φ, a standard Petri net $\mathcal{N}^>$ and an LTL formula $\psi^>$ are created: For each flow subformula $\mathbb{A}\,\phi_i$, create* (i) *a labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP_i}$ and* (ii) *the alternating tree automaton $T_{\mathscr{D},\neg\phi_i}$, construct* (iii) *the alternating word automaton $\mathscr{A}_{\neg\phi_i} = \mathscr{K}_{\mathcal{N}_T,AP_i} \times T_{\mathscr{D},\neg\phi_i}$, and from that* (iv) *the Büchi automaton $A_{\neg\phi_i}$ with edges E_i, which then* (v) *is transformed into a Petri net $\mathcal{N}_i^>$. These subnets are composed to a Petri net $\mathcal{N}^>$ such that they get subsequently triggered for every transition fired by the original net. The constructed formula $\psi^>$ skips for the run part of φ these subsequent steps and checks the acceptance of the guessed flow tree for each automaton in the corresponding subnet. The initial model checking problem is then solved by checking $\mathcal{N}^> \models_{\text{LTL}} \psi^>$.*

5.3.1 Automaton Construction for Flow Formulas

In Step 1 of the reduction method, we create for each flow subformula $\mathbb{A}\,\phi_i$ of the Flow-CTL* formula φ with atomic propositions AP_i a nondeterministic Büchi automaton $A_{\neg\phi_i}$ which loosely speaking accepts a sequence of transitions of a given run if *any* of the corresponding flow trees satisfies $\neg\phi_i$. This construction is done in four steps:

(i) Create the labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP_i}$ which, triggered by transitions $t \in \mathcal{T}$, tracks a flow chain of $\mathcal{N}_T$. Each path in the labeled Kripke structure corresponds to a flow chain of a run of $\mathcal{N}_T$.

(ii) Create the alternating tree automaton $T_{\mathscr{D},\neg\phi_i}$ for the negation of the CTL* formula ϕ_i and the set of degrees $\mathscr{D} = \{|post^{\Upsilon}(p,t)| \mid p \in \mathcal{P} \wedge t \in post^{\mathcal{N}_T}(p)\}$ which accepts all 2^{AP_i}-labeled $\mathscr{D}$-trees satisfying $\neg\phi_i$ [KVW00].

(iii) Create the alternating word automaton $\mathscr{A}_{\neg\phi_i} = \mathscr{K}_{\mathcal{N}_T,AP_i} \times T_{\mathscr{D},\neg\phi_i}$ similarly to [KVW00].

(iv) Alternation elimination for $\mathscr{A}_{\neg\phi_i}$ yields the nondeterministic Büchi automaton $A_{\neg\phi_i}$

with $\mathscr{L}(\mathscr{A}_{\neg\phi_i}) = \mathscr{L}(A_{\neg\phi_i})$ [MH84; DK08].

Step (ii) and Step (iv) are well-established constructions. For Step (iii), we modify the construction of [KVW00] by, roughly speaking, applying the idea not on all outgoing edges of a Kripke structure's state, but on the groups of equally labeled outgoing edges. By this, we obtain an alternating word automaton with the alphabet $A = \mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}$ of the labeled Kripke structure rather than an alternating word automaton over a 1-letter alphabet. The letters using the symbol $\mathfrak{s}$ are used for the stuttering of finite firing sequences or finite chains. This allows us to check whether the, by the input transitions *dynamically* created, system satisfies the CTL* subformula $\neg\phi_i$.

Constructing a Labeled Kripke Structure

Step (i) of the subnet construction creates the labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP_i} = (AP, A, S, S_0, \ell, R)$ of a Petri net with transits $\mathscr{N}_T$ and a set of atomic propositions AP_i for each flow subformula. These n labeled Kripke structures only differ in the labeling of the states, i.e., the labels are restricted to the atomic propositions AP_i and in introducing additional states dependent on whether the fired transition is contained in the atomic propositions. The latter is only introduced due to optimization reasons. The Kripke structure serves (in combination with the tree automaton) for checking the satisfaction of a flow tree of a given run. Hence, the states track the current place of the considered chain of the tree and additionally, when the transition extending the chain into the place occurs in the formula, also this ingoing transition. The initial states S_0 are either the tuples of transitions t_j and places p_j which start a flow chain, i.e., all $(t_j, p_j) \in \mathscr{T} \times \mathscr{P}$ with $(\triangleright, p_j) \in \Upsilon(t_j)$ when $t_j \in AP$ or only the place p_j otherwise. The labeling function ℓ labels the states with its components as long as they occur in AP. The transition relation R connects the states with respect to the transits and allows for switching from every state corresponding to a place $p \in \mathscr{P}$ into a respective stuttering state $p_\mathfrak{s}$ to permit the stuttering for finite firing sequences and finite flow chains. Note that the construction of the Petri net (Step 2) nondeterministically selects the violating tree of the flow subformula and for this tree, the Kripke structure is used to check the satisfaction.

We formally define the labeled Kripke structure of which the unwinding triggered by an input sequence corresponding to a covering firing sequence of a run corresponds to one flow tree of a run of $\mathscr{N}_T$.

▶ **Definition 25 (Labeled Kripke Structure).** For a given a Petri net with transits $\mathscr{N}_T$ and a set of atomic propositions AP_i we construct the *labeled Kripke structure* $\mathscr{K}_{\mathscr{N}_T,AP_i} = (AP, A, S, S_0, \ell, R)$ with

- the finite set of *atomic propositions* $AP = AP_i \subseteq \mathscr{P} \cup \mathscr{T}$,
- the *alphabet* $A = \mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}$,
- the finite set of *states* $S \subseteq ((\mathscr{T} \cap AP) \times \mathscr{P}) \cup \mathscr{P} \cup \{p_\mathfrak{s} \mid p \in \mathscr{P}\}$,
- the finite set of *initial states* $S_0 \subseteq S$,

- the *labeling function* $\ell : S \to 2^{AP}$,
- the *labeled transition relation* $R \subseteq S \times A \times S$.

The *initial states* $S_0 = \{(t,p) \in \mathscr{T} \times \mathscr{P} \mid t \in AP : (\triangleright, p) \in \Upsilon(t)\} \cup \{p \in \mathscr{P} \mid \exists t \in \mathscr{T} \setminus AP : (\triangleright, p) \in \Upsilon(t)\}$ correspond to all tuples of transitions $t \in AP$ and places (or only to the places) which start a data flow in $\mathscr{N}_T$.
The *labeling function* ℓ labels the states with its components which occur in AP: $\forall (t,p) \in (\mathscr{T} \cap AP) \times \mathscr{P} : \ell((t,p)) = \{t,p\} \cap AP$, $\forall p \in \mathscr{P} : \ell(p) = \{p\} \cap AP$, and $\forall p \in \mathscr{P} : \ell(p_\mathfrak{s}) = \{p\} \cap AP$. The *transition relation* is composed out of three sets $R = R' \cup R'' \cup R'''$. The relation R' connects the states with respect to the transits:

$$\begin{aligned} R' =& \{(p,t,q) \in S \times \mathscr{T} \times S \mid (p,q) \in \Upsilon(t) \wedge t \notin AP\} \\ & \cup \{(p,t,(t,q)) \in S \times \mathscr{T} \times S \mid (p,q) \in \Upsilon(t) \wedge t \in AP\} \\ & \cup \{((t',p),t,q) \in S \times \mathscr{T} \times S \mid (p,q) \in \Upsilon(t) \wedge t \notin AP\} \\ & \cup \{((t',p),t,(t,q)) \in S \times \mathscr{T} \times S \mid (p,q) \in \Upsilon(t) \wedge t \in AP\}. \end{aligned}$$

The relation R'' adds $\mathfrak{s}$-labeled loops and loops for each transition $t \in \mathscr{T}$ to states $p_\mathfrak{s}$ with $p \in \mathscr{P}$ to stutter in these states for finite firing sequences and finite flow chains: $R'' = \{(p_\mathfrak{s}, x, p_\mathfrak{s}) \in S \times (\{\mathfrak{s}\} \cup \mathscr{T}) \times S \mid p \in \mathscr{P}\}$. For switching into the stuttering mode, the relation R''' adds $\mathfrak{s}_p$-labeled edges between each state (t,p) or p and $p_\mathfrak{s}$ to guess the beginning of the local stuttering: $R''' = \{(x, \mathfrak{s}_p, p_\mathfrak{s}) \in S \times \{\mathfrak{s}_q \mid q \in \mathscr{P}\} \times S \mid x = (t,p) \in \mathscr{T} \times \mathscr{P} \vee x = p \in \mathscr{P}\}$. The *states* S are exactly the states reachable from the initial states.

We define the function $\mathfrak{s}_\mathscr{K} : S \times A \to 2^S$ with $\mathfrak{s}_\mathscr{K}(s,l) = \{s' \in S \mid (s,l,s') \in R\}$ returning all l-labeled successors of a state s. ◀

▶ **Example 2.** As an example we build the labeled Kripke structure corresponding to the example representing the lecture hall depicted as Petri net with transits $\mathscr{N}_T$ in Fig. 2.4 on page 22 and to the atomic propositions $AP = \{e, y\}$. We use the first letter of each node as abbreviation for the node. The set AP contains only the atomic propositions corresponding to the *emergency* transition and the place *yard*, because only these will be used in the formula to be checked. This is only an optimization to obtain a smaller automaton. But if we want to be independent of a particular formula, we can simply use the set of all nodes. Applying Definition 25 yields the labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP} = (\{e,y\}, \mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}, \{c,h,y,c_\mathfrak{s},h_\mathfrak{s},y_\mathfrak{s},(e,c),(e,h),(e,y)\}, \{c\}, \ell, R)$ with ℓ and R depicted in Fig. 5.3. Note the nondeterminism in state h and (e,h) for the edges labeled with $l_{h \to [c,y]}$ corresponding to the branching of the data flow in transition $l_{h \to [c,y]}$. ◀

The size of the labeled Kripke structure is dependent on the number of transits used in the specification for the local data flow:

▶ **Lemma 1 (Size of the Kripke Structure).** *The Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP_i}$ constructed in Definition 25 has $\boldsymbol{O}(|AP_i \cap \mathscr{T}| \cdot |\mathscr{N}_T| + |\mathscr{N}_T|)$ states.* ◀

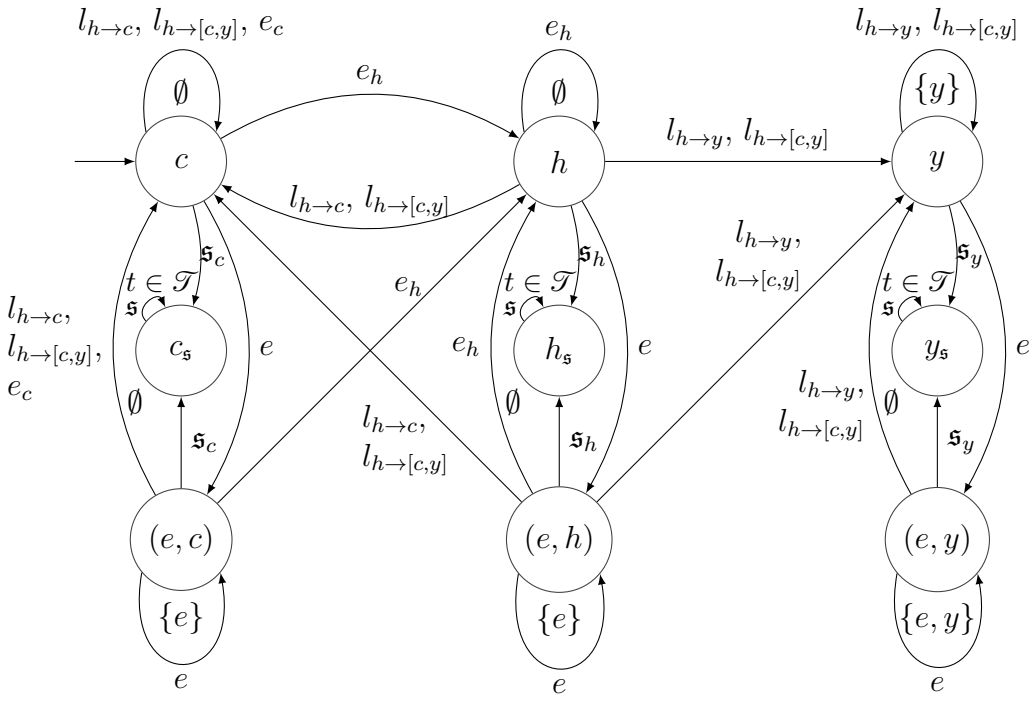


Fig. 5.3: *The labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP}$ corresponding to the Petri net with transits $\mathcal{N}_T$ depicted in Fig. 2.4 and to the atomic propositions $AP = \{e, y\}$. Each state of a run corresponds to the current state of the tracked chain, i.e., either only to the place or, in case the transition extending the chain into the place occurs in the formula, to the place and this ingoing transition.*

Proof. The number of states of $\mathscr{K}_{\mathcal{N}_T,AP_i}$ follows directly from Definition 25. □

We create two differently labeled trees for a given covering firing sequence of a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$ and a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ starting a flow chain. The first one is a S-labeled tree corresponding to the unwinding of the respective labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP_i}$ along the firing sequence. The second tree has the same structure but is labeled with the labels of the states of the Kripke structure. Thus, it is a 2^{AP}-labeled tree. This tree corresponds to the flow tree of β starting with transition t in place p.

▶ **Definition 26 (Trees of Unwinding).** Given a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$, a covering firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ with a transition $t_i \in \mathscr{T}^R$ and a place $p \in M_{i+1} \subseteq \mathscr{P}^R$ such that $(\triangleright, p) \in \Upsilon^R(t_i)$, a set of atomic propositions AP, and the corresponding labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP} = (AP, A, S, S_0, \ell, R)$. We define a S-labeled $\mathscr{D}$-tree $(T'_\mathscr{K}, v'_\mathscr{K})$ by unwinding $\mathscr{K}_{\mathcal{N}_T,AP}$ along ζ:

(IA) The root ϵ is labeled with the initial state $s_0 = (\rho(t_i), \rho(p))$ if $\rho(t_i) \in AP$ and $s_0 = \rho(p)$ of the labeled Kripke structure: $v'_\mathscr{K}(\epsilon) = s_0$.

(IS) Let $t_{j-1} \in \mathscr{T}^R$ be the last transition of ζ so far handled by this inductive definition (or $j = 0$ for the first step) and $n_0, \ldots, n_k \in T'_\mathscr{K}$ the leafs of the current tree.

For each n_l for $l \in \{0, \ldots, k\}$ with $v'_{\mathscr{K}}(n_l) = s_l$ (either $s_l = (t, p')$, $s_l = p'$, or $s_l = p'_{\mathfrak{s}}$ for some $t \in \mathscr{T}$ and $p' \in \mathscr{P}$) we consider all successors $s' \in S$ (either $s' = (t', q)$, $s' = q$, or $s' = q_{\mathfrak{s}}$ for some $t' \in \mathscr{T}$ and $q \in \mathscr{P}$) with $(s_l, \rho(t_j), s') \in R$. If $s' \neq q_{\mathfrak{s}}$ and there is some successor, we know we can order all successors according to $post^{\Upsilon}(p', \rho(t_j))$ due to the definition of R. Thus, for all $0 \leq m < |post^{\Upsilon}(p', \rho(t_j))|$ we add the node $n_l \cdot m$ to $T'_{\mathscr{K}}$ with $v'_{\mathscr{K}}(n_l \cdot m) = s'$ with $q = \langle post^{\Upsilon}(p', \rho(t_j))\rangle_m$. If there is no transition $t_x \in \{t_{j+1}, t_{j+2}, \ldots\}$ with $(q, \cdot) \in \Upsilon(\rho(t_x))$, i.e., a finite flow chain ends in a place corresponding to q, we add a grand child $n_l \cdot m \cdot 0$ to $T'_{\mathscr{K}}$ with $v'_{\mathscr{K}}(n_l \cdot m \cdot 0) = s''$ for $(s', \mathfrak{s}_q, s'') \in R$ (i.e., $s'' = q_{\mathfrak{s}}$). If $s' = q_{\mathfrak{s}}$, we add the node $n_l \cdot 0$ with $v'_{\mathscr{K}}(n_l \cdot 0) = s'$ for the unique successor $s' = q_{\mathfrak{s}}$.

(F) Finally, due to finite firing sequences $T'_{\mathscr{K}}$ can still contain leafs. Thus, if there are any leafs $v'_{\mathscr{K}}(n_l) = s_l$ in $T'_{\mathscr{K}}$ left, i.e., $s_l = p'_{\mathfrak{s}}$ for some $p' \in \mathscr{P}$, we add the successor node $n = n_l \cdot 0$ to $T'_{\mathscr{K}}$ and label it with $v'_{\mathscr{K}}(n_l \cdot 0) = s'$ for $(s_l, \mathfrak{s}, s') \in R$, i.e., $s' = p'_{\mathfrak{s}}$. This step is repeated infinitely often.

We define the 2^{AP}-labeled $\mathscr{D}$-tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ with $T_{\mathscr{K}} = T'_{\mathscr{K}}$ and $v_{\mathscr{K}}(n) = \ell(v'_{\mathscr{K}}(n))$ for all nodes $n \in T_{\mathscr{K}}$. ◀

It is easy to see that there is a one-to-one mapping of such a constructed tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ and a flow tree of a given run $\beta = (\mathscr{N}_T^R, \rho)$ of a Petri net with transits $\mathscr{N}_T$. Especially, the trace trees and these unwindings coincide. Lemma 9 on page 71 makes this correspondence formal and proves this result.

Constructing the Alternating Tree Automaton

Step (ii) of the construction creates an alternating tree automaton $T_{\mathscr{D},\neg\phi_i}$ for the negation of the CTL* formula ϕ_i and the set of possible degrees $\mathscr{D} = \{|post^{\Upsilon}(p,t)| \mid p \in \mathscr{P} \wedge t \in post^{\mathscr{N}_T}(p)\}$. This well-established construction is elaborately presented in [KVW00]. In this section we only recall the necessary parts to continue our running example.

First, hesitant alternating tree automata (HATA), a special kind of alternating tree automata, are introduced. HATA extend *Weak Alternating Automata (WAA)* [MSS88] by introducing more restrictions on the transition structure but allowing a more powerful acceptance condition. A WAA has a Büchi condition and suffices for the fragment of CTL formulas, whereas HATA use a combination of a Büchi and a co-Büchi condition.

▶ **Definition 27 (Hesitant Alternating Tree Automata [KVW00]).** A *Hesitant Alternating Tree Automaton (HATA)* is a five-tuple $T = (\Sigma, D, Q, \delta, q_0, F)$ with an *alphabet* Σ, a set of possible *degrees* $D \subset \mathbb{N}$, a set of *states* Q, a *transition relation* $\delta : Q \times \Sigma \times D \to \mathbb{B}^+(\{0, \ldots, \max(D)-1\} \times Q)$, an *initial state* q_0, and an *acceptance condition* $F = \langle B, C\rangle$ with $B, C \subseteq Q$ satisfying the following constraints:

(i) the states Q can be partitioned into disjoint sets Q_i,

(ii) there exists a partial order $\leq$ on Q_i,

(iii) Each set Q_i is classified as either *transient*, *existential*, or *universal* such that for all $q \in Q_i$, $\sigma \in \Sigma$, and $k \in D$ the following constraints hold:

(T) If Q_i is a *transient* set, then $\delta(q, \sigma, k)$ contains only elements of sets $Q_j \neq Q_i$ with $Q_j \leq Q_i$,

(E) If Q_i is an *existential* set, then in $\delta(q, \sigma, k)$ all elements of Q_i are disjunctively related and all other elements belong to a $Q_j \neq Q_i$ with $Q_j \leq Q_i$, and

(U) If Q_i is a *universal* set, then in $\delta(q, \sigma, k)$ all elements of Q_i are conjunctively related and all other elements belong to a $Q_j \neq Q_i$ with $Q_j \leq Q_i$.

The acceptance condition is a combination of a Büchi and a co-Büchi condition, i.e., the word w of the labels of an infinite path of a run tree is accepted iff $\texttt{Inf}(w)$ is existential and $\texttt{Inf}(w) \cap B \neq \emptyset$ or $\texttt{Inf}(w)$ is universal and $\texttt{Inf}(w) \cap C = \emptyset$. ◀

Note that each path of a run tree must eventually get trapped inside either an existential or a universal set Q_i. It cannot stay in transient sets because each successor must be in a smaller set and also for the existential and universal set, the successors are either in the smaller set or stay in the same one. The set of Q_i are finite, hence we cannot infinitely decrease such that $\texttt{Inf}(w)$ must either be existential or universal.

▶ **Theorem 1 (CTL* to HATA [KVW00]).** *Given a CTL* formula ϕ and a set of degrees $D \subset \mathbb{N}$, we can construct a hesitant alternating tree automaton $T_{D,\phi}$ of size $\mathcal{O}(|D| \cdot 2^{|\phi|})$ (of size $\mathcal{O}(|D| \cdot |\phi|)$ for a CTL formula ϕ) such that $\mathscr{L}(T_{D,\phi})$ is exactly the set of D-trees satisfying ϕ.* ◀

In the proof of the theorem in [KVW00] a sophisticated construction of the hesitant alternating tree automaton for a CTL* formula is given. Since we do not use any details of the construction in this thesis, we do not recall the definitions. But to give an intuition of the handling of branching properties and to continue our running example, we provide here the easier construction of a CTL formula also presented in [KVW00]. The *closure* $cl(\phi)$ of a CTL formula ϕ collects all CTL state subformulas of ϕ including ϕ itself. As optimization we can avoid collecting the subformulas for the abbreviations `true` and `false` in the closure and treat them separately in the construction by mapping them directly to `true` and `false` respectively.

▶ **Definition 28 (CTL to HATA [KVW00]).** Given a CTL formula ϕ with atomic propositions AP and a set of degrees $D \subset \mathbb{N}$, we construct in linear running time a HATA $T_{D,\phi} = (2^{AP}, D, cl(\phi), \delta, \phi, F)$ where for all $\sigma \in 2^{AP}$ and $k \in D$ the transition relation δ is defined as follows:

- for $p \in AP$:
  - $\delta(p, \sigma, k) = \texttt{true}$ if $p \in \sigma$,
  - $\delta(p, \sigma, k) = \texttt{false}$ if $p \notin \sigma$,
  - $\delta(\neg p, \sigma, k) = \texttt{true}$ if $p \notin \sigma$,
  - $\delta(\neg p, \sigma, k) = \texttt{false}$ if $p \in \sigma$,

- $\delta(\varphi_1 \wedge \varphi_2, \sigma, k) = \delta(\varphi_1, \sigma, k) \wedge \delta(\varphi_2, \sigma, k)$,
- $\delta(\varphi_1 \vee \varphi_2, \sigma, k) = \delta(\varphi_1, \sigma, k) \vee \delta(\varphi_2, \sigma, k)$,
- $\delta(\mathbf{AX}\,\varphi, \sigma, k) = \bigwedge_{c=0}^{k-1}(c, \varphi)$,
- $\delta(\mathbf{EX}\,\varphi, \sigma, k) = \bigvee_{c=0}^{k-1}(c, \varphi)$,
- $\delta(\mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2), \sigma, k) = \delta(\varphi_2, \sigma, k) \vee (\delta(\varphi_1, \sigma, k) \wedge \bigwedge_{c=0}^{k-1}(c, \mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2)))$,
- $\delta(\mathbf{E}(\varphi_1\,\mathbf{U}\,\varphi_2), \sigma, k) = \delta(\varphi_2, \sigma, k) \vee (\delta(\varphi_1, \sigma, k) \wedge \bigvee_{c=0}^{k-1}(c, \mathbf{E}(\varphi_1\,\mathbf{U}\,\varphi_2)))$,
- $\delta(\mathbf{A}(\varphi_1\,\mathbf{R}\,\varphi_2), \sigma, k) = \delta(\varphi_2, \sigma, k) \wedge (\delta(\varphi_1, \sigma, k) \vee \bigwedge_{c=0}^{k-1}(c, \mathbf{A}(\varphi_1\,\mathbf{R}\,\varphi_2)))$,
- $\delta(\mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2), \sigma, k) = \delta(\varphi_2, \sigma, k) \wedge (\delta(\varphi_1, \sigma, k) \vee \bigvee_{c=0}^{k-1}(c, \mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2)))$.

The acceptance set $F = (B, C)$ is defined such that B consists of all formulas $\varphi \in cl(\phi)$ of the form $\varphi = \mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2)$ and C consists of all formulas of the form $\varphi = \mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2)$. The partition is given by singleton sets $Q_i = \{\varphi_i\}$ for each $\varphi_i \in cl(\phi)$ and the partial order $\leq$ by $\{\varphi\} \leq \{\varphi'\}$ iff $\varphi \in cl(\varphi')$. ◀

This construction transforms the **A** operator of the formula into conjunctions in the automaton relation and the **E** operator of the formula into disjunctions to enforce that all path must be accepted or there exists an accepting path, respectively. For the **U** and **R** operators the construction exploits the recursive equivalences part of the *expansion laws* [BK08]: $\mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \mathbf{AX}\,\mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2))$ and $\mathbf{E}(\varphi_1\,\mathbf{U}\,\varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \mathbf{EX}\,\mathbf{E}(\varphi_1\,\mathbf{U}\,\varphi_2))$.

Note that the CTL formula has to be in *positive normal form*, i.e., negations are only applied to the atomic propositions, and only the operators provided in Definition 28 can be used. With the help of standard equivalence transformations, any formula can be converted into such a form. Furthermore, the constructed automaton can be seen as a WAA with the simple Büchi condition F consisting of all formulas $\varphi \in cl(\phi)$ of the form $\varphi = \mathbf{A}(\varphi_1\,\mathbf{R}\,\varphi_2)$ or $\varphi = \mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2)$ [KVW00].

▶ **Example 3.** As an example we use the subformula presented in Sec. 2.2 stating that on all paths to every point in time it holds that when there is an *emergency* then there must be a path such that the people finally reach the *yard*:

$$\phi = \mathbf{AG}(\mathit{emergency} \rightarrow \mathbf{EF}\mathit{yard}).$$

First, using the abbreviations $e = \mathit{emergency}$ and $y = \mathit{yard}$, we transform $\neg\phi$ into an equivalent formula in positive normal form only using the operators used in Definition 28: $\neg\mathbf{AG}(e \rightarrow \mathbf{EF}y) \equiv \mathbf{EF}(e \wedge \neg\mathbf{EF}y) \equiv \mathbf{E}(\mathtt{true}\,\mathbf{U}\,e \wedge \neg\mathbf{E}(\mathtt{true}\,\mathbf{U}\,y))$; moving the negation to the proposition yields the final formula

$$\neg\phi \equiv \underbrace{\mathbf{E}(\mathtt{true}\,\mathbf{U}\,e \wedge \mathbf{E}(\mathtt{false}\,\mathbf{R}\,\neg y))}_{=:F_0}.$$

Tab. 5.1: *The transition relation δ of the HATA $T_{\mathscr{D},\neg\phi}$ of Example 3.*

q	$\delta(q,\emptyset,k)$	$\delta(q,\{e\},k)$	$\delta(q,\{y\},k)$	$\delta(q,\{e,y\},k)$
F_2	`false`	`true`	`false`	`true`
F_5	`false`	`false`	`true`	`true`
F_4	`true`	`true`	`false`	`false`
F_3	$\bigvee_{c=0}^{k-1}(c,F_3)$	$\bigvee_{c=0}^{k-1}(c,F_3)$	`false`	`false`
F_1	`false`	$\bigvee_{c=0}^{k-1}(c,F_3)$	`false`	`false`
F_0	$\bigvee_{c=0}^{k-1}(c,F_0)$	$\bigvee_{c=0}^{k-1}(c,F_3) \vee \bigvee_{c=0}^{k-1}(c,F_0)$	$\bigvee_{c=0}^{k-1}(c,F_0)$	$\bigvee_{c=0}^{k-1}(c,F_0)$

Thus, we have the atomic propositions $AP = \{e, y\}$ and the closure

$$cl(\neg\phi) = \{F_0, \underbrace{e \wedge \mathbf{E}(\mathtt{false}\,\mathbf{R}\,\neg y)}_{=:F_1}, \underbrace{e}_{=:F_2}, \underbrace{\mathbf{E}(\mathtt{false}\,\mathbf{R}\,\neg y)}_{=:F_3}, \underbrace{\neg y}_{=:F_4}, \underbrace{y}_{=:F_5}\}.$$

Given the set of degrees $\mathscr{D} = \{1, 2\}$, Definition 28 creates the alternating tree automaton $T_{\mathscr{D},\neg\phi} = (2^{AP}, \mathscr{D}, cl(\neg\phi), \delta, \neg\phi, \{\mathbf{E}(\mathtt{false}\,\mathbf{R}\,\neg y)\})$ with the transition relation δ depicted in Tab. 5.1. ◀

Constructing the Alternating Word Automaton

In Step (iii) of the automaton construction for a CTL* subformula ϕ_i of the Flow-CTL* formula, we combine the two previous structures into a product automaton. For an alternating tree automaton $T_{\mathscr{D},\neg\phi_i}$ created of $\neg\phi_i$ and a labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP_i}$, we define an alternating word automaton $\mathscr{A}_{\neg\phi_i}$. The idea is that this automaton accepts, when the flow trees belonging to a sequence of transitions of a run of $\mathscr{N}_T$, with adequately switching into the stuttering mode, satisfy $\neg\phi_i$. The definition is very similar to [KVW00], we only apply the edge definition on each equally labeled group of transitions separately and add skipping transitions for input letters which not involve the current branch of the flow tree.

▶ **Definition 29 (Product Automaton).** Given an alternating tree automaton $T_{\mathscr{D},\phi} = (2^{AP}, \mathscr{D}, Q_\phi, \delta_\phi, q_0, F_\phi)$ accepting exactly all $\mathscr{D}$-trees satisfying ϕ corresponding to Theorem 1 and a labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP} = (AP, A, S, S_0, \ell, R)$ created by Definition 25 with degrees in $\mathscr{D}$. For the *product automaton* $\mathscr{A}_\phi = (A, S \times Q_\phi, \delta, S_0 \times \{q_0\}, F)$ of $T_{\mathscr{D},\phi}$ and $\mathscr{K}_{\mathscr{N}_T,AP}$ the transition relation δ and the acceptance set F is defined as follows:

- For every $(s,q) \in S \times Q_\phi$, $l \in A$, $\mathtt{s}_\mathscr{K}(s,l) \neq \emptyset$, and $\delta_\phi(q, \ell(s), |\mathtt{s}_\mathscr{K}(s,l)|) = \theta$, we have an edge $\delta((s,q),l) = \theta'$, where the positive Boolean formula θ' is obtained from θ by replacing the atoms (c,q') with $(\langle \mathtt{s}_\mathscr{K}(s,l)\rangle_c, q')$, where $\langle \mathtt{s}_\mathscr{K}(s,l)\rangle_c$ is the c-th value of the ordered list of successors with respect to the transit's post place. If $\mathtt{s}_\mathscr{K}(s,l) = \emptyset$, we have a self-loop $\delta((s,q),l) = (s,q)$.

- The acceptance condition of F_ϕ is transferred to F by preserving the type and building the cross product of the acceptance set(s) with S.

This represents an alternating word automaton, which also allows for the partitioning of the states and the partial order to be taken over from the alternating tree automaton. ◀

Note that due to the additional self-loops, where especially $T_{\mathscr{D},\phi}$ is not advanced, each state of $\mathscr{A}_\phi$ has a successor for each label (or the formula results in `false` already due to the alternating tree automaton). Thus, for concurrent transitions of the firing sequence not involving the current chain under consideration, and for the local stutter transitions there is a successor which not triggers the automaton checking the formula. Later we ensure that runs using these loops infinitely often consecutively are omitted in the model checking procedure.

▶ **Example 4.** Let $AP = \{e, y\}$ be the set of atomic propositions, $A = \mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}$ be the set of labels, and $S = \{c, h, y, c_\mathfrak{s}, h_\mathfrak{s}, y_\mathfrak{s}, (e,c), (e,h), (e,y)\}$ the set of states of the labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP} = (AP, A, S, \{c\}, \ell, R)$ of Example 2. Furthermore, let $T_{\mathscr{D},\neg\phi} = (2^{AP}, \mathscr{D}, \{F_0, \ldots, F_5\}, \delta, F_0, \{F_3\})$ be the alternating tree automaton of Example 3. The alternating word automaton $\mathscr{A}_{\neg\phi} = (A, \mathcal{S}, \delta', \{(c, F_0)\}, \{c, h, y, (e,c), (e,h), c_\mathfrak{s}, h_\mathfrak{s}\} \times \{F_3\})$ with the set of states $\mathcal{S} = S \times \{F_0, F_3\} \setminus \{(y_\mathfrak{s}, F_3), ((e,y), F_3)\}$ constructed with Definition 29 is depicted in Fig. 5.4. We omitted all stutter states (i.e., $(c_\mathfrak{s}, F_0), (c_\mathfrak{s}, F_3), (h_\mathfrak{s}, F_0), (h_\mathfrak{s}, F_3), (y_\mathfrak{s}, F_0)$), the loops in these states for all $t \in \mathscr{T}$ and $\mathfrak{s}$, and the edges $\mathfrak{s}_p$ from each state containing a place $p \in \mathscr{P}$ to the stutter state $p_\mathfrak{s}$ for more clearance.

Note that there is no stutter state $(y_\mathfrak{s}, F_3)$, because due to R we could only reach this state from states $(y, \cdot), ((e,y), \cdot) \in \mathcal{S}$. But the transition relation depicted in Tab. 5.1 for the alternating tree automaton $T_{\mathscr{D},\phi}$ has no F_3 successor for the columns $\delta_\phi(q, \{y\}, k)$ and $\delta_\phi(q, \{e,y\}, k)$. We either switch into F_0 or have no successor at all. For example for state (y, F_3) Tab. 5.1 yields that $\delta_\phi(F_3, \{y\}, k) = \mathtt{false}$ for all $k \in \mathscr{D}$. Thus, there is no successor in $\mathscr{A}_{\neg\phi}$ for (y, F_3) for the transition $(y, \mathfrak{s}_y, y_\mathfrak{s}) \in R$ of the Kripke structure.

Furthermore, it is interesting to notice that the state (y, F_3) is a Büchi state and even though the transition relation of the alternating tree automaton has no successors for this situation, there exists self-loops at this state. These belong to transitions which cannot transit the data flow in y, i.e., there is no successor in the labeled Kripke structure for these labels, but the loops are added due to the $\mathfrak{s}_\mathscr{K}(s,l) = \emptyset$ case of Definition 29. Even though (y, F_3) is a Büchi state theses loops are not making any harm for the acceptance, because we later reject runs where we do not switch into a stutter state when no transition further transiting the data flow is ever fired. ◀

The size of the alternating word automaton $\mathscr{A}_\phi$ directly follows from the definition as a product automaton (see the proof on page 72).

▶ **Lemma 2 (Size of the Product Automaton).** *The constructed alternating word automaton $\mathscr{A}_\phi$ in Definition 29 has $\boldsymbol{O}(2^{|\phi|} \cdot |\mathscr{N}_T|^3)$ states for a CTL* formula ϕ and $\boldsymbol{O}(|\phi| \cdot |\mathscr{N}_T|^3)$ states for a CTL formula ϕ.* ◀

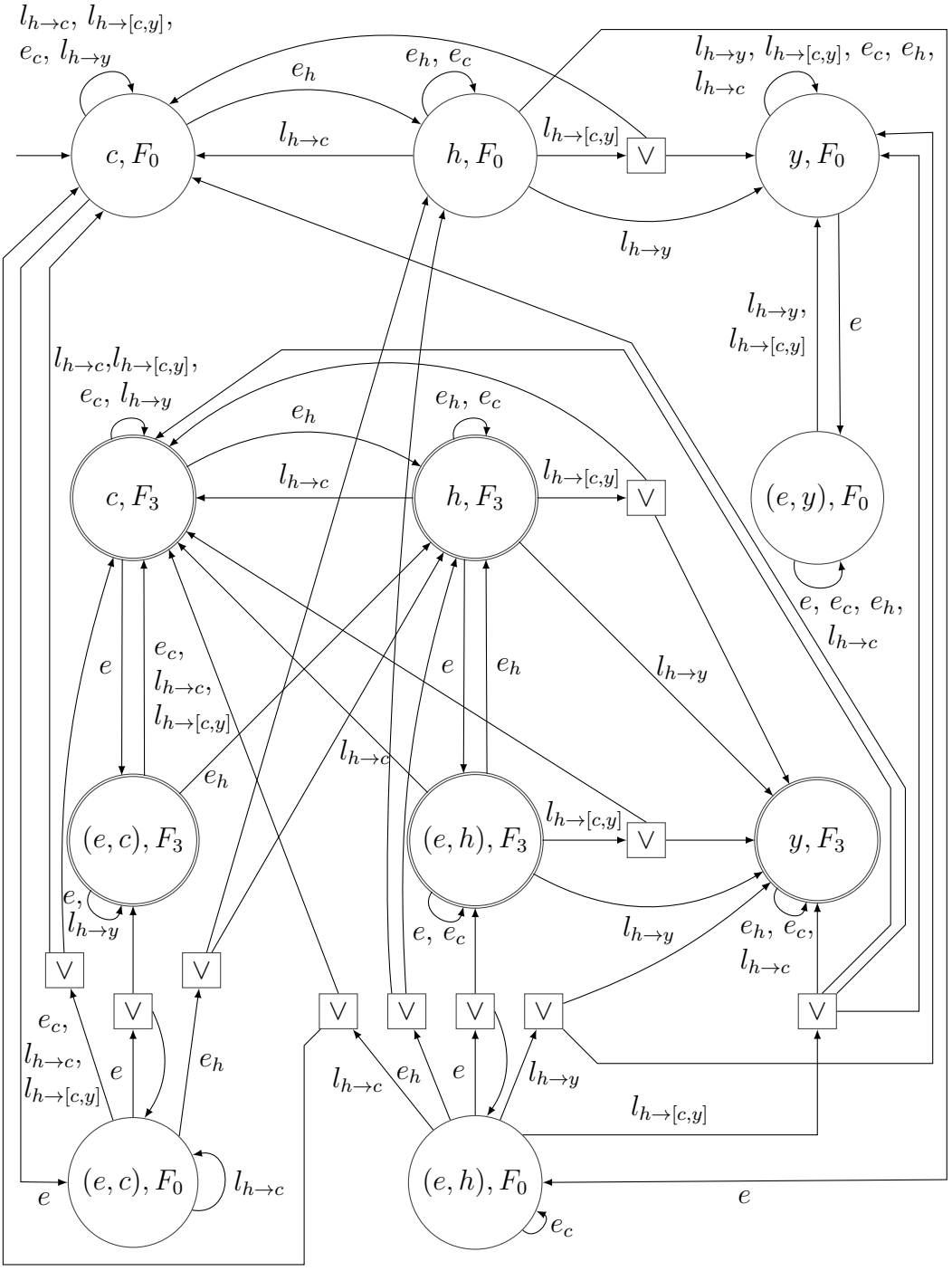


Fig. 5.4: *The alternating Büchi word automaton $\mathscr{A}_{\neg\phi} = \mathscr{K}_{\mathscr{N}_T,AP} \times T_{\mathscr{D},\neg\phi}$ corresponding to the product of the labeled Kripke structure constructed in Example 2 on page 54 and the alternating tree automaton $T_{\mathscr{D},\neg\phi}$ constructed in Example 3 on page 58. The additional stutter states $(c_\$, F_0), (c_\$, F_3), (h_\$, F_0), (h_\$, F_3), (y_\$, F_0)$ and the corresponding edges are omitted to increase the readability. In this example, the edges of the alternating automaton consists only of disjunctions.*

To prove that the product automaton behaves as expected, we restrict the set of words $w \in (\mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\})^\omega$ in which we are interested to those corresponding to a given firing sequence of a run and adhering correct local and global stuttering. Furthermore, we only consider those suffixes starting the flow tree under consideration. Thus, for a given run $\beta = (\mathscr{N}_T^R, \rho)$ of $\mathscr{N}_T$ and a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ with $(\triangleright, p) \in \Upsilon^R(t)$, we first define

$$\begin{aligned}\mathscr{L}_{t,p}(\beta) = \{&w \in (\mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_q \mid q \in \mathscr{P}\})^\omega \mid \exists \zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots \in Z(\beta) : \\ &\exists i \in \mathbb{N} : t = t_i \wedge p \in M_{i+1} \wedge \\ &(\zeta \text{ is infinite } \rightarrow w = \rho(t_i) w_{\mathfrak{s}}^i \rho(t_{i_1}) w_{\mathfrak{s}}^{i+1} \cdots) \wedge \\ &(\zeta = M_0[t_0\rangle \cdots [t_n\rangle M_{n+1} \text{ is finite } \rightarrow w = \rho(t_i) w_{\mathfrak{s}}^i \cdots \rho(t_n) w_{\mathfrak{s}}^n \mathfrak{s}^\omega) \wedge \\ &\forall \xi = t'_0, p'_0, \ldots, t'_m, p'_m \in \Xi(\beta) : t = t'_0 \wedge p = p'_0 \rightarrow \exists k \geq i : t'_m = t_k \wedge \\ &\exists k' \geq k : \mathfrak{s}_{\rho(p'_m)} \in w_{\mathfrak{s}}^{k'} \wedge \forall j \geq i : w_{\mathfrak{s}}^j \in \{\mathfrak{s}_q \mid q \in \mathscr{P}\}^*\}.\end{aligned}$$

The global stuttering is done for finite covering firing sequences of the run by adding $\mathfrak{s}^\omega$ to the end of the words. Switching into the local stuttering with transition $\mathfrak{s}_{\rho(p'_m)}$ for a finite flow chain ending in place p'_m is put at some point after the last transition of the chain has fired. The last conjunct of the predicate ensures that such local stuttering transitions only occur finitely often consecutively.

▶ **Lemma 3 (Correctness of the Product Automaton).** *Given a run $\beta = (\mathscr{N}_T^R, \rho)$ of a Petri net with transits $\mathscr{N}_T$, a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ such that $(\triangleright, p) \in \Upsilon^R(t)$, a CTL* formula ϕ with atomic propositions AP, the alternating tree automaton $T_{\mathscr{D},\phi}$ corresponding to Theorem 1, and the 2^{AP}-labeled $\mathscr{D}$-tree $(T_\mathscr{K}, v_\mathscr{K})$ created by Definition 26 from the labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP}$. For the alternating word automaton $\mathscr{A}_\phi$ in Definition 29*

$$\mathscr{L}(\mathscr{A}_\phi) \cap \mathscr{L}_{t,p}(\beta) \neq \emptyset \text{ iff } (T_\mathscr{K}, v_\mathscr{K}) \text{ is accepted by } T_{\mathscr{D},\phi}$$

holds. ◀

Proof Sketch. We can prove this lemma along the proof for the product automaton of a standard Kripke structure and an alternating tree automaton for CTL* specifications presented in [KVW00]. In doing so, we have to particularly take care of the skipping of concurrent transitions not transiting the current data flow of the branch and the local stuttering of finite data flow chains. The main idea is to transform the accepting runs for both directions. Thus, we transform a $\mathbb{N}^* \times Q_\phi$-labeled tree (T_r, v_r) for $T_{\mathscr{D},\phi}$ into a $S \times Q_\phi$-labeled tree (T'_r, v'_r) for $\mathscr{A}_\phi$ and vice versa. Without the stuttering and skipping of concurrent transitions we could remain the tree structure and only change the labeling of the run trees. Thus, we keep the branching of the alternation, but change the role of the states S of the Kripke structure. For the run (T'_r, v'_r) of the product automaton we have the states S as first component of the states of $\mathscr{A}_\phi$, i.e., they are already part of the label of the run (T'_r, v'_r). However, for the tree automaton $T_{\mathscr{D},\phi}$ these states are encoded in the $\mathbb{N}^*$ part of the labeling function of the run (T_r, v_r). This is because (T_r, v_r) is

a run on the 2^{AP}-labeled tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ constructed in Definition 26 from the labeled Kripke structure. Furthermore, this tree corresponds to the S-labeled tree $(T'_{\mathscr{K}}, v'_{\mathscr{K}})$. Thus, we can encode the additional branching for running over a tree rather than a word directly in the labeling. Thus, we label the created tree either according to the accepted 2^{AP}-labeled tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ ($(T'_{\mathscr{K}}, v'_{\mathscr{K}})$ respectively) or according to the accepted word $w \in \mathscr{L}(\mathscr{A}_\phi) \cap \mathscr{L}_{t,p}(\beta) \neq \emptyset$.

To handle the stuttering and the skipping of concurrent transitions, we pump up the tree (T_r, v_r), such that concurrent transitions and the starts of a local stuttering add new leaves to all branches of the tree. By that and by adding the starting edge for the local stuttering when creating the accepting word w, we obtain that each letter adds a new level to the tree. The $\mathtt{s}_{\mathscr{K}}(s, l) = \emptyset$ case of δ with the self-loops ensures the run property for such additional steps. Since the loops can only be taken finitely often consecutively and the acceptance condition is transferred to the cross product, the run is accepted. This construction is similar for the other direction but we project the tree (T'_r, v'_r) onto the steps involving the transiting of the current flow chain for each branch. The decisions are taken with respect to the first component (the states of the Kripke structure) of the labels of (T'_r, v'_r). For more details see the proof on page 72. □

Constructing the Nondeterministic Büchi Automaton

Given the alternating word automaton $\mathscr{A}_{\neg\phi_i}$. The well-established constructions (e.g., in [MH84; DK08]) allow for the alternation elimination yielding a nondeterministic Büchi automaton $A_{\neg\phi_i}$ accepting the same language. As we do not need the details of the constructions for the Rabin condition, we only recall the result here and provide only the construction for a Büchi condition due to Miyano and Hayashi to proceed with the running example.

▶ **Definition 30 (ABA to NBA [MH84]).** Given an alternating Büchi word automaton $\mathscr{A} = (\Sigma, Q, I, \delta, \mathtt{BUCHI}(F))$, we construct the nondeterministic Büchi automaton $A = (\Sigma, Q', I', \delta', \mathtt{BUCHI}(F'))$ with

- $Q' = 2^Q \times 2^Q$,
- $I' = \{(\{q_0\}, \emptyset) \mid q_0 \in I\}$,
- $\delta' = \{((Q_r, \emptyset), \sigma, (Q'_r, Q'_r \setminus F)) \mid Q'_r \models \bigwedge_{q \in Q_r} \delta(q, \sigma)\} \cup$
  $\{((Q_r, Q_{\bar{B}}), \sigma, (Q'_r, Q'_{\bar{B}} \setminus F)) \mid Q'_r \models \bigwedge_{q \in Q_r} \delta(q, \sigma) \wedge Q_{\bar{B}} \neq \emptyset \wedge Q'_{\bar{B}} \subseteq Q'_r$
  $\wedge\, Q'_{\bar{B}} \models \bigwedge_{q \in Q_{\bar{B}}} \delta(q, \sigma)\}$, and
- $F' = \{(Q_r, \emptyset) \mid Q_r \subseteq Q\}$,

such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(A)$. ◀

We can see the first component of the states (denoted by Q_r in the definition of δ') as the set of states the alternating automaton is currently in and the second component (denoted by $Q_{\bar{B}}$ in the definition of δ') as the subset of these states which still have to

visit a Büchi state. Thus, we track the current states of the alternating automaton in this component as long as they have not visited a Büchi state. When all path reached a Büchi state, i.e., the set is empty, we mark this state as the new Büchi state and newly start the tracking of the states which have not visited a Büchi state.

▶ **Theorem 2 (Alternation Elimination [DK08]).** *Given an alternating word automaton $\mathscr{A}$, we can construct a nondeterministic Büchi automaton A with $\mathscr{L}(\mathscr{A}) = \mathscr{L}(A)$. For a Büchi condition the NBA A is of size $\boldsymbol{O}(2^{2n})$ and for a Rabin condition the NBA A is of size $2^{\boldsymbol{O}(nk \log nk)}$ with n the size and k the index of $\mathscr{A}$.*

Note that for specifications in CTL the alternating word automaton $\mathscr{A}_{\neg\phi_i}$ constructed in Definition 29 has a Büchi condition due to the Büchi condition of the constructed WAA. For CTL* specifications the alternating word automaton $\mathscr{A}_{\neg\phi_i}$ has a combination of a Büchi and a co-Büchi acceptance condition because the constructed HATA for $\neg\phi_i$ has such a combination (B, C) as acceptance condition. This combination can be transformed into a single Rabin condition with only one pair $F_R = (I, F)$ [KB17]. Let Q^E be all states corresponding to existential sets and Q^U all states corresponding to universal sets. For the construction of the HATA for CTL* specifications we have $B \subseteq Q^E$ and $C \subseteq Q^U$. Thus, $B \cap C = \emptyset$ and we can choose the Rabin pairs $F = C$ and $I = B \cup (Q^U \setminus C)$. Remember the Rabin condition for one pair is given with $\texttt{RABIN}((I, F)) = \{w \in Q^\omega \mid \texttt{Inf}(w) \cap I \neq \emptyset \wedge \texttt{Inf}(w) \cap F = \emptyset\}$. Thus, w is accepted iff $\texttt{Inf}(w) \cap (B \cup (Q^U \setminus C)) \neq \emptyset$ and $\texttt{Inf}(w) \cap C = \emptyset$. When $\texttt{Inf}(w)$ is universal, i.e., $\texttt{Inf}(w) \subseteq Q^U$, the former is always true because $\texttt{Inf}(w) \neq \emptyset$ by definition. Hence, in this case the Rabin condition matches the co-Büchi condition of the HATA. When $\texttt{Inf}(w)$ is existential, i.e., $\texttt{Inf}(w) \subseteq Q^E$, the later is always true and the additional states of the union of the former formula cannot occur in $\texttt{Inf}(w)$ yielding that the Rabin condition also matches the Büchi condition of the HATA in this case.

The size of the Büchi automaton is dominated by the tree automaton construction and the removal of the alternation. Each construction adds one exponent for specifications in CTL*. Hence, the size of the nondeterministic Büchi automata is single-exponential for specifications given in CTL and double-exponential in specifications given in CTL*.

▶ **Lemma 4 (Size of the Büchi Automaton).** *The size of a Büchi automaton A_ϕ obtained through alternation elimination of the constructed alternating word automaton $\mathscr{A}_\phi$ in Definition 29 is double-exponential for specifications ϕ in CTL* and single-exponential for specifications in CTL in the size of the formula ϕ and the Petri net with transits $\mathcal{N}_T$.*◀

Proof. This results directly follow from Theorem 2 and Lemma 2. □

Putting it all together, we can show that the constructed Büchi automaton for a CTL* flow subformula ϕ_i can be used to check whether all corresponding flow chains satisfy ϕ_i.

▶ **Lemma 5 (Correctness of the Büchi Automaton).** *Given a run $\beta = (\mathcal{N}_T^R, \rho)$ of $\mathcal{N}_T$, a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ such that $(\triangleright, p) \in \Upsilon^R(t)$, and a CTL* formula ϕ.*

For the Büchi automaton $A_{\neg\phi}$ obtained through alternation elimination of the constructed alternating word automaton $\mathscr{A}_{\neg\phi}$ in Definition 29,

$$\mathscr{L}(A_{\neg\phi}) \cap \mathscr{L}_{t,p}(\beta) = \emptyset \text{ iff } \forall \xi = t, p, \ldots \text{ flow chain of } \beta : \beta, \sigma_F(\xi) \models_{\mathtt{CTL}^*} \phi$$

holds. ◀

Proof. Let $\beta = (\mathcal{N}_T^R, \rho)$ be a run of $\mathcal{N}_T$, $t \in \mathcal{T}^R$ a transition and $p \in \mathcal{P}^R$ a place such that $(\triangleright, p) \in \Upsilon^R(t)$, and ϕ a CTL*. Theorem 2 yields that $\mathscr{L}(A_{\neg\phi}) = \mathscr{L}(\mathscr{A}_{\neg\phi})$ for the alternating word automaton $\mathscr{A}_{\neg\phi}$ of Definition 29. Lemma 3 yields $\mathscr{L}(\mathscr{A}_{\neg\phi}) \cap \mathscr{L}_{t,p}(\beta) = \emptyset$ iff the 2^{AP}-labeled $\mathscr{D}$-tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ created by Definition 26 from the labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T, AP}$ is not accepted by $T_{\mathscr{D},\neg\phi}$. Theorem 1 states that $T_{\mathscr{D},\neg\phi}$ accepts exactly all $\mathscr{D}$-trees satisfying $\neg\phi$ and likewise accepts $T_{\mathscr{D},\phi}$ exactly all $\mathscr{D}$-trees satisfying ϕ. Thus, $\mathscr{L}(\mathscr{A}_{\neg\phi}) \cap \mathscr{L}_{t,p}(\beta) = \emptyset$ iff $(T_{\mathscr{K}}, v_{\mathscr{K}})$ is accepted by $T_{\mathscr{D},\phi}$. Lemma 9 shows that the trace trees and the unwindings of the labeled Kripke structure $(T_{\mathscr{K}}, v_{\mathscr{K}})$ coincide. Thus, $\mathscr{L}(A_{\neg\phi}) \cap \mathscr{L}_{t,p}(\beta) = \emptyset$ iff the corresponding trace tree satisfies ϕ. This is the case iff all paths satisfy ϕ, i.e., $\forall \xi = t, p, \ldots$ flow chain of $\beta : \beta, \sigma_F(\xi) \models_{\mathtt{CTL}^*} \phi$ hold. □

Note that the selection of correct words regarding the stuttering, in this lemma selected by the intersection with $\mathscr{L}_{t,p}(\beta)$, is solved in the construction of Petri net $\mathcal{N}^>$ in the next section and in the construction of the corresponding formula $\psi^>$ in the next but one section.

5.3.2 From Petri Nets with Transits to Petri Nets

In Step 2, we construct for the Petri net with transits $\mathcal{N}_T$ and the Büchi automata $A_{\neg\phi_i}$ for each flow subformula $\varphi_{\mathbb{A}i} = \mathbb{A}\,\phi_i$ of the Flow-CTL* formula φ, a Petri net $\mathcal{N}^>$ by composing a copy of $\mathcal{N}_T$ (without transits), denoted by $\mathcal{N}_O^>$, with subnets $\mathcal{N}_i^>$ corresponding to $A_{\neg\phi_i}$ such that each copy is sequentially triggered when a transition of $\mathcal{N}_O^>$ fires. In this section we give a complete, but more intuitive definition of the construction for a lighter weight understanding. Definition 31 in Sec. 5.4 on page 75 is a fully formal definition of this construction.

We divide the construction of $\mathcal{N}^>$ into the construction of the *original part* of the net, the construction of the *subnets*, the part that belong to the additional elements needed for a correct *stuttering* of the system, and the part that belong to properly *connecting* the single subnets:

Original Part: The subnet $\mathcal{N}_O^>$ with places $\mathcal{P}_O^>$ and transitions $\mathcal{T}_O^>$ is a copy of the Petri net with transits $\mathcal{N}_T$ without any transits. Furthermore, there is an activation place $\overrightarrow{a_o} \in \mathcal{P}_O^>$ for composing the subnets in a sequential manner. Finally, we have two places $\boldsymbol{\mathcal{N}}, \boldsymbol{\mathcal{S}} \in \mathcal{P}_O^>$ and a transition $t_{\boldsymbol{\mathcal{N}} \to \boldsymbol{\mathcal{S}}}$ switching into the *global* stuttering mode and the corresponding transition $t_{\mathfrak{s}}$ informing the subnets about the stuttering.

Subnets: The subnet $\mathcal{N}_i^>$, with places $\mathcal{P}_i^>$ and transitions $\mathcal{T}_i^>$, when triggered by transitions $t \in \mathcal{T}$, guesses nondeterministically the violating flow tree of the operator $\mathbb{A}$

and simulates $A_{\neg\phi_i}$. Thus, a token from the initially marked place $[\iota]_i$ is moved via one transition for each transition and place combination starting a flow chain to the place corresponding to the respective initial state of $A_{\neg\phi_i}$. For each state s of $A_{\neg\phi_i}$, we have a place $[s]_i$, and, for each edge (s, l, s'), there is a transition labeled by l which moves the token from $[s]_i$ to $[s']_i$. To also allow the checking of this specific flow tree to be skipped (and maybe check another one created later) there is one transition $[t_{\Rrightarrow}]_i$ for each transition $t \in \mathscr{T}$ with $[\iota]_i$ in its pre- and postset. These transitions are also used to skip transitions not starting a flow chain in case no tracking of a tree started yet. We have one activation place $[\overrightarrow{a}]_i$ and one global stuttering place $[\overrightarrow{\mathfrak{s}}]_i$ for each subnet and additionally there is one activation place $[\overrightarrow{t}]_i$ for each transition $t \in \mathscr{T}$. These are used to connect the subnets in a sequential manner.

Stuttering: There are two types of stuttering: *global* stuttering for finite runs and *local* stuttering for finite flow chains. To guess the starting point of the *global* stuttering, the token in the initially marked place $\boldsymbol{\mathcal{N}}$ of the original part can be move into the place $\boldsymbol{\mathcal{S}}$ via transition $t_{\boldsymbol{\mathcal{N}}\to\boldsymbol{\mathcal{S}}}$. This can be done to any point in time. The standard transitions $t \in \mathscr{T}$ can only fire in normal mode (place $\boldsymbol{\mathcal{N}}$ is occupied) and the global stuttering transition $t_{\mathfrak{s}}$ only in stuttering mode (place $\boldsymbol{\mathcal{S}}$ is occupied). For the subnets, the global stuttering transitions (labeled with $\mathfrak{s}$) are dependent on $[\overrightarrow{\mathfrak{s}}]_i$. The *local* stuttering transitions of a subnet $\mathscr{N}_i^>$ corresponding to edges of $A_{\neg\phi_i}$ with label $\mathfrak{s}_p$ with $p \in \mathscr{P}$, can fire whenever the subnet is activated. We omit runs where such transitions fire infinitely often consecutively with the formula $\psi^>$ defined in Sec. 5.3.3.

Connecting: The original part $\mathscr{N}_O^>$ and the subnets $\mathscr{N}_i^>$ are connected in a sequential manner. The net $\mathscr{N}_O^>$ has an initially marked activation place $\overrightarrow{a_o}$ in the preset of each transition, the subnets have one activation place $[\overrightarrow{t}]_i$ in the preset of every transition $t^>$ corresponding to a transition $t \in \mathscr{T}$. The transitions move the activation token to the corresponding places of the next subnet (or back to $\mathscr{N}_O^>$). The skipping transitions $[t_{\Rrightarrow}]_i$ for each transition $t \in \mathscr{T}$ can only fire when there is still a token in $[\iota]_i$ and they just move the activation tokens to the next subnet. For the *global* stuttering, the transition $t_{\mathfrak{s}}$ in $\mathscr{N}_O^>$ puts the activation token into the place $[\overrightarrow{\mathfrak{s}}]_1$ and each $\mathfrak{s}$-labeled transition of the subnets moves this token to the next subnet (or back to $\mathscr{N}_O^>$). For the *local* stuttering we take and directly put back the activation token into place $[\overrightarrow{a}]_i$ to allow for switching into the local stuttering mode whenever the subnet is activated.

By that, we can check the acceptance of each $A_{\neg\phi_i}$ by checking if the subnet infinitely often reaches any places corresponding to a Büchi state of $A_{\neg\phi_i}$. This and only allowing to correctly guess the time point of the stutterings is achieved with the formula described in Sec. 5.3.3. A formal definition of this construction is given in Definition 31 on page 75

▶ **Example 5.** In Fig. 5.5 we give a schematic overview of this construction for the Petri net with transits depicted in Fig. 2.4 and n flow subformulas. The original part $\mathscr{N}_O^>$ of the constructed Petri net $\mathscr{N}^>$ is depicted on the left. The indicated subnet $\mathscr{N}_T'$ is a copy of the Petri net with transits $\mathscr{N}_T$ without the transits. At the bottom of the figure the

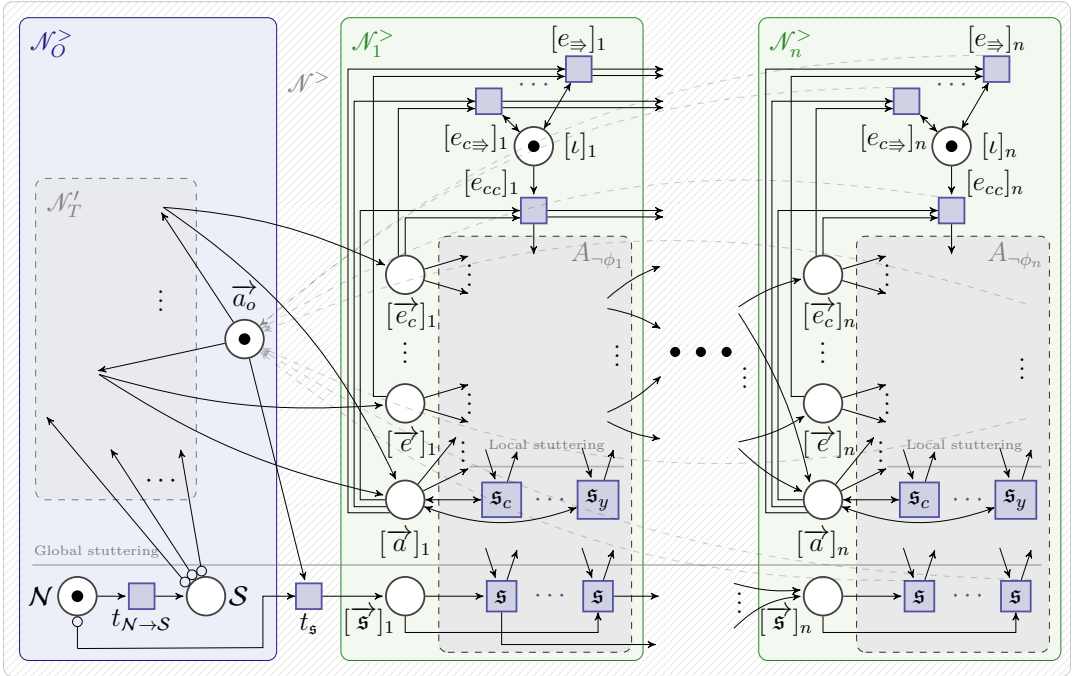


Fig. 5.5: *A schematic overview of the construction of the Petri net $\mathcal{N}^>$ from the Petri net with transits $\mathcal{N}_T$ depicted in Fig. 2.4 and n flow subformulas. The presentation concentrates on the sequential composition of the subnets at the left of each subnet, the guessing of the considered data flow tree at the top, and the global stuttering at the bottom. The original part of the net $\mathcal{N}_O^>$ is depicted at the left with the input net without transits $\mathcal{N}_T'$. The subnets $\mathcal{N}_i^>$ for each flow subformula contain the places and transitions corresponding to the Büchi automaton $A_{\neg\phi_i}$.*

global stuttering is depicted. Each transition of $\mathcal{N}_T'$ is dependent on the emptiness of place $\mathcal{S}$ due to the indicated inhibitor arcs. Furthermore, it takes the activation token in $\overrightarrow{a_o}$ and moves it to both, the activation place of the first subnet $[\overrightarrow{a}]_1$ and the activation place corresponding to the firing of this transition. At the top of each subnet $\mathcal{N}_i^>$ the guessing of the data flow tree happens. Whenever it is the turn of the subnet and transition e_c has fired, a token can be put into the corresponding initial state of the Petri net simulating $A_{\neg\phi_i}$. If this creation of a data flow tree should not be checked, or any other transition has fired, the skipping transitions can be used to pass the token to the next subnet. The global stuttering is depicted at the bottom of the figure and the switches for the local stuttering of each subnet are shown above. The last subnet puts the activation token back to the activation place $\overrightarrow{a_o}$ of $\mathcal{N}_O^>$ with the gray dashed arcs. The different representation of these arcs is only for the sake of clarity. ◀

The size of the constructed Petri net is dominated by the respective single- or double-exponential size of the nondeterministic Büchi automata.

▶ **Lemma 6 (Size of the Constructed Net).** *The constructed Petri net with inhibitor arcs $\mathcal{N}^>$ for a Petri net with transits $\mathcal{N}_T$ and n nondeterministic Büchi automata*

$A_{\neg\phi_i} = (\mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}, Q_i, I_i, E_i, F_i)$ *has* $\boldsymbol{O}(|\mathscr{N}_T| \cdot n + |\mathscr{N}_T| + \sum_{i=1}^{n} |Q_i|)$ *places and* $\boldsymbol{O}(|\mathscr{N}_T|^2 \cdot n + |\mathscr{N}_T| + \sum_{i=1}^{n} |E_i|)$ *transitions.* ◀

The main parts here (the sum over the sizes of the states Q_i and the sum over the sizes of the edges E_i) directly follow from the construction of the subnet simulating the automata $A_{\neg\phi}$. For more details see the proof on page 77.

5.3.3 From Flow-CTL* Formulas to LTL Formulas

The formula transformation from a given Flow-CTL* formula φ and a Petri net with transits $\mathscr{N}_T$ into an LTL formula (Step 3) consists of three parts and is done via substitutions. An overview of the used subformula substitutions is given in Tab. 5.2.

First, we substitute the flow formulas $\varphi_{\mathbb{A}_i} = \mathbb{A}\,\phi_i$ with the acceptance check of the corresponding automaton $A_{\neg\phi_i}$, i.e., we substitute $\varphi_{\mathbb{A}_i}$ with $\neg\Box\Diamond\bigvee_{b\in F_i}[b]_i$ for the Büchi states F_i of $A_{\neg\phi_i}$.

Second, the sequential manner of the constructed net $\mathscr{N}^{>}$ requires an adaptation of the run part of φ. For a subformula $\psi_1\,\mathcal{U}\,\psi_2$ with transitions $t \in \mathscr{T}$ as atomic propositions or a subformula $\bigcirc\,\psi$ in the run part of φ, the sequential steps of the subnets have to be skipped. Let $\mathscr{T}_O^{>}$ be the transitions of the original part $\mathscr{N}_O^{>}$, $t_{\mathcal{N}\to\mathcal{S}}$ the transition switching $\mathscr{N}_O^{>}$ from normal to stuttering mode, and $\mathscr{T}_{O^-}^{>} = \mathscr{T}_O^{>} \setminus \{t_{\mathcal{N}\to\mathcal{S}}\}$. Then, because of the ingoing semantics, we can select all states corresponding to the run part with $\mathtt{M} = \bigvee_{t\in\mathscr{T}_{O^-}^{>}} t$ together with the initial state $\mathtt{i} = \neg\bigvee_{t\in\mathscr{T}^{>}} t$. Hence, we replace each subformula $\psi_1\,\mathcal{U}\,\psi_2$ containing transitions $t \in \mathscr{T}$ as atomic propositions with $((\mathtt{M}\vee\mathtt{i}) \to \psi_1)\,\mathcal{U}((\mathtt{M}\vee\mathtt{i}) \wedge \psi_2)$ from the inner- to the outermost occurrence. For the next operator, the second state is already the correct next state of the initial state also in the sense of the global timeline of $\psi^{>}$. For all other states belonging to the run part (selected by the until construction above), we have to get the next state and then skip all transitions of the subnet. Thus, we replace each subformula $\bigcirc\,\psi$ with $(\mathtt{i} \to \bigcirc\,\psi) \wedge (\neg\mathtt{i} \to \bigcirc(\bigvee_{t\in\mathscr{T}^{>}\setminus\mathscr{T}_{O^-}^{>}} t\,\mathcal{U}\bigvee_{t'\in\mathscr{T}_{O^-}^{>}} t' \wedge \psi))$ from the inner- to the outermost occurrence.

Third, we need to make sure that we only consider runs that trigger the automaton so that the correctness result of the automaton $A_{\neg\phi_i}$ can be used, i.e., we only consider runs such that the language of the automaton is restricted to $\mathscr{L}_{t,p}(\beta)$. For the global stuttering, the structure of the constructed net $\mathscr{N}^{>}$ ensures that at the end of finite firing sequences only stuttering transitions are allowed, because no activation place $[\overrightarrow{t}]_i$ can get occupied after switching into stuttering mode. With $\Box\Diamond\overrightarrow{a_o}$ we can select runs which infinitely often visit the original part of the net. For $post^{\Upsilon}(p) = \{t \in post(p) \mid \exists q \in \mathscr{P} : (p,q) \in \Upsilon(t)\}$ we select with

$$\mathtt{stutt}_s = \Box \bigwedge_{i\in\{1,\dots,n\},p\in\mathscr{P}} ([\mathfrak{s}_p]_i \to \Box\neg \bigvee_{t\in post^{\Upsilon}(p)} [t]_i)$$

runs obeying: when once switched into the local stuttering mode for some chain, no transition extending this chain is ever fired. The other direction, i.e., that we eventually

Tab. 5.2: *Subformula substitutions in the process of creating $\psi^{>}$ from φ. The rules in the second and third row are applied from the inner- to the outermost occurrence. The rule in row two is only applied when ψ_1 or ψ_2 contain transitions as atomic propositions.*

φ	$\psi^{>}$
$\mathbb{A}\,\phi_i$	$\neg\Box\Diamond\bigvee_{b\in F_i}[b]_i$
$\psi_1\,\mathcal{U}\,\psi_2$	$((\mathtt{M}\vee\mathtt{i})\rightarrow\psi_1)\,\mathcal{U}\,((\mathtt{M}\vee\mathtt{i})\wedge\psi_2)$
$\bigcirc\,\psi$	$(\mathtt{i}\rightarrow\bigcirc\,\psi)\wedge(\neg\mathtt{i}\rightarrow\bigcirc(\bigvee_{t\in\mathcal{T}^{>}\setminus\mathcal{T}^{>}_{O^-}}t\,\mathcal{U}\bigvee_{t'\in\mathcal{T}^{>}_{O^-}}t'\wedge\psi))$

switch into the local stuttering mode, when no transition extending this chain will ever fire, is done with

$$\mathtt{stutt}_c=\Box\bigwedge_{i\in\{1,\dots,n\},p\in\mathcal{P},q_p\in Q_p}(([q_p]_i\wedge\Box\neg\bigvee_{t\in post^{\Upsilon}(p)}[t]_i)\rightarrow\Diamond[\mathfrak{s}_p]_i)$$

for $Q_p=\{(Q_r,\cdot)\in Q_i\mid(p,\cdot)\in Q_r\vee((t,p),\cdot)\in Q_r\}$. So for each subnet we consider the states belonging to some place p and when no transition extending the data flow in p is ever fired again, then eventually the corresponding transition switching into the stuttering mode $\mathfrak{s}_p$ has to fire.

Hence, we obtain the final formula

$$\psi^{>}=((\Box\Diamond\overrightarrow{a_o})\wedge\mathtt{stutt}_s\wedge\mathtt{stutt}_c)\rightarrow\psi'$$

by only selecting the runs where the original part is infinitely often activated and each subnet chooses its stuttering modes correctly. The LTL formula ψ' is obtained from φ by applying the rules defined above. Since the size of the formula depends on the size of the constructed Petri net $\mathcal{N}^{>}$, it is also dominated by the Büchi automaton construction.

▶ **Lemma 7 (Size of the Constructed Formula).** *The size of the constructed formula $\psi^{>}$ is double-exponential for specifications given in CTL* and single-exponential for specifications in CTL.* ◀

Proof. The next and the until replacements introduces disjunctions over all transitions of the net, i.e., especially over the edges of the Büchi automaton. Furthermore, the stuttering subformula $\mathtt{stutt}_c$ ranges over the states of the Büchi automaton, i.e., about the corresponding places in $\mathcal{P}^{>}$. However, none of the formulas introduce a further exponent. Hence, the size of the formula depends on the number of transitions $\mathcal{T}^{>}$ and the number of places $\mathcal{P}^{>}$ and is therewith double-exponential for specifications in CTL* and single-exponential for specifications in CTL due to Lemma 6 and Lemma 4. □

We can show that the construction of the net and the formula adequately fit together such that the additional sequential steps of the subnets are skipped in the formula and the triggering of the subnets simulating the Büchi automata as well as the stuttering is handled properly. The correctness of the transformation is based on the correctness of the Kripke structure and the automata constructions. The constructed formula and the constructed net together ensure the correct triggering of the automata.

▶ **Lemma 8 (Correctness of the Transformation).** *For a safe Petri net with transits $\mathcal{N}_T$ and a Flow-CTL* formula φ, there exists a safe Petri net $\mathcal{N}^>$ with inhibitor arcs and an LTL formula $\psi^>$ such that $\mathcal{N}_T \models \varphi$ iff $\mathcal{N}^> \models_{\text{LTL}} \psi^>$.* ◀

Proof Sketch. The main part of this proof is done in Lemma 5 by showing that the constructed Büchi automaton accepts a firing sequence of the run with correct stuttering transitions iff all corresponding flow chains of the run satisfy the CTL* formula. By checking all runs of the net $\mathcal{N}^>$ and the nondeterministic guessing of the violating flow tree in each subnet we check all possible flow trees. The run which never decides to track any tree for a subnet introduces no problems because the initial place $[\iota]_i$ cannot be part of the Büchi states. The remaining part is to show that, on the one hand, we only consider firing sequences triggering the automaton with a correct stuttering and, on the other hand, that the run formula of the Flow-CTL* formula φ is properly replaced for $\psi^>$ such that the additional steps for the sequentially triggering of the subnets corresponding to the Büchi automata are omitted in the evaluation. The former is done with the premise of $\psi^>$ and the latter is due to adequately replacing all elements of φ concerning the timeline. This can be shown by translating the counterexamples of the contraposition of the statement with a nested structural induction over φ. For more details see the proof on page 78. □

The complexity of the model checking problem of Petri nets with transits and Flow-CTL* is dominated by the automata constructions for the CTL* subformulas. The alternation removal (Step (iv) of the construction) is due to the checking of branching properties on structures chosen by linear properties. In contrast to standard CTL* model checking on a static Kripke structure, we check on Kripke structures dynamically created for specific runs.

▶ **Theorem 3.** *A safe Petri net with transits $\mathcal{N}_T$ can be checked against a Flow-CTL* formula φ in triple-exponential time in the size of $\mathcal{N}_T$ and φ. For a Flow-CTL formula φ', the model checking algorithm runs in double-exponential time in the size of $\mathcal{N}_T$ and φ'.* ◀

Proof. For a Petri net with transits $\mathcal{N}_T$ and a Flow-CTL* formula φ, Lemma 6 and Lemma 7 yield the double-exponential size of the constructed Petri net with inhibitor arcs $\mathcal{N}^>$ and the constructed formula $\psi^>$ (the single-exponential size for the CTL fragment). Lemma 8 yields the correctness. Checking a safe Petri net against LTL can be seen as checking a Kripke structure of exponential size (due to the markings of the net). Since this can be checked in linear time in the size of the state space and in exponential time in the size of the formula [CHVB18], we obtain a triple-exponential algorithm for CTL* formulas and a double-exponential algorithm for CTL formulas in the size of the net and the formula. □

Note that a single-exponential time algorithm for the fragment Flow-LTL is presented in Chap. 6.

5.4 Proofs and Formal Constructions

In this section we provide more details about the model checking procedure for Petri net with transits against Flow-CTL* presented in the previous section. In particular, we provide the full proof for the correctness of the automaton construction (Sec. 5.4.1) and the formal construction of the standard Petri net $\mathcal{N}^>$ of a given Petri net with transits and the number of flow subformulas of a CTL* formula and its correctness proof (Sec. 5.4.2). Those parts have been initially omitted for increasing the readability of the previous section.

5.4.1 Correctness of the Automaton Construction

In this section we provide details for the construction of the Büchi automaton $A_{\neg\phi}$ for a CTL* formula ϕ presented in Sec. 5.3.1. Especially, we provide the detailed proof for the key part of the construction, i.e., the correctness of the construction of the product automaton (Lemma 3).

First, we provide the formal correspondence between the flow trees and the unwindings along some firing sequence of the labeled Kripke structure defined in Definition 26.

▶ **Lemma 9 (Coincidence of Trace Tree and Unwindings).** *Let $\beta = (\mathcal{N}_T^R, \rho)$ be a run of a Petri net with transits $\mathcal{N}_T$. For each flow tree (T, v) of β there is a corresponding tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ obtained by the unwinding of $\mathscr{K}_{\mathcal{N}_T,AP}$ along some firing sequence ζ with the atomic propositions AP in Definition 26 and vice versa. For these trees*

- $\sigma_T(T) = T_{\mathscr{K}}$ *and*
- $\forall n \in \sigma_T(T) : v(n) = (t, p) \rightarrow v_{\mathscr{K}}(n) = \{t, p\} \cap AP$

holds. ◀

Proof. Given run $\beta = (\mathcal{N}^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$. Each flow tree (T, v) of β starts with some transition $t \in \mathscr{T}^R$ in a place $p \in \mathscr{P}^R$ with $(\triangleright, p) \in \Upsilon^R(t)$. Taking an arbitrary covering firing sequence ζ, Definition 26 creates the corresponding tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$. Since the firing sequence covers the run, it must contain transition t. For the other direction we know that not more trees are created by Definition 26 because the inductive definition does not do anything for transitions in the firing sequence which does not transit the current data flow in a node. Since we only consider covering firing sequences all sequences create the same tree.

The definition of flow trees (Definition 19) yields that there are only children of a node corresponding to the transiting of the data flow from this node. These children are ordered according to the postset places of the transit. The relation R ensures that also for $T_{\mathscr{K}}$ only those children are added in the induction step of Definition 26, and there also the same ordering is ensured. When there is no transition in the firing sequence left transiting the data flow in the current node, the induction step of Definition 26 switches into the stuttering mode, where only loops are available. Thus, only one child is added.

This corresponds to the situation of a finite branch of the flow tree. These are extended by the σ_T function also with an infinite sequence of single children. Thus, $\sigma_T(T) = T_{\mathscr{K}}$.

The labeling is equally easy to see because $T_{\mathscr{K}}$ is labeled corresponding to the name of the current state of the Kripke structure. These names are exactly the ones corresponding to the data flow in the current state, but only restricted to the set of atomic propositions AP. □

That the size of the constructed alternating word automaton $\mathscr{A}_\phi$ in Definition 29 is single-exponential for specifications in CTL* and polynomial for specifications in CTL, easily follows from the size of the corresponding alternating tree automaton and the size of the labeled Kripke structure and the definition of the alternating word automaton as product automaton. The corresponding lemma can be found on page 60.

Proof (Lemma 2: Size of the Product Automaton): Theorem 1 states the size of the corresponding alternating tree automaton $T_{\mathscr{D},\phi}$ with $\mathcal{O}(|\mathscr{D}| \cdot 2^{|\phi|})$ for specifications in CTL* and $\mathcal{O}(|\mathscr{D}| \cdot |\phi|)$ for specifications in CTL. The set of degrees $\mathscr{D}$ for this construction belong to the branching possibilities of the data flow $\mathscr{D} = \{|post^{\Upsilon}(p,t)| \mid p \in \mathscr{P} \wedge t \in post^{\mathcal{N}_T}(p)\}$. Thus, we can maximally branch in all places $p \in \mathscr{P}$. Hence, at most $|\mathscr{D}| \leq |\mathscr{P}|$. Lemma 1 states that the Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP}$ has $\mathcal{O}(|AP \cap \mathscr{T}| \cdot |\mathcal{N}_T| + |\mathcal{N}_T|)$ states. Hence, the product yields that the number of states is in $\mathcal{O}(|\mathscr{P}| \cdot 2^{|\phi|} \cdot (|\mathcal{N}_T| \cdot |AP \cap \mathscr{T}| + |\mathcal{N}_T|)) = \mathcal{O}(2^{|\phi|} \cdot |\mathcal{N}_T|^3)$ for CTL* specification and in $\mathcal{O}(|\phi| \cdot |\mathcal{N}_T|^3)$ for CTL specifications for the product automaton. □

A key part of the correctness of the model checking procedure is the correctness of the automaton checking the CTL* specifications for the data flow subformulas. Lemma 3 on page 62 provides this correctness such that for a given run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$, a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ with $(\triangleright, p) \in \Upsilon^R(t)$, a CTL* formula ϕ with atomic propositions AP, the alternating tree automaton $T_{\mathscr{D},\phi}$ corresponding to Theorem 1, and the 2^{AP}-labeled $\mathscr{D}$-tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ created by Definition 26 from the labeled Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP}$, a restriction of the language of the alternating word automaton $\mathscr{A}_\phi$ defined in Definition 29 $(\mathscr{L}(\mathscr{A}_\phi) \cap \mathscr{L}_{t,p}(\beta))$ is empty if and only iff $(T_{\mathscr{K}}, v_{\mathscr{K}})$ is accepted by the alternating tree automaton $T_{\mathscr{D},\phi}$.

Proof (Lemma 3: Correctness of the Product Automaton): We prove the lemma along the proof for the product automaton of a standard Kripke structure and an alternating tree automaton for CTL* specifications presented in [KVW00].

Case $(T_{\mathscr{K}}, v_{\mathscr{K}})$ is accepted by $T_{\mathscr{D},\phi}$. Hence, there is an accepting run (T_r, v_r) of $T_{\mathscr{D},\phi}$ over $(T_{\mathscr{K}}, v_{\mathscr{K}})$. We now construct an infinite word $w \in \mathscr{L}(\mathscr{A}_\phi) \cap \mathscr{L}_{t,p}(\beta) \subseteq (\mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\})^\omega$ and an accepting run (T'_r, v'_r) of $\mathscr{A}_\phi$ on w. For this proof we have four different kind of trees:

- $(T_{\mathscr{K}}, v_{\mathscr{K}})$ – a 2^{AP}-labeled tree corresponding to a flow tree and constructed in Definition 26 from the labeled Kripke structure,
- $(T'_{\mathscr{K}}, v'_{\mathscr{K}})$ – a S-labeled tree containing the same tree as above, but labeled with the states of the Kripke structure,

- (T_r, v_r) – a $\mathbb{N}^* \times Q_\phi$-labeled tree representing the accepting run of $T_{\mathscr{D},\phi}$ on $(T_{\mathscr{K}}, v_{\mathscr{K}})$ for the formula ϕ, and
- (T'_r, v'_r) – a $S \times Q_\phi$-labeled tree representing the constructed run accepted by the product automaton $\mathscr{A}_\phi$ on the constructed word w.

Consider the firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ and the index $i \in \mathbb{N}$ such that $t = t_i$ from the construction of $(T_{\mathscr{K}}, v_{\mathscr{K}})$ (Definition 26) with the corresponding S-labeled tree $(T'_{\mathscr{K}}, v'_{\mathscr{K}})$. We define w and the $S \times Q_\phi$-labeled $\mathscr{D}$-tree (T'_r, v'_r) inductively over the length of ζ and associate nodes of the tree T'_r of the accepting run of $T_{\mathscr{D},\phi}$ to tree nodes in $\mathscr{T}'_r$ in each step. Intuitively, we pump up T_r by adding stutter steps for concurrent transitions and the beginning of the local stuttering. This makes no harm, because in such cases in the product automaton only the Kripke structure moves and not the tree automaton.

Initially, $w_0 = \rho(t_i)$ and the root $\varepsilon \in T'_r$ is labeled with $v'_r(\varepsilon) = (v'_{\mathscr{K}}(\varepsilon), q_0)$ and we associate $\varepsilon \in T_r$ to $\varepsilon \in T'_r$ for this step.

For the induction step let t_{j-1} be the last considered transition of ζ, w' the so far created prefix of w, and $n_0, \ldots, n_k \in T_r$ the nodes associated in the previous step to the leafs $n'_0, \ldots, n'_k \in T'_r$ of the current tree. First, we extend the current input word with the next transition: $w = w' \cdot \rho(t_j)$. Then, we extend the current run (T'_r, v'_r). For each node n_l with $l \in \{0, \ldots, k\}$ and label $v_r(n_l) = (x, q) \in \mathbb{N}^* \times Q_\phi$ we have the state $s = v'_{\mathscr{K}}(x) \in S$ of the Kripke structure for which the copy of $T_{\mathscr{D},\phi}$ in state q reads the tree obtained by unwinding $\mathscr{K}$ from s along ζ.

If a) $\mathfrak{s}_{\mathscr{K}}(s, \rho(t_j)) = \emptyset$ then we add one child to T'_r with $v'_r(n'_l \cdot 0) = (s, q)$ and associate the node n_l to this new node. Note that the run property for this new node is satisfied, due to the self-loop condition of the product automaton.

Case b) $\mathfrak{s}_{\mathscr{K}}(s, \rho(t_j)) \neq \emptyset$. Definition 26 yields that $d(x) = |\mathfrak{s}_{\mathscr{K}}(s, \rho(t_j))|$ holds. Let $\delta_\phi(q, v_{\mathscr{K}}(x), d(x)) = \theta$. Since (T_r, v_r) is a run there is by definition a set $Y = \{(c_0, q_0), \ldots, (c_m, q_m)\} \subseteq \{0, \ldots, d(x) - 1\} \times Q_\phi$ such that $Y \models \theta$ and for all $0 \leq i \leq m$ we have $n_l \cdot i \in T_r$ and $v_r(n_l \cdot i) = (x \cdot c_i, q_i)$. We now add for all $0 \leq i \leq m$ nodes $n'_l \cdot i \in T'_r$ and label them with $v'_r(n'_l \cdot i) = (v'_{\mathscr{K}}(x \cdot c_i), q_i)$. We associate the corresponding nodes for the next step. Again we obey the definition of children for a run, because the set of the children's labels $Y' = \{v'_r(n'_l \cdot i) \mid 0 \leq i \leq m\} = \{(v'_{\mathscr{K}}(x \cdot c_i), q_i) \mid 0 \leq i \leq m\}$ is due to Definition 26 $Y' = \{(\langle \mathfrak{s}_{\mathscr{K}}(s, \rho(t_j))\rangle_{c_i}, q_i) \mid 0 \leq i \leq m\}$. For $\delta(v'_r(n'_l), \rho(t_j)) = \delta((s, q), \rho(t_j)) = \theta'$ we know that θ' is obtained from θ by replacing (c_i, q_i) with $(\langle \mathfrak{s}_{\mathscr{K}}(s, \rho(t_j))\rangle_{c_i}, q_i)$ due to the definition of δ. Hence, $Y' \models \theta'$.

Now, we check whether any of the newly reached nodes n of T_r has a child c which belongs to the start of a local stuttering, i.e., for $v_r(c) = (x_c, q_c) \in \mathbb{N}^* \times Q_\phi$ and $v_r(n) = (x_n, q_n) \in \mathbb{N}^* \times Q_\phi$ we check whether $s_c = v'_{\mathscr{K}}(x_c) \in \{p'_{\mathfrak{s}} \mid p' \in \mathscr{P}\} \subseteq S$ and $v'_{\mathscr{K}}(x_n) \notin \{p'_{\mathfrak{s}} \mid p' \in \mathscr{P}\} \subseteq S$. If this is the case for the nodes $n^0, \ldots, n^m$, we extend the word for each n^o for $o \in \{0, \ldots, m\}$ with the switch into the stuttering mode: $w = w' \cdot \mathfrak{s}_{p'}$ for the corresponding state $s^o = v'_{\mathscr{K}}(x_o) = p'_{\mathfrak{s}}$ for $v_r(n^o) = (x_o, q_o)$ and we extend the current run (T'_r, v'_r) similar to the cases a) and b) but instead of $\rho(t_j)$ we use the label $\mathfrak{s}_{p'}$. In this situation case b) is simplified because there is no nondeterminism for stuttering transitions, hence, the degree of the nodes is one. This concludes the induction step.

Finally, we add the global stuttering in case that ζ is finite. Thus, we infinitely often extend the word $\boldsymbol{w}$ with $\mathfrak{s}$ and again apply the step analog to case b) with $\mathtt{s}_{\mathscr{K}}(s, \mathfrak{s})$. We do not have to consider steps corresponding to case a) because for finite firing sequences we have to have switched into a stuttering place and each of these states of the labeled Kripke structure has a stutter edge.

Since each of the steps obey the run properties, (T'_r, v'_r) is a run of $\mathscr{A}_\phi$ on $\boldsymbol{w}$. Furthermore, $\boldsymbol{w} \in \mathscr{L}_{t,p}(\beta)$ holds, because we stutter for finite firing sequences at the end, we add the local stuttering transitions corresponding to Definition 26, i.e., directly at the end of finite flow chains, and all other letters correspond to the firing sequence.

It remains to show that the run (T'_r, v'_r) is accepted by $\mathscr{A}_\phi$. This directly follows from the transferring of the acceptance condition from $T_{\mathscr{D},\phi}$ to $\mathscr{A}_\phi$ on the cross product of the acceptance sets and $T_{\mathscr{D},\phi}$ accepting (T'_r, v'_r). Since δ moves the second component for each edge as for δ_ϕ the only interesting case is the additional loop for $\mathtt{s}_{\mathscr{K}}(s, l) = \emptyset$. Since we only consider acceptance conditions over infinite words, this edge would only be a problem for the acceptance, when it is taken infinitely often consecutively. This cannot happen for (T'_r, v'_r), because otherwise we would have an infinite sequence of transitions not extending some chain, this means this chain is finite and thus we must have switched into some $q_{\mathfrak{s}}$ state due to Definition 26. But in these states we have loops for every $t \in \mathscr{T}$.

Case $\mathscr{L}(\mathscr{A}_\phi) \cap \mathscr{L}_{t,p}(\beta) \neq \emptyset$. Thus, there is $\boldsymbol{w} \in \mathscr{L}_{t,p}(\beta)$ which is accepted by $\mathscr{A}_\phi$. Let (T'_r, v'_r) be the accepted run of $\mathscr{A}_\phi$ on $\boldsymbol{w} \in \mathscr{L}_{t,p}(\beta)$. We have to show that $(T_{\mathscr{K}}, v_{\mathscr{K}})$ is accepted by $T_{\mathscr{D},\phi}$. Hence, we create a run (T_r, v_r) on $(T_{\mathscr{K}}, v_{\mathscr{K}})$ which is accepted by $T_{\mathscr{D},\phi}$. We do this inductively over $\boldsymbol{w}$ and skip the concurrent transitions and the local stuttering steps not involving the current branch. The decisions are taken with respect to the first component (the states of the Kripke structure) of the labels of (T'_r, v'_r).

Initially, for $w_0 = \rho(t)$, we define $v_r(\varepsilon) = (\varepsilon, q_0)$.

For the induction step let $\boldsymbol{w}_{i-1}$ be the last letter processed by this definition, $n'_0, \ldots, n'_m \in T'_r$ be the corresponding nodes, and $n_0, \ldots, n_m \in T_r$ the nodes of T_r associated to these nodes (exactly the leafs of T_r). Consider one of these nodes n'_j for $j \in \{0, \ldots, m\}$ with $v'_r(n'_j) = (s, q)$ and $v_r(n_j) = (x, q)$.

Case $\boldsymbol{w}_i \in \mathscr{T} \cup \{\mathfrak{s}_{p'} \mid p' \in \mathscr{P}\}$:

If a) $\mathtt{s}_{\mathscr{K}}(s, \boldsymbol{w}_i) = \emptyset$, thus, $n'_j \cdot 0$ is the only child with $v'_r(n'_j \cdot 0) = (s, q)$ due to the self-loop in the definition of δ, we add nothing to the tree T_r and only associate n_j to $n'_j \cdot 0$. Thus, we skip the concurrent transitions and the start of the local stuttering, when the branch is not impacted by the transition, the local stuttering, or is already in stuttering mode and $w_i \in \{\mathfrak{s}_{p'} \mid p' \in \mathscr{P}\}$.

Case b) $\mathtt{s}_{\mathscr{K}}(s, \boldsymbol{w}_i) \neq \emptyset$. Let $n'_j \cdot k \in T'_r$ be the $k+1$ children of n'_j with $v'_r(n'_j \cdot k) = (s'_k, q'_k)$. We add successors $n_j \cdot k \in T_r$ to T_r and label them with $v_r(n_j \cdot k) = (x \cdot i_k, q'_k)$ for $i_k \in \mathbb{N}$ such that $v'_{\mathscr{K}}(x \cdot i_k) = s'_k$. This means that we label the new child with the successor state of the tree $T_{D,\phi}$ (i.e., q'_k), and the node of the input tree $(T_{\mathscr{K}}, v_{\mathscr{K}})$ labeled with the successor state of the Kripke structure $\mathscr{K}_{\mathcal{N}_T,AP}$ (i.e., s'_k). This obeys the property of being a run because we have that $\delta_\phi(q, v_{\mathscr{K}}(x), d(x)) = \delta_\phi(q, \ell(s), |\mathtt{s}_{\mathscr{K}}(s, \boldsymbol{w}_i)|) = \theta$ and

due to δ we know that there is an edge $\delta((s,q),\mathfrak{w}_i) = \theta'$ where θ' is obtained from θ by replacing the atoms (c,q'_k) with $(\langle \mathfrak{s}_{\mathscr{K}}(s,\mathfrak{w}_i)\rangle_c, q'_k)$. Since (T'_r, v'_r) is a run, the set of all (s'_k, q'_k) satisfies θ'. Due to the replacement and that in $(T'_{\mathscr{K}}, v'_{\mathscr{K}})$ the children $x \cdot i_k$ are also ordered according to the transits, we have that the set of all (i_k, q'_k) satisfies θ.
Case $\underline{\mathfrak{w}_i = \mathfrak{s}}$:

Due to $\mathfrak{w} \in \mathscr{L}_{t,p}(\beta)$ this case can only happen at the end of finite firing sequences. Furthermore, for finite firing sequences each chain must be finite. Thus, due to $\mathfrak{w} \in \mathscr{L}_{t,p}(\beta)$, there must be an index $j < i$ such that for the transition $\mathfrak{s}_{p'}$ corresponding to the finite chain, $\mathfrak{w}_j = \mathfrak{s}_{p'}$ holds. hence all branches switched into the stuttering mode. Due to the $\mathfrak{s}$-labeled loops in such states in the definition of the labeled Kripke structure $\mathscr{K}_{\mathscr{N}_T,AP}$ we can apply case b) infinitely often.

The acceptance of the run (T_r, v_r) again directly follows from defining the acceptance condition of $\mathscr{A}_\phi$ as cross product of all states of $\mathscr{K}_{\mathscr{N}_T,AP}$ with the acceptance set(s) for $T_{\mathscr{D},\phi}$. The only interesting part is again whether we can stay infinitely long in the a) case. Since all $\mathfrak{w}^x_{\mathfrak{s}}$ in the definition of $\mathscr{L}_{t,p}(\beta)$ are finite, we cannot stay in case a) forever and leave leaves in T_r due to the stutter transitions. Furthermore, we also cannot leaves leaves due to infinite sequences of transitions $t_x \in \mathscr{T}$ with $\mathfrak{s}_{\mathscr{K}}(s,t_x) = \emptyset$ because this can only happen when a branch does not switch into some stutter state $p'_{\mathfrak{s}}$ (in these states we have loops for all $t \in \mathscr{T}$). But this cannot happen because such sequences must have a finite chain, and thus the definition of $\mathscr{L}_{t,p}(\beta)$ enforces the switch into the stutter mode. □

5.4.2 Formal Construction of the Petri net $\mathscr{N}^>$ and the Correctness

In Sec. 5.3.2 a more descriptive definition for the construction of the Petri net with inhibitor arcs $\mathscr{N}^>$ from a Petri net with transits $\mathscr{N}_T$ and the n Büchi automata $A_{\neg\phi_i}$ for the reduction presented in Sec. 5.3 is given. Here we provide the completely formal definition of this construction. We fix a Petri net with transits $\mathscr{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$, a Flow-CTL* formula φ with $n \in \mathbb{N}$ flow subformulas $\varphi_{\mathbb{A}i} = \mathbb{A}\,\phi_i$ with atomic propositions AP_i, and the Büchi automata $A_{\neg\phi_i} = (\mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}, Q_i, I_i, E_i, F_i)$.

▶ **Definition 31 (Petri Net with Transits to Petri Net).** Given $\mathscr{N}_T$ and the corresponding n Büchi automata $A_{\neg\phi_i}$. We define the Petri net with inhibitor arcs $\mathscr{N}^> = (\mathscr{P}^>, \mathscr{T}^>, \mathscr{F}^>, \mathscr{F}_I^>, In^>)$ with

$$\mathscr{P}^> = \mathscr{P}_o^> \cup \bigcup_{i\in\{1,\dots,n\}} \mathscr{P}_i^>, \qquad \mathscr{T}^> = \mathscr{T}_o^> \cup \bigcup_{i\in\{1,\dots,n\}} \mathscr{T}_i^>,$$
$$\mathscr{F}^> = \mathscr{F}_O^> \cup \mathscr{F}_C^> \cup \bigcup_{i\in\{1,\dots,n\}} \mathscr{F}_i^>$$

and a partial labeling function $\lambda : \mathscr{T}^> \to \mathscr{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathscr{P}\}$ by:

(o) The original part of the net is a copy of $\mathscr{N}_T$ without the transits and with an additional activation place $\overrightarrow{a_o}$. Furthermore, we use the places $\boldsymbol{\mathcal{N}}$ and $\boldsymbol{\mathcal{S}}$, the

switch $t_{\mathcal{N}\to\mathcal{S}}$, and the stuttering transition $t_{\mathfrak{s}}$ for checking finite runs, in which case we have to trigger the automata $A_{\neg\phi_i}$ infinitely often to handle the stuttering:

$$\begin{aligned}
\mathscr{P}_O^{>} =& \mathscr{P} \cup \{\overrightarrow{a_o}, \boldsymbol{\mathcal{N}}, \boldsymbol{\mathcal{S}}\},\\
\mathscr{T}_O^{>} =& \mathscr{T} \cup \{t_{\mathcal{N}\to\mathcal{S}}, t_{\mathfrak{s}}\},\\
\mathscr{F}_O^{>} =& \mathscr{F} \cup \{(\overrightarrow{a_o}, t) \mid t \in \mathscr{T}\} \cup \{(\boldsymbol{\mathcal{N}}, t_{\mathcal{N}\to\mathcal{S}}), (t_{\mathcal{N}\to\mathcal{S}}, \boldsymbol{\mathcal{S}}), (\overrightarrow{a_o}, t_{\mathfrak{s}})\},\\
\mathscr{F}_I^{>} =& \{(\boldsymbol{\mathcal{S}}, t) \mid t \in \mathscr{T}\} \cup \{(\boldsymbol{\mathcal{N}}, t_{\mathfrak{s}})\}
\end{aligned}$$

(sub) For each Büchi automaton $A_{\neg\phi_i}$, we create the places, transitions, and flows to simulate the automaton.

The *places* are the states of the automaton with a special place $[\iota]_i$ for initially guessing the violating tree, one activation place $[\overrightarrow{a}]_i$ for activating this subnet, an activation place $[\overrightarrow{t}]_i$ for each transition $t \in \mathscr{T}$ for allowing only the firing of transitions labeled with t, and one place $[\overrightarrow{\mathfrak{s}}]_i$ for the global stuttering transitions:

$$\mathscr{P}_i^{>} = \{[s]_i \mid s \in Q_i\} \cup \{[\iota]_i, [\overrightarrow{a}]_i, [\overrightarrow{\mathfrak{s}}]_i\} \cup \{[\overrightarrow{t}]_i \mid t \in \mathscr{T}\}.$$

The *transitions* consists of

$$\mathscr{T}_i^{>} = \mathscr{T}_{\triangleright_i} \cup \mathscr{T}_{\Rrightarrow_i} \cup \mathscr{T}_{E_i}$$

with one transition for each initial flow chain: $\mathscr{T}_{\triangleright_i} = \{[t_p]_i \mid t \in \mathscr{T} \wedge (\triangleright, p) \in \Upsilon(t)\}$, one skipping transition $[t_{\Rrightarrow}]_i$ for each transition $t \in \mathscr{T}$ to skip checking the corresponding flow tree or for skipping transition when the tracking has not yet started: $\mathscr{T}_{\Rrightarrow_i} = \{[t_{\Rrightarrow}]_i \mid t \in \mathscr{T}\}$, and one transition for each edge of $A_{\neg\phi_i}$: $\mathscr{T}_{E_i} = \{[e]_i \mid e \in E_i\}$.

The *labeling* function λ labels every transition $t^{>} \in \mathscr{T}^{>}$ corresponding to a transition $t \in \mathscr{T}$ or the stuttering symbol, respectively: $\forall [t_p]_i \in \mathscr{T}_{\triangleright_i} : \lambda([t_p]_i) = t$, $\forall [t_{\Rrightarrow}]_i \in \mathscr{T}_{\Rrightarrow_i} : \lambda([t_{\Rrightarrow}]_i) = t$, and $\forall [e]_i \in \mathscr{T}_{E_i} : \lambda([e]_i) = l$ with $e = (s, l, s') \in E_i$.

The *flows* connect each transition $t^{>} \in \mathscr{T}_i^{>}$ not corresponding to the global stuttering to the activation place $[\overrightarrow{a}]_i$ of this subnet, and all transitions corresponding to a transition $t \in \mathscr{T}$ with the corresponding activation place $[\overrightarrow{t}]_i$:

$$\begin{aligned}
\mathscr{F}_{\to i}^{>} =& \{([\overrightarrow{a}]_i, t^{>}) \mid t^{>} \in \mathscr{T}_i \wedge \lambda(t^{>}) \neq \mathfrak{s}\}\\
&\cup \{([\overrightarrow{t}]_i, t^{>}) \mid t^{>} \in \mathscr{T}_i \wedge t \wedge \lambda(t^{>}) = t\}.
\end{aligned}$$

The transitions for guessing a flow chain are connected to the corresponding initial state of $A_{\neg\phi_i}$:

$$\begin{aligned}
\mathscr{F}_{\triangleright_i}^{>} = \{([\iota]_i, [t_p]_i), ([t_p]_i, q_0) \mid & [t_p]_i \in \mathscr{T}_{\triangleright_i}^{>} \wedge q_0 = (Q_r, \emptyset) \in I_i \wedge\\
& ((p, \cdot) \in Q_r \vee ((t_p, p), \cdot) \in Q_r)\}.
\end{aligned}$$

To not enforce the subnet to always track the first occurring of a flow tree and to skip all transition before the tracking starts, we allow to just pass the activation

tokens forward to the next net (done in **(con)**), when the initial token is still available:

$$\mathcal{F}^{>}_{\Rrightarrow_i} = \{([\iota]_i, [t_{\Rrightarrow}]_i), ([t_{\Rrightarrow}]_i, [\iota]_i) \mid [t_{\Rrightarrow}]_i \in \mathcal{T}_{\Rrightarrow_i}\}.$$

The transitions corresponding to an edge of $A_{\neg\phi_i}$ move the tokens accordingly:

$$\mathcal{F}^{>}_{E_i} = \{([s]_i, [e]_i), ([e]_i, [s']_i) \mid [e]_i \in \mathcal{T}_{E_i} \wedge e = (s, \lambda([e]_i), s')\}.$$

For the *global stuttering* global stuttering transitions are dependent on the stuttering place $[\overrightarrow{\mathfrak{s}}]_i$. For the *local stuttering* we put the activation token back into the activation place of the subnet to allow for switching into the local stuttering mode for any chain concurrently whenever the net is activated:

$$\begin{aligned}\mathcal{F}^{>}_{\mathfrak{s}_i} =& \{([\overrightarrow{\mathfrak{s}}]_i, t^{>}) \mid t^{>} \in \mathcal{T}^{>}_{E_i} \wedge \lambda(t^{>}) = \mathfrak{s}\} \cup \\ & \{(t^{>}, [\overrightarrow{a}]_i) \mid t^{>} \in \mathcal{T}^{>}_{E_i} \wedge \exists p \in \mathcal{P} : \lambda(t^{>}) = \mathfrak{s}_p\}.\end{aligned}$$

The flows of the subnet are the union of the previous sets:

$$\mathcal{F}^{>}_i = \mathcal{F}^{>}_{\rightarrow i} \cup \mathcal{F}^{>}_{\triangleright_i} \cup \mathcal{F}^{>}_{\Rrightarrow_i} \cup \mathcal{F}^{>}_{E_i} \cup \mathcal{F}^{>}_{\mathfrak{s}_i}.$$

(con) The nets are connected in a sequential manner:

$$\begin{aligned}\mathcal{F}^{>}_C =& \{(t, [\overrightarrow{t}]_1), (t, [\overrightarrow{a}]_1) \mid t \in \mathcal{T}\} \cup \{(t_{\mathfrak{s}}, [\overrightarrow{\mathfrak{s}}]_1)\} \\ & \cup \bigcup_{i \in \{1,\dots,n-1\}} \{(t^{>}, [\overrightarrow{t}]_{i+1}), (t^{>}, [\overrightarrow{a}]_{i+1}) \mid t^{>} \in \mathcal{T}_i{}^{>} \wedge \lambda(t^{>}) = t \in \mathcal{T}\} \\ & \cup \bigcup_{i \in \{1,\dots,n-1\}} \{(t^{>}, [\overrightarrow{\mathfrak{s}}]_{i+1}) \mid t^{>} \in \mathcal{T}_i{}^{>} \wedge \lambda(t^{>}) = \mathfrak{s}\} \\ & \cup \{(t^{>}, \overrightarrow{a_o}) \mid t^{>} \in \mathcal{T}_n{}^{>} \wedge \lambda(t^{>}) \in \mathcal{T} \cup \{\mathfrak{s}\}\}.\end{aligned}$$

(in) The initial marking is the original initial marking with the activation place for the original part, the place to guess to global stuttering, and one place for each subnet to start guessing the flow tree:

$$In^{>} = In \cup \{\overrightarrow{a_o}, \boldsymbol{\mathcal{N}}\} \cup \{[\iota]_i \mid i \in \{1, \dots, n\}\}.$$ ◀

In the case that $n = 0$ flow subformulas exists, we just take the input Petri net with transits $\mathcal{N}_T$ and omit the transits. In this case we can also leave the input Flow-CTL* formula φ as it is because it is already an LTL formula.

The size of the constructed net is dominated by the nondeterministic Büchi automata $A_{\neg\phi_i} = (\mathcal{T} \cup \{\mathfrak{s}\} \cup \{\mathfrak{s}_p \mid p \in \mathcal{P}\}, Q_i, I_i, E_i, F_i)$ checking the CTL* subformulas. The corresponding lemma is given on page 67.

Proof (Lemma 6: Size of the Constructed Net): The number of *places* is $|\mathcal{P}^{>}_O| = |\mathcal{P}|+3$ and $|\mathcal{P}^{>}_i| = |Q_i| + 3 + |\mathcal{T}|$. Hence, $|\mathcal{P}^{>}| = |\mathcal{P}| + 3 + (3 + |\mathcal{T}|) \cdot n + \sum_{i=1}^{n} |Q_i|$.
For the number of *transitions*, we have $|\mathcal{T}^{>}_O| = |\mathcal{T}| + 2$ and the size of $\mathcal{T}_i{}^{>}$ is maximally

$|\mathscr{T}|\cdot|\mathscr{P}|+|\mathscr{T}|+|E_i|$ because $|\mathscr{T}_{\triangleright_i}|$ is maximally $|\mathscr{T}|\cdot|\mathscr{P}|$, $|\mathscr{T}_{\Rrightarrow_i}|=|\mathscr{T}|$, and $|\mathscr{T}_{E_i}|=|E_i|$. Hence, $|\mathscr{T}^{>}|$ is in $\mathcal{O}(|\mathscr{T}|+2+(|\mathscr{T}|+|\mathscr{T}|\cdot|\mathscr{P}|)\cdot n+\sum_{i=1}^{n}|E_i|)$. For a double-exponential number of states Q_i and edges E_i for specifications ϕ_i in CTL* and single-exponential for specifications in CTL (Lemma 4) the size of the constructed net is in the respective classes. □

We can finally show the correctness of the transformation (Lemma 8 on page 70), by showing its contraposition via a nested structural induction.

Proof (Lemma 8: Correctness of the Transformation): Showing the contraposition $\mathscr{N}_T \not\models \varphi$ iff $\mathscr{N}^{>} \not\models_{\mathtt{LTL}} \psi^{>}$ of Lemma 8 consists of four major steps:

1. mutually translating the counterexamples,
2. using a structural induction over φ using the LTL parts of the run formula and the flow subformulas as induction base,
3. showing the correctness of the LTL parts of the run formula via a structural induction over φ, and
4. showing the correctness of the flow subformulas via a structural induction over φ.

This structure and also the single steps are very similar to those in the proofs for Flow-LTL presented in Sec. 6.4.3. We decided to present the single steps more elaborately in Sec. 6.4.3 for Flow-LTL rather than here for Flow-CTL*, since for Flow-CTL* Step 4 is mainly already shown due to Lemma 5 proving the correctness of the Büchi automaton. However, for Flow-LTL, this step is analogously proven to Step 3, why it is beneficial to have the more elaborate version close by.

For Step 1 we consider both directions. First, we take a run $\beta=(\mathscr{N}_T^R,\rho)$ of $\mathscr{N}_T$ and a covering firing sequence $\zeta=M_0[t_0\rangle M_1[t_1\rangle\cdots$ such that $\beta,\sigma_R(\zeta)\not\models\varphi$ and create a sequence $\zeta^{>}=M_0^{>}[t_0^{>}\rangle M_1^{>}[t_1^{>}\rangle\cdots$ such that $\beta^{>},\sigma_R(\zeta^{>})\not\models\psi^{>}$ holds for the corresponding run $\beta^{>}=(\mathscr{N}^{>R},\rho^{>})$. (i) The marking $M_0^{>}$ corresponds to the initial marking of $\mathscr{N}^{>}$, i.e., $\rho^{>}(M_0^{>})=In^{>}$. (ii) As long as there is a transition t_j of ζ, we map those to the transition of $\zeta^{>}$ at the $j(n+1)$st position, i.e., $t^{>}_{j(n+1)}=t_j$ with $\rho^{>}(t_{j(n+1)})=\rho(t_j)$. For finite firing sequences we add at the end the switch into the stuttering mode with a transition $t^{>}$ with $\rho^{>}(t^{>})=t_{\mathcal{N}\to\mathcal{S}}$ and use for every $(n+1)$st further step a transition $t^{>}$ with $\rho^{>}(t^{>})=t_{\mathfrak{s}}$. (iii) For all transitions $t^{>}=t^{>}_{j(n+1)+i}$ for $i\in\{1,\dots,n\}$ in between such transitions, we add transitions according to the counterexample flow chain. In case $\beta,\sigma_R(\zeta)\models\mathbb{A}\,\phi_i$, we use the to $\rho(t_j)$ corresponding skipping transition, i.e., $\rho(t^{>})=[\rho(t_j)_{\Rrightarrow}]_i$. If $\beta,\sigma_R(\zeta)\not\models\mathbb{A}\,\phi_i$, there exists a flow chain $\xi^i=t^i_\iota,p^i_0,t^i_0,p^i_1,t^i_1,\dots$ such that $\beta,\sigma_F(\xi^i)\not\models_{\mathtt{CTL}^*}\phi_i$. Lemma 5 yields $\mathscr{L}(A_{\neg\phi_i})\cap\mathscr{L}_{t^i_\iota,p^i_0}(\beta)\neq\emptyset$. Thus, as long as the chain is not started, we again use the corresponding skipping transition. When the chain is started, we take the word $w\in\mathscr{L}(A_{\neg\phi_i})\cap\mathscr{L}_{t^i_\iota,p^i_0}(\beta)$ corresponding to ξ^i and iteratively map each $t^{>}$ according to this w. This already contains the switch into

the stuttering mode and stuttering for finite chains. (iv) Each marking $M_k^>$ of $\zeta^>$ corresponds to the firing of the previous transition. With $\Theta : \mathbb{N} \to \mathbb{N}$ we map each step of $\sigma_R(\zeta)$ to the firing of the corresponding transition in the original part of the net in $\sigma_R(\zeta^>)$. Especially, $\Theta(0) = 0$ and $\Theta(1) = 1$, and if ζ is finite, then the stutter steps are mapped to the firings of transitions $t^>$ with $\rho^>(t^>) = t_{\mathfrak{s}}$. Intuitively, this function jumps in $n+1$ steps modulo the switching into the global stuttering with $t_{\mathcal{N}\to\mathcal{S}}$, and the switching into the local stuttering modes with some $\mathfrak{s}_p$. This construction corresponds to Definition 40.

For the other direction we create $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ and ρ along Definition 41. We first delete the switches into the local and global stuttering modes and the transitions t with $\rho^>(t) = t_{\mathfrak{s}}$ or $\rho^>(t) = \mathfrak{s}$. (i) We take the 0st and then each $n+1$st step and project onto the elements of $\mathcal{N}_T$, i.e., $M_j = \{p \in M^>_{j\cdot(n+1)} \mid \rho^>(p) \in \mathscr{P}\}$ and $t_j = t^>_{j\cdot(n+1)}$ for all $j \in \mathbb{N}$ (as long as $M^>_{j\cdot(n+1)}$ and $t^>_{j\cdot(n+1)}$ are existent in the reduced firing sequence of $\zeta^>$). (ii) The net $\mathcal{N}_T^R$ is iteratively created from ζ and ρ is also the projection of $\rho^>$. (iii) The flow chain $\xi^i = t^i_\iota, p^i_0, t^i_0, p^i_1, t^i_1, \ldots$ for a subnet $i \in \{1, \ldots, n\}$ is only created when there is some transition $t^>_j$ in $\zeta^>$ which takes a token of $[\iota]_i$, i.e., $\rho^>(t_j) \in \mathscr{T}_{\triangleright_i}$. In this case, we iteratively collect the transitions and their, to the transit belonging, places of the pre- and postset according to the transitions taken of the subnet corresponding to edges (edges labeled with $\mathfrak{s}$ or $\mathfrak{s}_p$ for any $p \in \mathscr{P}$ are already deleted). This is either done infinitely often or as long as there is a switch into the local stuttering mode in $\zeta^>$ for this subnet. With $\Theta : \mathbb{N} \to \mathbb{N}$ we again map each step of $\sigma_R(\zeta)$ to the firing of the corresponding transition in the original part of the net in $\sigma_R(\zeta^>)$.

For Step 2 we can use Lemma 19 for the soundness and Lemma 20 for the completeness as blue prints and only have to adapt the used lemmas corresponding to Step 3 and Step 4, because the outer structure of Flow-LTL and Flow-CTL* is the same. The structural induction is completely straightforward because the formula construction in Sec. 5.3.3 does not concern the operators in the run part of the formula. For the soundness we only have to show that the constructed firing sequence $\zeta^>$ satisfies $\sigma_R(\zeta^>) \models_{\text{LTL}} (\Box\Diamond \overrightarrow{a_o}) \wedge \mathtt{stutt}_s \wedge \mathtt{stutt}_c$ with $\mathtt{stutt}_s = \Box \bigwedge_{i\in\{1,\ldots,n\}, p\in\mathscr{P}} ([\mathfrak{s}_p]_i \to \Box\neg \bigvee_{t\in post^\Upsilon(p)} [t]_i)$ for $post^\Upsilon(p) = \{t \in post(p) \mid \exists q \in \mathscr{P} : (p,q) \in \Upsilon(t)\}$ and $\mathtt{stutt}_c = \Box \bigwedge_{i\in\{1,\ldots,n\}, p\in\mathscr{P}, q_p\in Q_p} (([q_p]_i \wedge \Box\neg \bigvee_{t\in post^\Upsilon(p)} [t]_i) \to \Diamond [\mathfrak{s}_p]_i)$ for $Q_p = \{(Q_r, \cdot) \in Q_i \mid (p, \cdot) \in Q_r \vee ((t, p), \cdot) \in Q_r\}$. The first conjunct is satisfied, because (ii) of the construction of $\zeta^>$ uses for every $(n+1)$st transition a transition of $\mathcal{N}_O^>$ (modulo the switch into the stuttering mode), i.e., a transition moving the activation token from $\overrightarrow{a_o}$ and Definition 31 ensures that the transition of the last subnet puts it back. The second conjunct, i.e., $\mathtt{stutt}_s$ is satisfied because $\zeta^>$ is constructed in (iii) along the $w \in \mathscr{L}(A_{\neg\phi_i}) \cap \mathscr{L}_{t,p}(\beta)$ corresponding to ξ^i. Thus, only when ξ^i is finite the switch into the stuttering mode is used. The third conjunct, i.e., $\mathtt{stutt}_c$ is satisfied because (iii) of the construction of $\zeta^>$ chooses the transitions of the subnet according to a word $w \in \mathscr{L}_{t,p}(\beta)$ (cp. page 62) and this ensures that when there is a finite flow chain, eventually the switch into the stuttering mode has to fire.

For Step 3 we show for the firing sequences ζ of $\mathcal{N}_T$ and $\zeta^>$ of $\mathcal{N}^>$, either given or constructed from the respective other in Step 1, with the function Θ mapping the corresponding indices, that

$$\forall i \in \mathbb{N} : \sigma_R(\zeta)^i \not\models_{\text{LTL}} \psi \iff \sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \psi^>$$

holds via a structural induction over a given LTL formula ψ. The induction anchor is easy, because due to Definition 20 and the construction in Step 1 the mapping with Θ ensures the satisfaction. Since the construction of the formula in Sec. 5.3.3 leaves the formula unchanged for the conjunction operator, this case directly follows from the induction hypothesis. In the case where ψ is the negation of an LTL formula, again the induction hypothesis directly provides the correctness due to the equivalence of the statement and that the construction of the formula in Sec. 5.3.3 has no impact on this case. The interesting cases are those where the construction modifies the formula, i.e., the next and the until operator.

Case $\psi = \bigcirc \psi_1$**.** The construction in Sec. 5.3.3 yields $\psi^> = (\mathtt{i} \rightarrow \bigcirc \psi_1^>) \wedge (\neg \mathtt{i} \rightarrow \bigcirc(\bigvee_{t \in \mathcal{T}^> \setminus \mathcal{T}^>_{O^-}} t \,\mathcal{U}\, \bigvee_{t' \in \mathcal{T}^>_{O^-}} t' \wedge \psi_1^>))$ with $\mathcal{T}^>_{O^-} = \mathcal{T}^>_O \setminus \{t_{\mathcal{N} \rightarrow \mathcal{S}}\}$.
For the *soundness* the premise yields $\sigma_R(\zeta)^{i+1} \not\models_{\text{LTL}} \psi_1$ and the induction hypothesis $(\star)$ $\sigma_R(\zeta^>)^{\Theta(i+1)} \not\models_{\text{LTL}} \psi_1^>$. If $i = 0$, the subformula $\mathtt{i}$ is satisfied because no transition let into the initial state. Thus, the second conjunct of $\psi^>$ is satisfied as well as the premise for the first conjunct. Since $\Theta(0) = 0$ and $\Theta(1) = \Theta(i+1) = 1$, we have that $\sigma_R(\zeta^>)^{\Theta(i+1)} \not\models_{\text{LTL}} \psi_1^>$ implies $\sigma_R(\zeta^>)^{\Theta(i)+1} \not\models_{\text{LTL}} \psi_1^>$ and therewith $\sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \bigcirc \psi_1^>$. Hence, all together $\sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \bigcirc \psi^>$. We show the case $i > 0$ by contradiction. Assume $\sigma_R(\zeta^>)^{\Theta(i)} \models_{\text{LTL}} \bigcirc \psi^>$. Then, $\sigma_R(\zeta^>)^{\Theta(i)} \models_{\text{LTL}} \bigcirc(\bigvee_{t \in \mathcal{T}^> \setminus \mathcal{T}^>_{O^-}} t \,\mathcal{U}\, \bigvee_{t' \in \mathcal{T}^>_{O^-}} t' \wedge \psi_1^>)$ because $\mathtt{i}$ is not satisfied. In $\Theta(i)$ we are in a state where a transition of the original part of the net has fired. Hence, in the next state only transitions of the subnets (or the global switch into the stuttering mode) can fire due to the construction of $\mathcal{N}^>$ in Sec. 5.3.2. Those, transitions are skipped with the until operator, as long as another transition of the original part of the net fires. This is exactly at state $\Theta(i+1)$. Thus, $\sigma_R(\zeta^>)^{\Theta(i+1)} \models_{\text{LTL}} \psi_1^>$ holds which is a *contradiction* to $(\star)$.
For the *completeness* the premise is $\sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \psi^>$. When the subformula $\mathtt{i}$ is satisfied the premise yields $\sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \bigcirc \psi_1^>$. The subformula $\mathtt{i}$ can only be satisfied when $i = 0$ and since $\Theta(i) + 1 = 0 + 1 = 1 = \Theta(1) = \Theta(i+1)$ we have $\sigma_R(\zeta^>)^{\Theta(i+1)} \not\models_{\text{LTL}} \psi_1^>$. The induction hypothesis yields $\sigma_R(\zeta)^{(i+1)} \not\models_{\text{LTL}} \psi_1$ and so $\sigma_R(\zeta)^i \not\models_{\text{LTL}} \bigcirc \psi_1$. In the case that the subformula $\mathtt{i}$ is not satisfied, the premise yields $\sigma_R(\zeta^>)^{\Theta(i)} \not\models_{\text{LTL}} \bigcirc(\bigvee_{t \in \mathcal{T}^> \setminus \mathcal{T}^>_{O^-}} t \,\mathcal{U}\, \bigvee_{t' \in \mathcal{T}^>_{O^-}} t' \wedge \psi_1^>)$. We obtain $\sigma_R(\zeta^>)^{\Theta(i+1)} \not\models_{\text{LTL}} \psi_1^>$ with the same arguments as for the soundness case. Again, the induction hypothesis yields the result.

Case $\psi = \psi_1 \,\mathcal{U}\, \psi_2$**.** The construction in Sec. 5.3.3 yields $\psi^> = ((\mathtt{M} \vee \mathtt{i}) \rightarrow \psi_1^>) \,\mathcal{U}\, ((\mathtt{M} \vee \mathtt{i}) \wedge \psi_2^>)$.
For the *soundness* the premise yields $\forall k \geq 0 : \sigma_R(\zeta)^{i+k} \not\models_{\text{LTL}} \psi_2 \vee \exists 0 \leq l < k : \sigma_R(\zeta)^{i+l} \not\models_{\text{LTL}} \psi_1$. The induction hypothesis applied for all these ks and ls yields:

$(\star)$ $\forall k \geq 0 : \sigma_R(\zeta^>)^{\Theta(i+k)} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \sigma_R(\zeta^>)^{\Theta(i+l)} \not\models_{\text{LTL}} \psi_1^>$. We show the result by contradiction. Assume $\exists k \geq 0 : \sigma_R(\zeta^>)^{\Theta(i)+k} \models_{\text{LTL}} ((\mathtt{M} \vee \mathtt{i}) \wedge \psi_2^>) \wedge \forall 0 \leq l < k : \sigma_R(\zeta^>)^{\Theta(i)+l} \models_{\text{LTL}} (\mathtt{M} \vee \mathtt{i}) \rightarrow \psi_1^>$. Since $\sigma_R(\zeta^>)^{\Theta(i)+k} \models_{\text{LTL}} \mathtt{M} \vee \mathtt{i}$ we know that either a transition of the original part of the net has fired, or it is the initial state. Anyhow, this step belongs to $\mathscr{N}_O^>$ and therefore there is a $k' \geq 0$ such that $\Theta(i+k') = \Theta(i) + k$. Furthermore, $\sigma_R(\zeta^>)^{\Theta(i)+k} \models_{\text{LTL}} \psi_2^>$ holds and $\forall 0 \leq l < k : \sigma_R(\zeta^>)^{\Theta(i)+l} \models_{\text{LTL}} (\mathtt{M} \vee \mathtt{i}) \rightarrow \psi_1^>$. The latter ensures that all of these states which belong to $\mathscr{N}_O^>$ satisfy $\psi_1^>$ which is a *contradiction* to $(\star)$.
For the *completeness* the premise yields $\forall k \geq 0 : \sigma_R(\zeta^>)^{\Theta(i)+k} \not\models_{\text{LTL}} ((\mathtt{M} \vee \mathtt{i}) \wedge \psi_2^>) \vee \exists 0 \leq l < k : \sigma_R(\zeta^>)^{\Theta(i)+l} \not\models_{\text{LTL}} (\mathtt{M} \vee \mathtt{i}) \rightarrow \psi_1^>$. For each step $\Theta(i)+k$ or $\Theta(i)+l$ which belongs to the original part of the net, i.e., there is a k' or l' such that $\Theta(i+k') = \Theta(i)+k$ or $\Theta(i+l') = \Theta(i)+l$, we know $\mathtt{M} \vee \mathtt{i}$ is satisfied. Thus, $\forall k \geq 0 : \sigma_R(\zeta^>)^{\Theta(i+k)} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \sigma_R(\zeta^>)^{\Theta(i+l)} \not\models_{\text{LTL}} \psi_1^>$. The induction hypothesis for all ks and ls yields $\forall k \geq 0 : \sigma_R(\zeta)^{i+k} \not\models_{\text{LTL}} \psi_2 \vee \exists 0 \leq l < k : \sigma_R(\zeta)^{i+l} \not\models_{\text{LTL}} \psi_1$. Hence, $\sigma_R(\zeta)^i \not\models_{\text{LTL}} \psi$.

In Step 4 we show that for a flow formula $\mathbb{A}\,\phi_i$ of φ and the corresponding part $\phi_i^> = \neg\Box\Diamond\overline{\bigvee_{b \in F_i}[b]_i}$ of $\psi^>$ constructed in Sec. 5.3.3 the following holds:

(s) Given a run β, a covering firing sequence ζ, and the constructed firing sequence $\zeta^>$ with $\sigma_R(\zeta^>) \models_{\text{LTL}} (\Box\Diamond\overrightarrow{a_o}) \wedge \mathtt{stutt}_s \wedge \mathtt{stutt}_c$, then $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\phi_i \implies \sigma_R(\zeta^>) \not\models_{\text{LTL}} \neg\Box\Diamond\bigvee_{b \in F_i}[b]_i$ holds.

(c) Given a firing sequence $\zeta^>$ of a run of the Petri net $\mathscr{N}^>$ with $\sigma_R(\zeta^>) \models_{\text{LTL}} (\Box\Diamond\overrightarrow{a_o}) \wedge \mathtt{stutt}_s \wedge \mathtt{stutt}_c$, the constructed firing sequence ζ and corresponding run $\beta = (\mathscr{N}_T^R, \rho)$, then $\sigma_R(\zeta^>) \not\models_{\text{LTL}} \neg\Box\Diamond\bigvee_{b \in F_i}[b]_i \implies \beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\phi_i$ holds.

Regarding (s): Due to $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\phi_i$ we know there is a flow chain $\xi^i = t, p, \ldots$ such that $\beta, \sigma_F(\xi^i) \not\models_{\text{CTL}^*} \phi_i$. Lemma 5 yields $\mathscr{L}(A_{\neg\phi_i}) \cap \mathscr{L}_{t,p}(\beta) \neq \emptyset$. The construction of $\zeta^>$ ensures in (iii) that $\zeta^>$ skips the subnet as long as the chain is created and then starts tracking the chain along the word $w \in \mathscr{L}(A_{\neg\phi_i}) \cap \mathscr{L}_{t,p}(\beta)$. Since $w \in \mathscr{L}(A_{\neg\phi_i})$, we know that $\sigma_R(\zeta^>)$ must visit a place $[b]_i$ corresponding to a Büchi state of $A_{\neg\phi_i}$ infinitely often. Due to the finally operator we do not have to do anything special for skipping the subnets or the initial skipping of $\zeta^>$ and directly know $\sigma_R(\zeta^>) \not\models_{\text{LTL}} \neg\Box\Diamond\bigvee_{b \in F_i}[b]_i$.

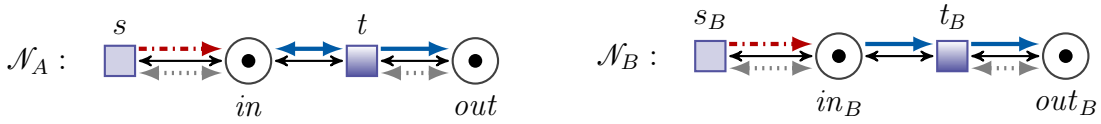
Regarding (c): Since $\sigma_R(\zeta^>) \models_{\text{LTL}} \Box\Diamond\bigvee_{b \in F_i}[b]_i$ there must be a transition $t \in \mathscr{T}_{\triangleright_i}$ taking the token from $[\iota]_i$ due to Definition 31 and $\iota \notin F_i$. Thus, there is a transition $t \in \mathscr{T}^R$ and a place $p \in \mathscr{P}^R$ such that $(\triangleright, p) \in \Upsilon^R(t)$ because of the construction of ζ. Definition 31 ensures that $\zeta^>$ corresponds to the run of the automaton after starting a chain. This translates to ζ because Definition 31 ensures that the subnets mimic the fired transitions of the original part and the construction of ζ takes exactly these transitions. Thus, $\mathscr{L}(A_{\neg\phi_i}) \cap \mathscr{L}_{t,p}(\beta) \neq \emptyset$ because $\sigma_R(\zeta^>)$ visits a Büchi state infinitely often. Lemma 5 yields that there is a flow chain ξ^i, such that $\beta, \sigma_F(\xi^i) \not\models_{\text{CTL}^*} \phi_i$. Hence, $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\phi_i$. □

6 Model Checking Petri Nets with Transits against Flow-LTL

The previous chapter provides a triple-exponential time algorithm for model checking distributed systems with local data flows modeled with Petri net with transits for specifications in Flow-CTL* and a double-exponential time algorithm for specifications in Flow-CTL. These algorithms are based on transforming the data flow part, i.e., the CTL* parts (or the CTL parts, respectively) of the Flow-CTL* formula into a standard Petri net via a sequence of automata constructions. These parts are connected in series such that the model checking problem is reduce to a model checking problem for standard Petri nets and LTL. This chapter provides dedicated algorithms for the fragment of Flow-LTL. Instead of using the sequence of automata constructions with the exponential blow-ups for the data flow part, we directly construct a subnet for checking the LTL formulas for the local data flow. By that, we obtain a single-exponential time algorithm for model checking distributed system with local data flows modeled with Petri nets with transits and specifications in Flow-LTL. In order to facilitate access for readers, we keep this chapter as self-contained as possible, but without redundant formalisms. Even though the understanding of the previous chapter, especially the intuition behind the different timelines, is certainly not a hindrance for digesting this chapter, formally only Definition 21 on page 47 and the fairness and maximality assumptions presented in Example 1 on page 47 are needed to process this chapter.

As a running example for this chapter we consider the two small Petri nets with transits depicted in Fig. 6.1. Both differ only in one transit. For a larger example see the example in Sec. 2.1 for the application domain of software-defined networking.

▶ **Example 6.** We introduce two Petri nets with transits $\mathcal{N}_A$ and $\mathcal{N}_B$ as small running examples in Fig. 6.1. Both nets handle infinitely many data flows, which are created by firing transition s or s_B, respectively. The difference is that the net $\mathcal{N}_A$ keeps all data flows both in places in and out when firing transition t. By contrast, $\mathcal{N}_B$ transfers the data flow to place out_B, such that immediately after firing t_B, no data flow resides in place in_B. We demand the transitions t and t_B to be weakly fair, i.e., if the respective transition is eventually enabled forever, it has to fire infinitely often (cp. Example 1 on page 47). Therefore, each network has to eventually transit the data flow via transition t or t_B, respectively. As a result, in network $\mathcal{N}_A$, not every data flow reaches place out, even if we assume that infinitely many transitions are fired. There is always a flow staying in place in. In $\mathcal{N}_B$, all data flows eventually reach place out_B if we assume that infinitely many transitions are fired, as t_B is weakly fair and eventually transits the data flow to out_B. ◀



(a) *The Petri net with transits $\mathcal{N}_A$ transits all in place 'in' created data flows via transition t to the places 'in' and 'out'. All flows are kept alive in places 'in'* and *'out'.*

(b) *The net $\mathcal{N}_B$, similar to $\mathcal{N}_A$ apart from moving the data flows only from in_B to out_B via t_B (no left blue solid arrow tip). Thereby, only out_B has all data flows.*

Fig. 6.1: *Two small running examples for Petri nets with transits. Both can create infinitely many data flows and can transfer them to their output node.*

This chapter is structured as follows: We first formally introduce Flow-LTL in Sec. 6.1, before providing two different decision procedures for model checking Petri nets with transits against Flow-LTL in Sec. 6.2. Similarly to the construction in the previous chapter, we create for each data flow part of the Flow-LTL formula a subnet and compose them adequately to reduce the model checking problem of Petri nets with transits against Flow-LTL to the model checking problem of standard Petri nets against LTL. The first construction composes the subnets again in a sequential manner such that we obtain a single-exponential time algorithm (cp. Sec. 6.2.1). The second construction, presented in Sec. 6.2.2, composes the subnets in a parallel manner resulting in a double-exponential time algorithm for specifications with more than one data flow subformula. However, the second exponent is only dependent on the number of flow subformulas n and for specifications which does not need to reason about the data flow of *different* selections of runs, this algorithm still runs in single-exponential time and in general outperforms the sequential approach for the examples of software-defined networking (cp. Chap. 7). In Sec. 6.3 we further reduce the model checking problem for safe Petri nets against LTL with places and transitions as atomic propositions to a hardware model checking problem by encoding the Petri net in a circuit. Section 6.4 collects formal details and proofs for the previous sections to improve the reading experience of the before mentioned sections.

This chapter is based on [FGHO19a] and [FGHO20a], and the corresponding full versions [FGHO19b] and [FGHO20b].

6.1 Flow-LTL

For Petri nets with transits, we wish to express requirements on several separate timelines. Based on the global timeline of the system run, global conditions like fairness and maximality can be expressed. Requirements on individual data flows, e.g., that the data flow does not enter a loop, are expressed in terms of the timeline of that specific data flow. *Flow-LTL* comprises of *run formulas* φ specifying the usual LTL behavior on markings and *data flow formulas* $\varphi_{\mathbb{A}}$ specifying properties of flow chains inside runs.

For example for the fairness assumptions we want to reason about the actual firing of transition. Hence, we define Flow-LTL with $AP = \mathcal{P} \cup \mathcal{T}$ as atomic propositions. Even though Flow-LTL is a syntactic fragment of Flow-CTL* we explicitly define the

semantics of Flow-LTL in this section to simplify the notation. Likewise, we define here traces only on flow chains and not on the flow chain suffixes because without the branching there is no need to find a suffix of the chain. Similarly, we can directly use the places and transitions of the Petri net with transits $\mathcal{N}_T$ for the traces of the flow chains and do not have to use the nodes of the run, because while evaluating the flow formula it suffices to shorten the traces to respective suffixes. Lastly, we consider in this chapter the outgoing semantics for the firing sequences and flow chains. This means that we bundle the current marking (or place) together with the transition leaving the marking (or the place). All this serves for a clearer presentation and easier notation in the case of Flow-LTL.

For a covering firing sequence of a run of a Petri net with transits we define a trace, such that a state of the system is the current marking together with the transition used to *leave* the marking. Furthermore, we *stutter* in the last marking for finite firing sequences to obtain an infinite semantics.

▶ **Definition 32 (Outgoing Semantics – Firing Sequences).** Given a run $\beta = (\mathcal{N}^R, \rho)$ of a Petri net or Petri net with transits $\mathcal{N}$. To a (finite or infinite) *covering firing sequence* $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ of β, we associate a *trace* $\sigma_R(\zeta) : \mathbb{N} \to 2^{AP}$ with

$$\sigma_R(\zeta)(i) = \rho(M_i) \cup \{\rho(t_i)\} \text{ for all } i \in \mathbb{N}$$

if ζ is infinite and

$$\sigma_R(\zeta)(i) = \begin{cases} \rho(M_i) \cup \{\rho(t_i)\} & \text{for } 0 \leq i < n \\ \rho(M_n) & \text{otherwise} \end{cases}$$

if $\zeta = M_0[t_0\rangle \cdots [t_{n-1}\rangle M_n$ is finite. ◀

Hence, a trace of a covering firing sequence is an infinite sequence of states collecting the current marking and outgoing transition of $\mathcal{N}$, which stutters on the last marking for finite sequences.

Similarly, for a flow chain of a run of a Petri net with transits we consider the current place p of the chain together with the transition used to transit the data flow from p to the next place. We also stutter on the last place for finite chains.

▶ **Definition 33 (Outgoing Semantics – Flow Chain).** Given a run $\beta = (\mathcal{N}_T^R, \rho)$ of a Petri net with transits $\mathcal{N}_T$. To a (finite or infinite) *flow chain* $\xi = t_0, p_0, t_1, p_1, t_2, \ldots$ of β we associate a trace $\sigma_F(\xi) : \mathbb{N} \to 2^{AP}$ with

$$\sigma_F(\xi)(i) = \{\rho(p_i), \rho(t_{i+1})\} \text{ for all } i \in \mathbb{N}$$

if ξ is infinite and

$$\sigma_F(\xi)(i) = \begin{cases} \{\rho(p_i), \rho(t_{i+1})\} & \text{for } 0 \leq i < n \\ \{\rho(p_n)\} \text{ for} & \text{otherwise} \end{cases}$$

if $\xi = t_0, p_0, t_1, p_1, \ldots, t_n, p_n$ is finite. ◀

Hence, a trace of a flow chain is an infinite sequence of states collecting the current place and outgoing transition of the flow chain, which stutter on the last place for finite flow chains.

The definition of the semantics of LTL on the flow chains fits to the definition of Flow-CTL* (Definition 23) on flow chain suffixes.

▶ **Definition 34 (LTL on Flow Chains).** Given a trace $\sigma_F(\xi)$ of a flow chain ξ of a run $\beta = (\mathcal{N}_T^R, \rho)$. The *semantics* of LTL on flow chains is defined as follows:

$$
\begin{array}{lll}
\sigma_F(\xi) \models_{\text{LTL}} a & \text{iff} & a \in \sigma_F(\xi)(0) \\
\sigma_F(\xi) \models_{\text{LTL}} \neg\psi & \text{iff} & \text{not } \sigma_F(\xi) \models_{\text{LTL}} \psi \\
\sigma_F(\xi) \models_{\text{LTL}} \psi_1 \wedge \psi_2 & \text{iff} & \sigma_F(\xi) \models_{\text{LTL}} \psi_1 \text{ and } \sigma_F(\xi) \models_{\text{LTL}} \psi_2 \\
\sigma_F(\xi) \models_{\text{LTL}} \bigcirc\psi & \text{iff} & \sigma_F(\xi)^1 \models_{\text{LTL}} \psi \\
\sigma_F(\xi) \models_{\text{LTL}} \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \text{there exists a } j \geq 0 \text{ with } \sigma_F(\xi)^j \models_{\text{LTL}} \psi_2 \text{ and} \\
 & & \text{for all } 0 \leq i < j \text{ the following holds: } \sigma_F(\xi)^i \models_{\text{LTL}} \psi_1
\end{array}
$$

with atomic propositions $a \in AP$ and LTL formulas ψ, ψ_1, and ψ_2. ◀

Note that since a trace of a flow chain maps to the set of places and transitions of the Petri net with transits $\mathcal{N}_T$ this definition also fits the definition of the semantics for LTL on runs given in Definition 21 on page 47.

Using the definition of LTL on runs (Definition 21) and on flow chains (Definition 34), we define the specification language Flow-LTL for Petri nets with transits.

▶ **Definition 35 (Flow-LTL).** We define the *syntax* of Flow-LTL with:

$$\varphi ::= \psi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \psi \rightarrow \varphi \mid \mathbb{A}\,\psi$$

where φ, φ_1, φ_2 are Flow-LTL formulas, ψ is an LTL formula. Hence, LTL formulas $\psi \in \text{LTL}$ may appear both inside the run part and the flow part of φ. We call $\varphi_{\mathbb{A}} = \mathbb{A}\,\psi$ *flow formulas* and all other subformulas which are not under the $\mathbb{A}$ operator *run formulas*.

The *semantics* of a Petri net with transits $\mathcal{N}_T = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ satisfying a Flow-LTL formula φ is defined over the covering firing sequences of its runs:

$$
\begin{array}{rll}
\mathcal{N}_T \models \varphi & \text{iff} & \text{for all runs } \beta \text{ of } \mathcal{N}_T : \beta \models \varphi \\
\beta \models \varphi & \text{iff} & \text{for all firing sequences } \zeta \text{ covering } \beta : \ \beta, \sigma_R(\zeta) \models \varphi \\
\beta, \sigma_R(\zeta) \models \psi & \text{iff} & \sigma_R(\zeta) \models_{\text{LTL}} \psi \\
\beta, \sigma_R(\zeta) \models \varphi_1 \wedge \varphi_2 & \text{iff} & \beta, \sigma_R(\zeta) \models \varphi_1 \text{ and } \beta, \sigma_R(\zeta) \models \varphi_2 \\
\beta, \sigma_R(\zeta) \models \varphi_1 \vee \varphi_2 & \text{iff} & \beta, \sigma_R(\zeta) \models \varphi_1 \text{ or } \beta, \sigma_R(\zeta) \models \varphi_2 \\
\beta, \sigma_R(\zeta) \models \psi \rightarrow \varphi & \text{iff} & \beta, \sigma_R(\zeta) \models \psi \text{ implies } \beta, \sigma_R(\zeta) \models \varphi \\
\beta, \sigma_R(\zeta) \models \mathbb{A}\,\psi & \text{iff} & \text{for all flow chains } \xi \text{ of } \beta : \sigma_F(\xi) \models_{\text{LTL}} \psi
\end{array}
$$

with LTL formulas ψ and Flow-LTL formulas φ, φ_1 and φ_2. We say a Petri net with transits $\mathcal{N}_T$ *satisfies* a Flow-LTL formula φ iff $\mathcal{N}_T \models \varphi$ holds. ◀

With this definition we can now formally state and check the in Example 6 intuitively described property that all data packets reach the egress node.

▶ **Example 7.** For the nets $\mathcal{N}_A$ or $\mathcal{N}_B$ of Fig. 6.1, we specify that every data flow eventually reaches the respective egress node *out* as follows (omitting the subscript B for all elements of $\mathcal{N}_B$): $\varphi_1 = (\texttt{max} \wedge \texttt{fair}) \rightarrow \mathbb{A}\Diamond\, out$ with the interleaving-maximality $\texttt{max} = \Box((in \vee (in \wedge out)) \rightarrow (s \vee t)) \equiv s \vee t$ and $\texttt{fair} = \Diamond\Box(in \wedge out) \rightarrow \Box\Diamond t \equiv \Box\Diamond t$ as the weak fairness assumption of t. Note that we adapted the definition of the interleaving-maximality to the outgoing semantics, i.e., we do not need the next operator in the definition. All other properties defined in Example 1 on page 47, e.g., the weak fairness property, do not differ for the ingoing and outgoing semantics due to the eventually and globally operators. This formula is not satisfied in $\mathcal{N}_A$, because the firing sequence $\zeta = M_0[s'\rangle M_1[t_0\rangle M_2[t_1\rangle \cdots$, with $\rho(M_j) = \{in, out\}$, and $\rho(s') = s$, $\rho(t_j) = t$ for $j \in \mathbb{N}$, yields a run β of $\mathcal{N}_A$ which also has the flow trace $\sigma_F(\xi) = \{in, t\}, \{in, t\}, \ldots$. The run β is maximal, since all covering firing sequences (this is only ζ) are infinite. The run is also fair for transition t, since t occurs infinitely often in $\sigma_R(\zeta)$. Since the flow trace $\sigma_F(\xi)$ never reaches the egress place *out*, the net $\mathcal{N}_A$ does not satisfy φ_1.

For $\mathcal{N}_B$, every firing sequence covering a run of $\mathcal{N}_B$ has to infinitely often fire a transition t' with $\rho(t') = t$ because of the maximality and fairness assumptions. Hence, either no flow chain exists, because no transition s' with $\rho(s') = s$ has ever fired, or every chain must eventually be transited to the place *out*. Thus, $\mathcal{N}_B$ satisfies φ_1. ◀

To check such properties automatically we present two algorithms in the next section.

6.2 Reduction to Model Checking Petri Nets against LTL

Central to our model checking routine for a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ, as already for Petri net with transits against Flow-CTL* specifications (cp. Chap. 5), is the reduction to a safe standard Petri net $\mathcal{N}^>$ and an LTL formula $\varphi^>$. The infinite state space of the Petri net with transits due to possibly infinitely many flow chains is reduced to a finite state model. The key idea is to guess and track a violating flow chain for each flow subformula $\mathbb{A}\,\psi_i$, for $i \in \{1, \ldots, n\}$, and to only once check the equivalent future of flow chains merging into a common place. As in the previous chapter the construction composes subnets: one copy of the original net $\mathcal{N}_T$ without the transits and one subnet for each of the n flow subformula for guessing the violating chain and checking its correctness. The constructed formula has to ensure that concurrent transitions not concerning the local timeline are skipped for the n subnets and that the composition of the subnets does not make any harm while checking the run part of the formula on the original part of the net.

In this section we present two approaches for this reduction, mainly differing in the composition technique of the subnets. Figure 6.2 and Fig. 6.3 give an overview of the *sequential* approach and the *parallel* approach, respectively. Both algorithms create one subnet $\mathcal{N}_i^>$ for each flow subformula $\mathbb{A}\,\psi_i$ to track the corresponding flow chain and have one subnet $\mathcal{N}_O^>$ to check the run part of the formula. The places of $\mathcal{N}_O^>$ are copies

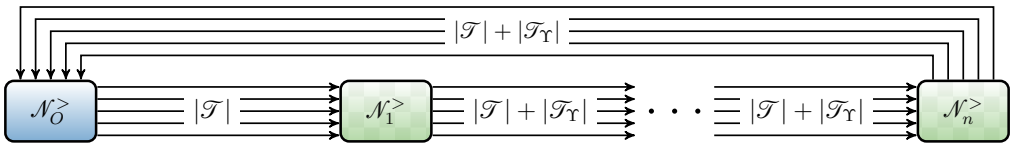


Fig. 6.2: *Overview of the* sequential approach*: Each firing of a transition of the original net is split into first firing a transition in the subnet for the run formula and subsequently firing a transition in each subnet tracking a flow formula. The constructed LTL formula skips the additional steps with until operators. Since we track only one chain, we have one transition for each transit in $\mathcal{N}_T$, i.e., $|\mathcal{T}_\Upsilon| = \sum_{t \in \mathcal{T}} |\Upsilon(t)|$.*

of the places in $\mathcal{N}_T$ such that the current state of the system can be memorized. The subnets $\mathcal{N}_i^>$ also consist of the original places of $\mathcal{N}_T$ but only use one token (initially residing on an additional place) to track the current state of the considered flow chain. The approaches differ in how these nets are connected to obtain $\mathcal{N}^>$. For a clear distinction between the individual approaches, we often use the subscript » for the sequential approach and $\|$ for the parallel approach.

Sequential Approach: The places in each subnet $\mathcal{N}_i^>$ are connected with one transition for each transit. An additional token iterates sequentially through the subnets to activate or deactivate the subnet. This allows each subnet to track a flow chain corresponding to the firing of a transition in $\mathcal{N}_O^>$. The formula $\varphi_»^>$ takes care of these additional steps by means of the until operator: In the run part of the formula, all steps corresponding to moves in a subnet $\mathcal{N}_i^>$ are skipped and, for each subformula $\mathbb{A}\,\psi_i$, all steps are skipped until the next transition of the corresponding subnet is fired which transits the tracked flow chain. This technique results in a polynomial increase of the size of the Petri net and the formula: $\mathcal{N}_»^>$ has $\mathcal{O}(|\mathcal{N}_T| \cdot n + |\mathcal{N}_T|)$ places and $\mathcal{O}(|\mathcal{N}_T|^3 \cdot n + |\mathcal{N}_T|)$ transitions and the size of $\varphi_»^>$ is in $\mathcal{O}(|\mathcal{N}_T|^3 \cdot n \cdot |\varphi| + |\varphi|)$. This approach is presented in Sec. 6.2.1

Parallel Approach: The n subnets are connected such that the current chain of each subnet is tracked simultaneously while firing an original transition $t \in \mathcal{T}$. Thus, there are $(|\Upsilon(t)| + 1)^n$ transitions for each transition $t \in \mathcal{T}$. Each of these transitions stands for exactly one combination of which subnet is tracking which (or no) transit. Hence, firing one transition of the original net is directly tracked in one step for all subnets. This significantly reduces the complexity of the run part of the constructed formula, since no until operator is needed to skip sequential steps. A disjunction over all transitions corresponding to an original transition suffices to ensure the correctness of the construction. Transitions and next operators in the flow parts of the formula still have to be replaced by means of the until operator to ensure that the next step of the tracked flow chain is checked at the corresponding step of the global timeline of $\varphi_\|^>$. In general, the parallel approach results in an exponential blow-up of the net and the formula: $\mathcal{N}_\|^>$ has $\mathcal{O}(|\mathcal{N}_T| \cdot n + |\mathcal{N}_T|)$ places and $\mathcal{O}(|\mathcal{N}_T|^{3n} + |\mathcal{N}_T|)$ transitions and the size of $\varphi_\|^>$ is in $\mathcal{O}(|\mathcal{N}_T|^{3n} \cdot |\varphi| + |\varphi|)$. For the practical examples, however, the parallel approach allows for model checking Flow-LTL with few flow subformulas

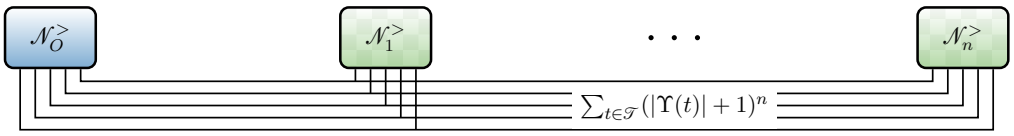


Fig. 6.3: *Overview of the* parallel approach*: The n subnets are connected such that for every transition $t \in \mathcal{T}$ there are $(|\Upsilon(t)|+1)^n$ transitions, i.e., there is one transition for every combination of which transit of t (or none) is tracked by which subnet. We use until operators in the constructed LTL formula to only skip steps not involving the tracking of the guessed chain in the flow formula.*

with a significant speed-up in comparison to the sequential approach (cp. Chap. 7). This approach is presented in Sec. 6.2.2.

To keep this section lighter, we outsource formal details and proofs for both subsections into Sec. 6.4.

6.2.1 A Sequential Approach

We solve the model checking problem of a Flow-LTL formula φ on a Petri net with transits $\mathcal{N}_T$ in three steps:

1. The Petri net with transits $\mathcal{N}_T$ is encoded as a standard Petri net $\mathcal{N}_»^>$ without transits obtained by composing suitably modified copies of $\mathcal{N}_T$ such that each flow subformula in φ can be checked for correctness using the corresponding copy. The constructed net $\mathcal{N}_»^>$ is of polynomial size in $|\mathcal{N}_T|$ and the number of flow subformulas n.

2. The Flow-LTL formula φ is transformed to an LTL formula $\varphi_»^>$ which skips the uninvolved composition copies when evaluating run and flow parts, respectively. The constructed formula $\varphi_»^>$ is of polynomial size in $|\mathcal{N}_T|$ and $|\varphi|$.

3. $\mathcal{N}_»^> \models_{\text{LTL}} \varphi_»^>$ is checked to answer $\mathcal{N}_T \models \varphi$.

Given a safe Petri net with transits $\mathcal{N}_T = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ and a Flow-LTL formula φ with flow subformulas $\varphi_{\mathbb{A}_i} = \mathbb{A}\,\psi_i$, where $i = 1, \ldots, n$ for some $n \in \mathbb{N}$, we construct a safe Petri net $\mathcal{N}_»^> = (\mathcal{P}^>, \mathcal{T}^>, \mathcal{F}^>, \mathcal{F}_I^>, In^>)$ with inhibitor arcs and an LTL formula $\varphi_»^>$. Details for the following constructions, as well as all proofs corresponding to the reduction can be found in Sec. 6.4.

From Petri Nets with Transits to P/T Petri nets

In this section we give a more intuitive, but still complete, definition of the construction of $\mathcal{N}_»^>$ from a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ with n flow subformulas for a lightweight understanding. The formal construction is given in Definition 38 on page 105. Figure 6.4 visualizes the process by an example.

We divide the construction of $\mathcal{N}_{\gg}^{>}$ into the construction of the *original part* of the net, the construction of the *subnets*, and the part that belong to properly *connecting* these elements:

Original Part: The *original part* of $\mathcal{N}_{\gg}^{>}$ is denoted by $\mathcal{N}_O^{>}$ with places $\mathscr{P}_O^{>}$ and transitions $\mathscr{T}_O^{>}$. This part is a copy of the original Petri net with transits $\mathcal{N}_T$ without the transit relation. Furthermore, it contains an initially marked activation place $\overrightarrow{o}$ in the preset of each transition to only allow for the firing of transitions of this subnet when it is the turn of the original part. This subnet is used to check the run part of the formula φ.

Subnets: For each subformula $\mathbb{A}\,\psi_i$ of φ, a *subnet* (denoted by $\mathcal{N}_i^{>}$, with places $\mathscr{P}_i^{>}$ and transitions $\mathscr{T}_i^{>}$) is composed to $\mathcal{N}_O^{>}$. The subnet introduces the possibility to decide for the tracking of up to one specific flow chain by introducing a copy $[p]_i$ of each place $p \in \mathscr{P}$ and transitions simulating the transits. This means a place of the subnet stands for the current state of the flow chain and we have one transition for each possibility to move the token to a successor place of a flow chain. For example, when a transition $t \in \mathscr{T}$ splits the data flow in a place $p \in \mathscr{P}$ into two successor places ($|post^{\Upsilon}(p,t)| = 2$), $\mathscr{T}_i^{>}$ contains two different transitions corresponding to the two different extensions of the data flow chain. The additional place $[\iota]_i$ serves for starting the tracking. This place is initially marked and the subnet can nondeterministically choose whether to start the tracking of a newly created flow chain or not. For that, skipping transitions $t_{\Rightarrow_i}$ for each $t \in \mathscr{T}$ are introduced. These can be fired if and only if no place $[p]_i$ for a place $p \in pre(t)$ is occupied, meaning that the corresponding chain is not tracked in this run. These transitions are also used in cases where concurrent transitions are fired not involving the currently tracked flow chain. Each transition $t^{>} \in \mathscr{T}_i^{>}$ corresponds to a transition $t \in \mathscr{T}$ and is labeled with $\lambda(t^{>}) = t$ accordingly. Each run of such a subnet simulates one possible flow chain of $\mathcal{N}_T$, i.e., every firing sequence covering any run of $\mathcal{N}_T$ yields a flow chain. This subnet serves for checking the corresponding data flow part of the formula φ.

Connecting: An *activation token* iterates sequentially through these components via places $\overrightarrow{t}$ for $t \in \mathscr{T}$. In each step, the active component has to fire exactly one transition and pass the active token to the next component. The sequence starts by $\mathcal{N}_O^{>}$ firing a transition t and proceeds through every subnet simulating the data flows according to the transits of t. This implies that the subnets have to either move their data flow via a t-labeled non-skipping transition $t^{>} \in \mathscr{T}_i^{>}$ with $\lambda(t^{>}) = t$ or use the skipping transition $t_{\Rightarrow_i}$ if their chain is not involved in the firing of t or a newly created chain should not be considered in this run.

Note that different to the construction in Sec. 5.3 for model checking Flow-CTL* formulas, we do not need to take special care of the stuttering for finite flow chains or finite firing sequences. This problem is solved in the semantics for model checking the standard Petri net $\mathcal{N}_{\gg}^{>}$ against $\varphi_{\gg}^{>}$. A finite flow chain already stays at the corresponding place in the subnet and we do not have to first switch to the corresponding state of the automaton.

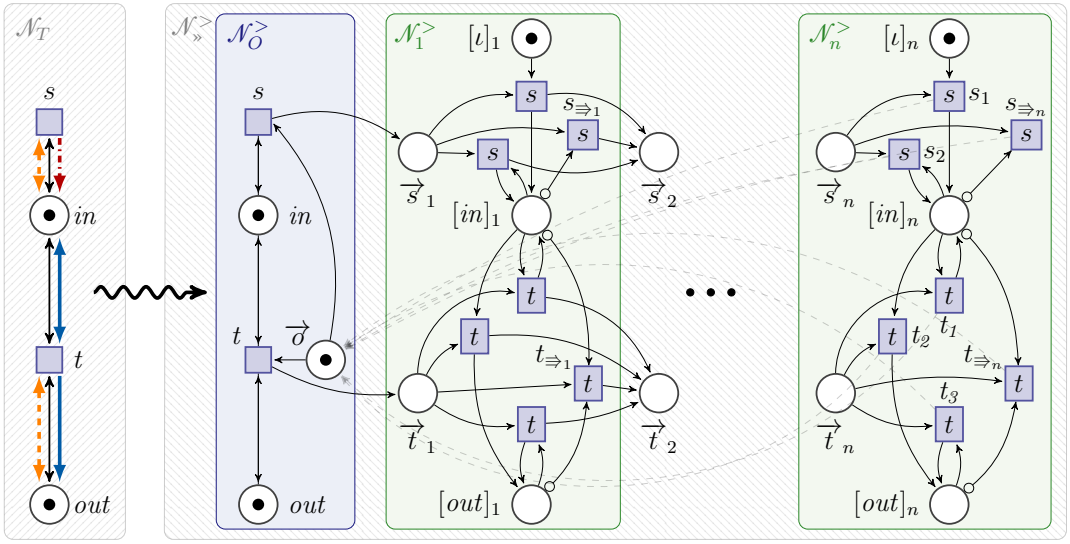


Fig. 6.4: *An overview of the constructed P/T Petri net $\mathcal{N}_\gg^>$ (on the right) for an example Petri net with transits $\mathcal{N}_T$ (on the left, here $\mathcal{N}_A$ of Fig. 6.1a) and n flow subformulas $\mathbb{A}\,\psi_i$. A run of $\mathcal{N}_\gg^>$ repeats the following. First, the original part $\mathcal{N}_O^>$ does a step by firing any enabled transition $t' \in \mathcal{T}_O$ (here t or s). Then the activation token moves sequentially through the subnets such that every subnet mimics the firing of t' (either by extending the data flow through t' by one of the t'-labeled non-skipping transitions or by skipping t' with $s_{\Rightarrow_i}$ or $t_{\Rightarrow_i}$). The inhibitor arcs ensure only valid choices of skipping transitions. Finally, the last subnet hands back the token to the original part (depicted as gray dashed arrows to increase the readability).*

▶ **Lemma 10 (Size of the Constructed Net).** *The constructed Petri net $\mathcal{N}_\gg^>$ has $\mathcal{O}(|\mathcal{N}_T| \cdot n + |\mathcal{N}_T|)$ places and $\mathcal{O}(|\mathcal{N}_T|^3 \cdot n + |\mathcal{N}_T|)$ transitions.* ◀

Proof Sketch. The results directly follow from the construction of $\mathcal{N}_\gg^>$. Having a copy of each original place for each flow subformula and an activation place for each original transition yields the quadratic number of places in the size of the net and the number of subformulas. The cubic number of the transitions in the size of the net and the number of subformulas follows from the possible quadratic number of transits of each transition and that for each flow subformula and for each transit a new transition is added for the subnet. For more details see the proof on page 107. □

The Petri net with transits $\mathcal{N}_T$ depicted in Fig. 6.4 on the left is exactly the Petri net with transits $\mathcal{N}_A$ of the running example depicted in Fig. 6.1a. Thus, we continue our running example.

▶ **Example 8.** The Petri net $\mathcal{N}_A^>$ created from the Petri net with transits $\mathcal{N}_A$ of Fig. 6.1a and the Flow-LTL formula $\varphi_1 = (\texttt{max} \wedge \texttt{fair}) \rightarrow \mathbb{A} \Diamond \mathit{out}$ of Example 7 is depicted in Fig. 6.4 by combining $\mathcal{N}_O^>$ with the rightmost subnet $\mathcal{N}_n^>$. Since φ_1 only contains one flow subformula, so $n = 1$ and $\mathcal{N}_A^>$ consists of the original part and only one subnet for tracking the local data flow. We can see that transition t is split into the transitions t_1,

t_2, and t_3 due to its three transits. One transition for keeping the data flow in place *out* and two transitions for splitting the data flow coming from place *in* into two successor places. There is only one transition in the postset of $[\iota]_n$ because in $\mathcal{N}_A$ only s can create new data flows and this only in place *in*.

The constructed Petri net for the Petri net with transits $\mathcal{N}_B$ depicted in Fig. 6.1b only differs in missing the looping transition t_1 due to not splitting up the data flow with transition t_B. ◀

From Flow-LTL Formulas to LTL Formulas and the Correctness

In this section we define the construction of an LTL formula $\varphi_{\gg}^{>}$ from a Flow-LTL formula φ and the Petri net $\mathcal{N}_{\gg}^{>}$ constructed of a Petri net with transits $\mathcal{N}_T$ in the previous section. The main part of this construction is to properly handle the points in time an atomic proposition should hold for a firing sequence in $\mathcal{N}_T$ and the corresponding atomic proposition should hold for a firing sequence in $\mathcal{N}_{\gg}^{>}$. For these points in time there are two shift. First, we have the shift for atomic propositions for flow subformulas due to the difference between the global and the local timelines (cp. Fig. 5.1). This means shifts originating from situations in $\mathcal{N}_T$ where concurrent transitions are fired that do not affect the considered flow chain. Second, due to the sequential composition of the subnets the points in time are additionally shifted for the subnets but also for the run part of the formula. This means we want to skip the situations which stem from the sequential processing of the other subnets in $\mathcal{N}_{\gg}^{>}$.

These two different kinds of shifting are encoded in the LTL formula $\varphi_{\gg}^{>}$. On the one hand, the data flow formulas $\mathbb{A}\,\psi_i$ in φ are now checked on the corresponding subnets $\mathcal{N}_i^{>}$ and, on the other hand, the run formula part of φ is checked on the original part of the net $\mathcal{N}_O^{>}$. In both cases, we need to ignore places and transitions from other parts of the composition. This is achieved by replacing each next operator $\bigcirc\,\phi$ and atomic proposition $t \in \mathcal{T}$ inside φ with an until operator. Transitions which are not representing the considered timeline are called *unrelated*, others *related*. Via the until operator, all unrelated transitions can fire until a related transition is fired. This is formalized in Tab. 6.1 using the sets $O = \mathcal{T}^{>} \setminus \mathcal{T}$ and $O_i = (\mathcal{T}^{>} \setminus \mathcal{T}_i^{>}) \cup \{t_{\Rightarrow_i} \in \mathcal{T}_i^{>} \mid t \in \mathcal{T}\}$, for the *unrelated* transitions of the original part and of the subnets, respectively. The *related* transitions of the original part are given by $\mathcal{T}$ and for the subnets by $M_i(t) = \{t' \in \mathcal{T}_i^{>} \setminus \{t_{\Rightarrow_i}\} \mid \lambda(t') = t\}$ and $M_i = \mathcal{T}_i^{>} \setminus \{t_{\Rightarrow_i} \in \mathcal{T}_i^{>} \mid t \in \mathcal{T}\}$. Removing the skipping transitions from the transitions of the subnet for the related transitions and adding them to the unrelated transitions ensures the correct handling of the global and the local timelines. Since the subnet $\mathcal{N}_O^{>}$ is just a copy of the original net, we can keep the original atomic propositions $t \in \mathcal{T}$ for the run part of the formula for the related transitions. Since we add one transition to the subnet $\mathcal{N}_i^{>}$ for each data flow a transition $t \in \mathcal{T}$ transits, we have to use all of these transitions as atomic propositions for the flow part of the formula for the related transitions. Note that the second disjunctive in the replacement for the next operator is needed in case of finite firing sequences or flow chains. When no related transition ever fires again, we stutter on the last marking and the last place of the finite flow chain. Thus, we can directly check the formula in

Tab. 6.1: *Row 1 considers the substitutions in the run part of φ, row 2 the substitutions in each subformula φ_{F_i}. Column 1 considers simultaneously substitutions, column 2 substitutions from the inner- to the outermost occurrence.*

$t \in \mathcal{T}$	$\bigcirc\, \phi$
$(\bigvee_{t' \in O} t')\, \mathcal{U}\, t$	$((\bigvee_{t \in O} t)\, \mathcal{U}((\bigvee_{t' \in \mathcal{T}} t') \wedge \bigcirc\, \phi)) \vee (\Box(\neg(\bigvee_{t' \in \mathcal{T}} t')) \wedge \phi)$
$(\bigvee_{t_o \in O_i} t_o)\, \mathcal{U}(\bigvee_{t_m \in M_i(t)} t_m)$	$((\bigvee_{t \in O_i} t)\, \mathcal{U}((\bigvee_{t \in M_i} t) \wedge \bigcirc\, \phi)) \vee (\Box(\neg(\bigvee_{t \in M_i} t)) \wedge \phi)$

this state.

This construction ensures that exactly n transitions in a trace of a firing sequence of $\mathcal{N}_{\gg}^{>}$ are skipped for the transitions of the run part, i.e., one transition for each subnet, due to the construction of $\mathcal{N}_{\gg}^{>}$. For the flow subformula part, at least n states of the trace are skipped. When the original net fires concurrent transitions which do not involve the considered flow chain, with the skipping transition we take another round and even more states are skipped.

Additionally, every atomic proposition $p \in \mathcal{P}$ in the scope of a flow operator is simultaneously substituted with its corresponding place $[p]_i$ of the subnet. Every flow subformula $\mathbb{A}\, \psi_i$ is substituted with $\Box[\iota]_i \vee ([\iota]_i\, \mathcal{U}(\neg[\iota]_i \wedge \psi_i'))$ (abbreviated $[\iota]_i\, \mathcal{W}(\neg[\iota]_i \wedge \psi_i')$), where $\Box[\iota]_i$ represents that no flow chain is tracked and ψ_i' is the result of the substitutions of atomic propositions and next operators described before. With $[\iota]_i\, \mathcal{U}(\neg[\iota]_i \wedge \psi_i')$ we ensure to only check the flow subformula at the time the chain is created. Finally, restricting runs to not end in any of the subnets yields the final formula $\varphi^{>} = (\Box\Diamond\, \overrightarrow{o}) \rightarrow \varphi^{\mathbb{A}}$ with $\overrightarrow{o}$ being the activation place of the original part of the net and $\varphi^{\mathbb{A}}$ the result of the substitution of all flow subformulas. This construction is given as formal substitutions in Definition 39 on page 107.

This construction yields an LTL formula of polynomial size in the size of the input net $\mathcal{N}_T$ and formula φ.

▶ **Lemma 11 (Size of the Constructed Formula).** *The size of the constructed LTL formula $\varphi_{\gg}^{>}$ is in $\mathcal{O}(|\mathcal{N}_T|^3 \cdot n \cdot |\varphi| + |\varphi|)$.* ◀

Proof Sketch. The size of the constructed formula is a direct result of the construction and Lemma 10. The substitutions with the largest impact are the ones substituting the transitions and the next operators. Both have disjunctions over the transitions of the constructed net $\mathcal{N}_{\gg}^{>}$. For more details see the proof on page 110. □

As an example we transform the formula φ_1 of Example 7.

▶ **Example 9.** The above construction for the Flow-LTL formula $\varphi_1 = (\texttt{max} \wedge \texttt{fair}) \rightarrow \mathbb{A}\Diamond\, out$ of Example 7 with the formulas $\texttt{max} = \Box((in \vee (in \wedge out)) \rightarrow (s \vee t))$ and $\texttt{fair} = \Diamond\Box(in \wedge out) \rightarrow \Box\Diamond t$, and the constructed Petri net $\mathcal{N}_A^{>}$ of Example 8 yields the formula

$$\varphi_1^{>} = (\Box\Diamond\, \overrightarrow{o}) \rightarrow ((\texttt{max}^{>} \wedge \texttt{fair}^{>}) \rightarrow (\Box[\iota]_1 \vee ([\iota]_1\, \mathcal{U}(\neg[\iota]_1 \wedge \Diamond[out]_1))))$$

for $\texttt{max}^{>} = \square((in \vee (in \wedge out)) \rightarrow (\texttt{s} \vee \texttt{t}))$ with $\texttt{s} = (s_1 \vee s_2 \vee s_{\Rrightarrow_1} \vee t_1 \vee t_2 \vee t_3 \vee t_{\Rrightarrow_1})\,\mathcal{U}\, s$ and $\texttt{t} = (s_1 \vee s_2 \vee s_{\Rrightarrow_1} \vee t_1 \vee t_2 \vee t_3 \vee t_{\Rrightarrow_1})\,\mathcal{U}\, t$, and $\texttt{fair}^{>} = \diamondsuit\square(in \wedge out) \rightarrow \square\diamondsuit((s_1 \vee s_2 \vee s_{\Rrightarrow_1} \vee t_1 \vee t_2 \vee t_3 \vee t_{\Rrightarrow_1})\,\mathcal{U}\, t)$. Note that we do not use the reduced, equivalent formulas presented in Example 7 to show more substitutions that make the transformation clearer. ◀

A transformation of the counterexamples for $\mathcal{N}_T$ satisfying φ and $\mathcal{N}_{\Rrightarrow}^{>}$ satisfying $\varphi_{\Rrightarrow}^{>}$ can serve for showing the *correctness* of the construction. We break the correctness into two parts. First, we show that the construction is *sound*, i.e., $\mathcal{N}_{\Rrightarrow}^{>} \models_{\texttt{LTL}} \varphi_{\Rrightarrow}^{>}$ implies $\mathcal{N}_T \models \varphi$. This means, checking the transformed elements yields correct results. Second, we show the *completeness* of the construction, i.e., $\mathcal{N}_T \models \varphi$ implies $\mathcal{N}_{\Rrightarrow}^{>} \models_{\texttt{LTL}} \varphi_{\Rrightarrow}^{>}$. This means that all results that can be obtained by checking the input elements, can also be obtained by checking the transformed elements.

▶ **Lemma 12 (Correctness of the Transformation).** *For a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ, the constructions of the safe P/T Petri net $\mathcal{N}_{\Rrightarrow}^{>}$ with inhibitor arcs and the LTL formula $\varphi_{\Rrightarrow}^{>}$ yield $\mathcal{N}_T \models \varphi$ iff $\mathcal{N}_{\Rrightarrow}^{>} \models_{\texttt{LTL}} \varphi_{\Rrightarrow}^{>}$.* ◀

Proof Sketch. For the soundness and completeness of the construction we can translate the counterexamples from one to another.

Soundness. We pump up the counterexample for $\mathcal{N}_T$ satisfying φ, i.e., a firing sequence ζ, by the transitions of the subnets. This is done in the way that for every transition t in ζ, i.e., transitions of the original part, the skipping transitions are added for a subnet, when the violating chain of the flow subformula corresponding to this subnet is not affect by the firing of t. When there is no such violating chain, i.e., the subformula is satisfied by $\mathcal{N}_T$, we always use the skipping transitions. If the chain serving as counterexample for the satisfaction of the flow subformula is affected by the firing of t, the corresponding transition of the chain is added. This created firing sequence serves as counterexample, since the satisfying flow subformulas are still satisfied by the $\square[\iota]_i$ part of $\varphi_{\Rrightarrow}^{>}$ and all added states of the trace are properly skipped by the untils of the constructed formula.

Completeness. The firing sequence ζ serving as counterexample for $\mathcal{N}_T$ satisfying φ is gained from the firing sequence ζ' serving as counterexample for $\mathcal{N}_{\Rrightarrow}^{>}$ satisfying $\varphi_{\Rrightarrow}^{>}$ by projecting on the elements of $\mathcal{N}_T$. Since the net is created in the way that only runs are allowed which start with a firing of a transition of the original part, and afterwards each subnet has to mimic the firing of this transition one after another, and the inhibitor arcs ensure that a violation in the subnet parts are violations of maximal data flow chains, we can created the necessary flow chains serving as counterexample for the flow subformulas by following the tracked flow chain of the subnet. For more details see the proof on page 119. □

We show this translation of the counterexamples by the firing sequence given in Example 7 showing that our running example $\mathcal{N}_A$ does not satisfy the formula φ_1.

▶ **Example 10.** In Example 7 it is argued that the example Flow-LTL formula φ_1 is not satisfied in $\mathcal{N}_A$ because the trace $\sigma_R(\zeta) = M_0 \cup \{s\}, M_0 \cup \{t\}, M_0 \cup \{t\}, \ldots$, with

$M_0 = \{in, out\}$, serves as a counterexample. We now argue that also $\varphi_1^>$ from Example 9, which is constructed from φ_1 and $\mathcal{N}_A^>$, is not satisfied in $\mathcal{N}_A^>$. We translate the firing sequence ζ to a firing sequence ζ' covering a run of $\mathcal{N}_A^>$ by using Definition 40. This serves as counterexample for $\mathcal{N}_A^>$ satisfying $\varphi_1^>$. The trace $\sigma_R(\zeta') = In^> \cup \{s\}, M_1 \cup \{s_1\}, M_2 \cup \{t\}, M_3 \cup \{t_1\}, M_4 \cup \{t\}, M_3 \cup \{t_1\}, M_4 \cup \{t\} \cdots$ with

$$
\begin{aligned}
In^> &= \{in, out, \overrightarrow{o}, [\iota]_1\} \\
M_1 &= \{in, out, \overrightarrow{s}_1, [\iota]_1\} \\
M_2 &= \{in, out, \overrightarrow{o}, [in]_1\} \\
M_3 &= \{in, out, \overrightarrow{t}_1, [in]_1\} \\
M_4 &= \{in, out, \overrightarrow{o}, [in]_1\}
\end{aligned}
$$

is created by using the transitions from ζ whenever it is the turn of the original part and when it is the turn of the subnet, we choose the transitions of the flow trace $\sigma_F(\xi) = \{in, t\}, \{in, t\}, \ldots$ which served as counterexample in Example 7 for the flow subformula. Since $\beta^R, \sigma_R(\zeta) \not\models \mathbb{A} \Diamond out$, we decided to take s_1 rather than $s_{\Rrightarrow_1}$. The firing sequence ζ' is actually a counterexample for $\mathcal{N}_A^>$ satisfying $\varphi_1^>$ because $\Box\Diamond \overrightarrow{o}$ is satisfied by marking M_4 which occurs infinitely often in $\sigma_R(\zeta')$. Furthermore, the constraint $\texttt{max}^>$ is satisfied because even though *in* and *out* are satisfied in every state of the trace $\sigma_R(\zeta')$ (satisfying the premise), there are at most only s_1 or t_1 transitions before transition s or t is used to leave the state (satisfying the conclusion). The fairness constraint $\texttt{fair}^>$ is also satisfied because transition t occurs infinitely often in $\sigma_R(\zeta')$. But $[\iota]_1 \notin M_3 \cup M_4$ and $[out]_1 \notin M_3 \cup M_4$. This yields $\mathcal{N}_A^> \not\models_{\text{LTL}} \varphi_1^>$. ◀

The sizes of the constructed Petri net and LTL formula yield the single-exponential time bound for the model checking algorithm for Flow-LTL and Petri nets with transits.

▶ **Theorem 4.** *A safe Petri net with transits $\mathcal{N}_T$ can be checked against a Flow-LTL formula φ in single-exponential time in the size of $\mathcal{N}_T$ and φ.* ◀

Proof. Lemma 10 and Lemma 11 yield the polynomial sizes of the constructed Petri net $\mathcal{N}_»^>$ and the LTL formula $\varphi_»^>$. Lemma 12 yields that we can check $\mathcal{N}_»^> \models_{\text{LTL}} \varphi_»^>$ instead of $\mathcal{N}_T \models \varphi$ and as for Theorem 3 the exponential blow-up due to the markings of the Petri net yield the final result. □

Note that the reachability problem for safe Petri nets is a special case of checking safe Petri nets with transits against Flow-LTL. Since the former problem is PSPACE-complete [CEP95], model checking a safe Petri net with transits against Flow-LTL has a PSPACE-hard lower bound.

6.2.2 A Parallel Approach

The parallel approach for model checking a Petri net with transits $\mathcal{N}_T$ against a Flow-LTL φ is similar to the sequential approach. We are again proceeding in three steps:

1. The Petri net with transits $\mathcal{N}_T$ is encoded as a standard Petri net $\mathcal{N}_{\|}^{>}$ without transits obtained by composing suitably modified copies of $\mathcal{N}_T$ such that each flow subformula in φ can be checked for correctness using the corresponding copy. The constructed net $\mathcal{N}_{\|}^{>}$ is of polynomial size in $|\mathcal{N}_T|$, but exponential in the number of flow subformulas n.

2. The Flow-LTL formula φ is transformed to an LTL formula $\varphi_{\|}^{>}$ which handles the local versus the global timeline of the run and the flow part of φ. The constructed formula $\varphi_{\|}^{>}$ is of polynomial size in $|\mathcal{N}_T|$ and $|\varphi|$, but of exponential size in the number of flow subformulas n.

3. $\mathcal{N}_{\|}^{>} \models_{\text{LTL}} \varphi_{\|}^{>}$ is checked to answer $\mathcal{N}_T \models \varphi$.

The main difference of these two approaches is how we connect the subnets tracking the data flow chains. In the sequential approach we use an activation token which, when started in the original part, subsequently activates each subnet tracking a flow chain, and informs the subnet about the fired transition. With this approach, these additional steps had to be skipped within the constructed formula $\varphi_{\gg}^{>}$. With the parallel approach, we spare this blow-up of the formula at the expense of an exponential number of transitions with respect to the number of flow subformulas.

From Petri Nets with Transits to P/T Petri nets

Step 1 of the reduction procedure constructs a standard Petri net $\mathcal{N}_{\|}^{>} = (\mathscr{P}^{>}, \mathscr{T}^{>}, \mathscr{F}^{>}, \mathscr{F}_I^{>}, In^{>})$ for a given Petri net with transits $\mathcal{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and a Flow-LTL formula φ with n flow subformulas. In this section we give a textual definition of this construction, the formal definition is given in Definition 44 on page 121. In Fig. 6.5 the application of this definition to the running example is depicted.

We divide the construction of $\mathcal{N}_{\|}^{>}$ into the construction of the *original part* of the net, the construction of the *subnets*, and the part that belong to properly *connecting* these elements:

Original Part: For the parallel approach the *original part* of $\mathcal{N}_{\|}^{>}$ is just a copy of the places of the input Petri net with transits $\mathcal{N}_T$. Transitions are only added globally for connecting all subnets. This part is denoted by $\mathcal{N}_O^{>}$. This subnet is still used to check the run part of the formula φ.

Subnets: As in the sequential case there is a *subnet* $\mathcal{N}_i^{>}$ for each subformula $\mathbb{A}\,\psi_i$ of φ. The *places* $\mathscr{P}_i^{>}$ of the subnet are again the copies $[p]_i$ of each place $p \in \mathscr{P}$ with the initially marked place $[\iota]_i$ for guessing the violating flow chain. For the parallel approach there is no need for the activation places. The *transitions* are no longer assigned to a specific subnet, but are connecting all subnets together. Each subnet still serves for checking the corresponding data flow part of the formula φ.

Connecting: For connecting the subnet $\mathcal{N}_O^{>}$ and the n subnets $\mathcal{N}_i^{>}$, we introduce $(|\Upsilon(t)| + 1)^n$ transitions $t^{>} \in \mathscr{T}^{>}$ for each transition $t \in \mathscr{T}$ which are all labeled

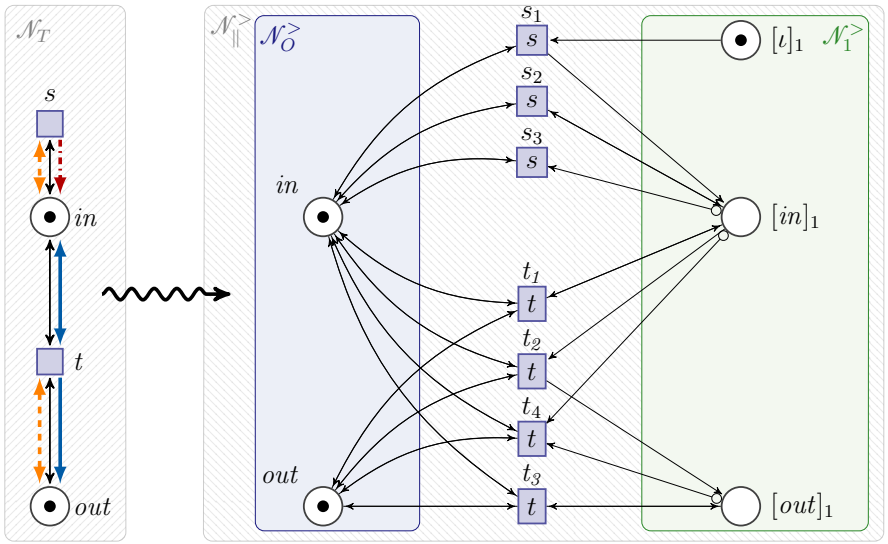


Fig. 6.5: *The constructed P/T Petri net $\mathcal{N}_\parallel^>$ (on the right) for an example Petri net with transits $\mathcal{N}_T$ (on the left, here $\mathcal{N}_A$ of Fig. 6.1a) and one flow subformula $\mathbb{A}\,\psi_1$. The subnets consists only of places, whereas the transitions connect all subnets simultaneously.*

with the original transition, i.e., $\lambda(t^>) = t$ and are connected to the original part $\mathcal{N}_O^>$ according to t. The transition $t^> \in \mathcal{T}^>$ is connected to the subnets dependent on whether this instance is used to track the data flow in this subnet or not. When the instance should track the chain it is connected to the places of the subnet according to the transit. Otherwise, it has inhibitor arcs to all places $[p]_i$ corresponding to place $p \in pre(t)$ to prevent this transition from firing when a chain extended by a transit of the transition is currently tracked.

The size of the constructed Petri net is dominated by the number of transits in the Petri net with transits $\mathcal{N}_T$ and the number n of flow subformulas in the Flow-LTL formula φ.

▶ **Lemma 13 (Size of the Constructed Net).** *The constructed Petri net $\mathcal{N}_\parallel^>$ has $O(|\mathcal{N}_T| \cdot n + |\mathcal{N}_T|)$ places and $O(|\mathcal{N}_T|^{3n} + |\mathcal{N}_T|)$ transitions.* ◀

Proof. The sizes can be directly read from the definition. For the number of transitions, we have $|\mathcal{P}| \cdot |\mathcal{P}| + |\mathcal{P}|$ possible transits for a transition $t \in \mathcal{T}$ in the worst-case. Thus, $|\Upsilon(t)|$ can at most be quadratic in the size of $\mathcal{N}_T$. Adding for each transition $t \in \mathcal{T}$ the $(|\Upsilon(t)| + 1)^n$ transitions yields the result. □

As an example, we use this definition to create the standard Petri net $\mathcal{N}_\parallel^>$ for our running example.

▶ **Example 11.** Given the Petri net with transits $\mathcal{N}_A$ of Fig. 6.1a and the Flow-LTL formula φ_1 of Example 7 with one flow subformula. The constructed standard Petri net $\mathcal{N}_\parallel^>$ is depicted in Fig. 6.5. In this case we can spare two transitions in comparision to the sequential case (the ones moving the tokens in the original part of the net).

This changes when the Flow-LTL formula under consideration has more than one flow subformula. Already for two flow subformulas, this example has nine s-labeled transitions and 16 t-labeled transitions. Consider for example the s-labeled transitions corresponding to transition $s \in \mathscr{T}$ for two flow subformulas. We define the set $X = \{(\circ,\circ),(\circ,\triangleright),(\circ,\leftrightarrow),(\triangleright,\circ),(\triangleright,\triangleright),(\triangleright,\leftrightarrow),(\leftrightarrow,\circ),(\leftrightarrow,\triangleright),(\leftrightarrow,\leftrightarrow)\}$ corresponding to the $(|\Upsilon(s)|+1)^2 = 9$ created transitions. Here, with the symbol $\circ$ we indicate that for this subnet the corresponding transition is not moving the data flow (has an inhibitor arc), with $\triangleright$ the new data flow is considered, and $\leftrightarrow$ denotes the case where the data flow is kept in place in. For each $(x_1,x_2) \in X$ the original place in is in the pre- and postset of the corresponding transition $t^{>}$. For $x_i = \triangleright$ we have $[\iota]_i$ in the preset and $[in]_i$ in the postset of $t^{>}$ for $i \in \{1,2\}$. Similarly for $x_i = \leftrightarrow$ we have $[in]_i$ in the pre- and the postset of $t^{>}$. For $x_i = \circ$ there is an inhibitor arc to the place $[in]_i$. Thus, we have a transition for each combination of which subnet tracks which transit of the transition s. The activation token and additional places can still be spared. ◀

From Flow-LTL Formulas to LTL Formulas and the Correctness

Step 2 of the reduction procedure creates an LTL formula $\varphi_{\|}^{>}$ to the Petri net $\mathscr{N}_{\|}^{>} = (\mathscr{P}^{>},\mathscr{T}^{>},\mathscr{F}^{>},\mathscr{F}_I^{>},In^{>})$ of a Petri net with transits $\mathscr{N}_T = (\mathscr{P},\mathscr{T},\mathscr{F},\Upsilon,In)$ and a Flow-LTL formula φ with $n \in \mathbb{N}$ flow subformulas $\varphi_{\mathbb{A}i} = \mathbb{A}\,\psi_i$. Again, the intricate part of the construction is to deal with the shifting of time points when the corresponding atomic propositions should hold. However, different to the sequential approach, for the parallel approach we do not have to additionally handle the extra shifts due to the sequential composition of the subnets. Thus, we only have to be adequately skip the global steps not concerning the flow chain for the flow subformulas (cp. Fig. 5.1). This is analogously done with until operators as for the sequential approach.

To have again a notion of related and unrelated transitions, we define the set of transitions tracking a chain of a specific subnet $i \in \{1,\ldots,n\}$ by $\mathscr{T}_i^{>} = \{t \in \mathscr{T}^{>} \mid \exists p \in \mathscr{P} : ([p]_i,t) \in \mathscr{F}^{>} \vee (t,[p]_i) \in \mathscr{F}^{>}\}$ (the *related* transitions of the subnet) and the set of all other transitions by $O_i = \mathscr{T}^{>} \setminus \mathscr{T}_i^{>}$ (the *unrelated* transitions of the subnet). For a transition $t \in \mathscr{T}$, the set $M_i(t) = \mathscr{T}_i^{>} \cap \{t^{>} \in \mathscr{T}^{>} \mid \lambda(t^{>}) = t\}$ collects all corresponding related transitions of the subnet.

First, the *places* of the *flow subformulas* have to be substituted by the corresponding places tracking the chain, i.e., all occurrences of a place $p \in \mathscr{P}$ in a flow subformula $\varphi_{\mathbb{A}_i}$ are simultaneously replaced by $[p]_i$. Second, the *transitions* of the flow subformulas have to be substituted such that all steps of the global timeline which do not involve the tracked flow chain are skipped until a transition involving the flow chain is fired, i.e., all occurrences of a transition $t \in \mathscr{T}$ in a flow subformula $\varphi_{\mathbb{A}_i}$ are simultaneously substituted by $(\bigvee_{t_o \in O_i} t_o)\,\mathcal{U}(\bigvee_{t_m \in M_i(t)} t_m)$. Similarly, the *next operators* of the flow subformulas have to be substituted such that the steps of the global timeline are skipped until a step involving the tracking subnet is taken. Here two cases have to be considered: either the chain ends, i.e., no transition of the subnet is ever fired again, then the formula has to directly hold in the stuttering part, or there is a transition of the subnet, then the formula has to hold in the direct successor state. This means all occurrences of a

Tab. 6.2: *An overview of the necessary substitutions to create $\varphi^{>}$ from φ. The next operator is substituted from the innermost to the outermost occurrence, the other subformulas are substituted simultaneously.*

	Run part of φ	Flow subformula $\mathbb{A}\,\psi_i$ part of φ
$p \in \mathcal{P}$	p	$[p]_i$
$t \in \mathcal{T}$	$\bigvee_{t' \in \{t' \in \mathcal{T}^{>} \mid \lambda(t') = t\}} t'$	$(\bigvee_{t_o \in O_i} t_o)\,\mathcal{U}\,(\bigvee_{t_m \in M_i(t)} t_m)$
$\bigcirc\,\phi$	$\bigcirc\,\phi$	$((\bigvee_{t \in O_i} t)\,\mathcal{U}\,((\bigvee_{t \in \mathcal{T}_i^{>}} t) \wedge \bigcirc\,\phi)) \vee (\Box\,(\neg(\bigvee_{t \in \mathcal{T}_i^{>}} t)) \wedge \phi)$
$\mathbb{A}\,\psi_i$	$[\iota]_i\,\mathcal{W}\,(\neg[\iota]_i \wedge \psi_i')$	–

subformula $\bigcirc\,\phi$ in a flow subformula $\varphi_{\mathbb{A}_i}$ are replaced from the inner- to the outermost occurrence by $((\bigvee_{t \in O_i} t)\,\mathcal{U}\,((\bigvee_{t \in \mathcal{T}_i^{>}} t) \wedge \bigcirc\,\phi)) \vee (\Box\,(\neg(\bigvee_{t \in \mathcal{T}_i^{>}} t)) \wedge \phi)$. These substitutions follow exactly the substitutions of the sequential approach.

For the *run part* of the formula, we can directly use the global timeline, i.e., the *next operator* needs no substitution. Further, the *places* are already correctly named. Only the *transitions* $t \in \mathcal{T}$ in the run part of φ have to be substituted simultaneously by $\bigvee_{t' \in \{t' \in \mathcal{T}^{>} \mid \lambda(t') = t\}} t'$ to consider all transitions corresponding to t.

Finally, the flow subformulas are simultaneously substituted by $[\iota]_i\,\mathcal{W}\,(\neg[\iota]_i \wedge \psi_i')$ (where ψ_i' is the result of the above mentioned substitutions within a flow subformula) such that all steps of the global timeline are skipped until a flow chain is created and tracked. Table 6.2 gives an overview of these substitutions.

This construction yields an LTL formula of polynomial size of the input net $\mathcal{N}_T$ and formula φ, but is of exponential size in the number n of flow subformulas.

▶ **Lemma 14 (Size of the Constructed Formula).** *The size of the constructed LTL formula $\varphi_{\|}^{>}$ is in $\mathcal{O}(|\mathcal{N}_T|^{3n} \cdot |\varphi| + |\varphi|)$.* ◀

Proof. The size of the constructed formula $\varphi_{\|}^{>}$ directly follows from the number of transitions added during the creation of $\mathcal{N}_{\|}^{>}$ (Lemma 13) and the substitutions introducing the disjunctions over these transitions in the creation of $\varphi_{\|}^{>}$. □

Note that there is only a significant blow-up in the formula when transitions are used as atomic propositions in either the flow or the run part of the formula, or when the next operator is used in the flow part of the formula. Moreover, even the usage of transitions as atomic propositions in the run part of the formula only leads to a disjunction over the combinations of transits of this transition with respect to the subnets, whereas the sequential approach uses a large subset of all transitions and an until operator. Oftentimes, this results (without any further optimizations) in smaller formulas for the parallel approach, because many examples need fairness assumptions, i.e., transitions in the run part of the formula, but have only few local requirements (cp. Chap. 7). Finally, even though model checking is exponential in the size of the formula [CHVB18], in practical applications the size of the model rather than the size of the formula turned out to be the driving factor for the running time anyhow [DLS06].

As an example we again transform the formula φ_1 of Example 7.

▶ **Example 12.** Given the Flow-LTL formula $\varphi_1 = (\texttt{max} \wedge \texttt{fair}) \rightarrow \mathbb{A} \Diamond \mathit{out}$ of Example 7 with $\texttt{max} = \Box((\mathit{in} \vee (\mathit{in} \wedge \mathit{out})) \rightarrow (s \vee t))$ and $\texttt{fair} = \Diamond\Box(\mathit{in} \wedge \mathit{out}) \rightarrow \Box\Diamond t$, and the constructed Petri net $\mathcal{N}_{\|}^{>}$ depicted in Fig. 6.5. The construction from this section yields the LTL formula

$$\varphi_{\|}^{>} = (\texttt{max}_{\|}^{>} \wedge \texttt{fair}_{\|}^{>}) \rightarrow ([\iota]_1 \mathscr{W} (\neg[\iota]_1 \wedge \Diamond[\mathit{out}]_1))$$

with $\texttt{max}_{\|}^{>} = \Box((\mathit{in} \vee (\mathit{in} \wedge \mathit{out})) \rightarrow ((s_1 \vee s_2 \vee s_3) \vee (t_1 \vee t_2 \vee t_3 \vee t_4))$ and $\texttt{fair}_{\|}^{>} = \Diamond\Box(\mathit{in} \wedge \mathit{out}) \rightarrow \Box\Diamond(t_1 \vee t_2 \vee t_3 \vee t_4)$. We see that we can especially spare three until operators and several transitions in the run part of the formula in comparison to formula $\varphi_1^{>}$ transformed with the sequential approach presented in Example 9. ◀

The constructions of the standard Petri net $\mathcal{N}_{\|}^{>}$ and the LTL formula $\varphi_{\|}^{>}$ yield another model checking procedure for Petri nets with transits and Flow-LTL. Analogously to the sequential approach, we again show the correctness of the construction by showing its soundness and completeness.

▶ **Lemma 15 (Correctness of the Transformation).** *For a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ, the constructions of the safe P/T Petri net $\mathcal{N}_{\|}^{>}$ with inhibitor arcs and the LTL formula $\varphi_{\|}^{>}$ yield $\mathcal{N}_T \models \varphi$ iff $\mathcal{N}_{\|}^{>} \models_{\texttt{LTL}} \varphi_{\|}^{>}$.* ◀

Proof. The proof of the correctness of the transformations for the parallel approach is analog to the one of the sequential approach given in Sec. 6.4.3. The formula replacements in the flow part of the formula follow the same idea and have just adapted the sets of related and unrelated transitions with respect to adaptions to the Petri net transformation. Thus, we can again mutually transform the counterexample to show the contraposition $\mathcal{N}_T \not\models \varphi$ iff $\mathcal{N}_{\|}^{>} \not\models_{\texttt{LTL}} \varphi_{\|}^{>}$.

Soundness. Here we do not have to pump up the covering firing sequence ζ of the run $\beta = (\mathcal{N}_T^R, \rho)$ serving as counterexample for $\mathcal{N}_T \models \varphi$ as in the sequential approach, because each firing of a transition in $\mathcal{N}_{\|}^{>}$ already updates all subnets tracking the data flows simultaneously. We only have to replace each transition t in ζ by a transition which adequately extends all flow chains of the counterexample. Thus, for each flow subformula $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\psi_i$ we use the witness flow chain ξ^i with $\sigma_F(\xi^i) \not\models_{\texttt{LTL}} \psi_i$ to decide which of the $(|\Upsilon^R(t)| + 1)^n$ transitions to choose. For those subformulas $\beta, \sigma(\zeta) \models \mathbb{A}\psi_i$ the chosen transition should not start tracking the chain. The run β determines the start point of replacing transitions according to the flow chain ξ^i.

Completeness. For the other direction, we can replace the transitions of the counterexample by the labels of the transitions and, analog to the sequential approach, iteratively concatenate the transitions and places of the subnets to gain the flow chains serving as counterexamples for the subformula part.

The complicated parts of the structural induction, i.e., adequately skipping the global time steps for the flow subformulas, can be done analogously because the formulas of the parallel approach and the sequential approach are similar in this case and fit to the different structure of the net. □

We show that for our running example $\mathcal{N}_{\|}^{>} \not\models_{\text{LTL}} \varphi_{\|}^{>}$ holds by translating the counterexample firing sequence given in Example 7 showing that our running example Petri net with transits $\mathcal{N}_A$ does not satisfy the formula φ_1.

▶ **Example 13.** Again we use the counterexample firing sequence with the trace $\sigma_R(\zeta) = M_0 \cup \{s\}, M_0 \cup \{t\}, M_0 \cup \{t\}, \ldots$, with $M_0 = \{in, out\}$, of Example 7 that shows that the Flow-LTL formula φ_1 is not satisfied by the Petri nets with transits $\mathcal{N}_A$. We now argue that also the LTL formula $\varphi_{\|}^{>}$ from Example 12, which is constructed from φ_1 and $\mathcal{N}_{\|}^{>}$, is not satisfied in the Petri net $\mathcal{N}_{\|}^{>}$ depicted in Fig. 6.5. For creating the firing sequence ζ' serving as counterexample we use again the flow chain with the trace $\sigma_F(\xi) = \{in, t\}, \{in, t\}, \ldots$ of Example 7 serving as counterexample for the satisfaction of the flow subformula. Consider the firing sequence with the trace $\sigma_R(\zeta') = In^{>} \cup \{s_1\}, M \cup \{t_1\}, M \cup \{t_1\}, \cdots$ with $In^{>} = \{in, out, [\iota]_1\}$ and $M = \{in, out, [in]_1\}$. We decided to use the s-labeled transition s_1 for s to start the tracking of the violating chain and decided to use the t-labeled transition t_1 for each transition t in $\sigma_R(\zeta)$ because the trace of the counterexample flow chain $\sigma_F(\xi)$ stayed in place in.

The firing sequence ζ' is actually a counterexample for $\mathcal{N}_{\|}^{>}$ satisfying $\varphi_{\|}^{>}$. The constraint $\texttt{max}_{\|}^{>}$ is satisfied because each state is left with transition s_1 or t_1 and $\texttt{fair}_{\|}^{>}$ is satisfied because transition t_1 occurs infinitely often in $\sigma_R(\zeta')$. However, $[\iota]_1 \notin M$ but also $[out]_1 \notin M$. Hence, $\mathcal{N}_{\|}^{>} \not\models_{\text{LTL}} \varphi_{\|}^{>}$. ◀

6.3 Petri Net Model Checking with Circuits

In the previous sections we reduce the model checking problem for a safe Petri net with transits and a Flow-LTL formula to the model checking problem of a safe standard Petri net $\mathcal{N}$ and an LTL formula ψ with places and transitions as atomic propositions. In this section we introduce a further reduction method and encode the safe Petri net $\mathcal{N}$ in a circuit $\mathcal{C}_{\mathcal{N}}$ and slightly transform ψ to an LTL formula ψ' such that the Petri net $\mathcal{N}$ satisfies ψ if and only if the circuit $\mathcal{C}_{\mathcal{N}}$ satisfies ψ'. This allows us in our tool AdamMC to use MCHyper [FRS15] to create a single circuit from the constructed circuit for the system $\mathcal{C}_{\mathcal{N}}$ and the constructed LTL formula ψ' and therewith enables the use of modern hardware model checkers such as ABC [BM10b; Ber] to answer the question whether $\mathcal{N} \models_{\text{LTL}} \psi$ holds (cp. Chap. 7). Thus, we create the circuit in this section in the style of [FRS15].

6.3.1 Construction of the Circuit

In the first step of the reduction we create the circuit $\mathcal{C}_{\mathcal{N}}$ simulating the given input Petri net $\mathcal{N}$. Therefore, we first define the structure of a circuit.

▶ **Definition 36 (Circuit).** A *circuit* $\mathcal{C} = (\mathcal{I}, \mathcal{O}, \mathcal{L}, \mathcal{F})$ consists of Boolean variables $\mathcal{I}$, $\mathcal{O}$, $\mathcal{L}$ for input, output, latches, and a Boolean formula $\mathcal{F}$ over $\mathcal{I} \times \mathcal{L} \times \mathcal{O} \times \mathcal{L}$, which is deterministic in $\mathcal{I} \times \mathcal{L}$.

The formula $\mathcal{F}$ can be seen as a transition relation from a valuation of the input variables and the current state of the latches to the valuation of the output variables and the next state of the latches. We use decorations to express the correspondence of the variables in the second and fourth component $\mathcal{L}$ of $\mathcal{F}$. Thus, if x denotes the current value of a latch in the second component $\mathcal{L}$ of $\mathcal{F}$ then x' denotes the new value of that latch after the next clock pulse in the fourth component $\mathcal{L}$ of $\mathcal{F}$. This decoration is also lifted to sets.

For a tuple $(I, L, O, L') \in 2^{\mathcal{I}} \times 2^{\mathcal{L}} \times 2^{\mathcal{O}} \times 2^{\mathcal{L}}$ we say (I, L, O, L') *satisfies* $\mathcal{F}$ (denoted by $(I, L, O, L') \models \mathcal{F}$) iff $\mathcal{F}$ is satisfied under the valuation which maps each occurring variable to `true` and all others to `false`. ◀

A circuit $\mathcal{C}$ can be interpreted as a Kripke structure $\mathscr{K}_{\mathcal{C}}$ such that the satisfaction of a formula ψ (denoted by $\mathcal{C} \models \psi$) can be defined by the satisfaction in the Kripke structure $\mathscr{K}_{\mathcal{C}} \models \psi$. A formal definition of the transformation is given in Definition 42 on page 119.

We now define the circuit $\mathcal{C}_{\mathcal{N}}$ simulating $\mathcal{N}$ such that

$$\mathcal{N} \models_{\text{LTL}} \psi \text{ iff } \mathcal{C}_{\mathcal{N}} \models \psi'$$

holds for an LTL formula ψ' constructed from ψ in the next section. The circuit $\mathcal{C}_{\mathcal{N}}$ has a latch for each place $p \in \mathscr{P}$ to store the current *marking*, a latch `i` for *initializing* this marking with *In* in the first step, and a latch `e` for handling *invalid* inputs. The inputs $\mathcal{I}$ consider the firing of a transition $t \in \mathscr{T}$. The latch `i` is `true` in every but the first step. The latch `e` is `true` whenever invalid values are applied on the inputs, i.e., the firing of not enabled, or more than one transition. The marking latches are updated according to the firing of the valid transition. If currently no valid input is applied, the marking is kept from the previous step. There is an output for each place (the current marking), for each transition (the transition leading to the next marking), and for the current value of the invalid latch. These properties are formalized in the following definition.

▶ **Definition 37 (P/T Petri Net to Circuit).** For a P/T Petri net with inhibitor arcs $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \mathscr{F}_I, In)$, we define the *circuit* $\mathcal{C}_{\mathcal{N}} = (\mathcal{I}, \mathcal{O}, \mathcal{L}, \mathcal{F})$ with the set of *input variables* $\mathcal{I} = \mathscr{T}$, the set of *output variables* $\mathcal{O} = \{p_o \mid p \in \mathscr{P}\} \cup \{t_o \mid t \in \mathscr{T}\} \cup \{e_o\}$, the set of *latches* $\mathcal{L} = \mathscr{P} \cup \{\mathtt{i}, \mathtt{e}\}$ with an *initialisation latch* `i` and a latch for *invalid inputs* `e`, and a *Boolean formula* $\mathcal{F} = \mathtt{out}_P \wedge \mathtt{out}_T \wedge \mathtt{out}_{\mathtt{e}} \wedge \mathtt{latch}_{\mathtt{e}} \wedge \mathtt{latch}_{\mathtt{i}} \wedge \mathtt{latch}_P$ over $\mathcal{I} \times \mathcal{L} \times \mathcal{O} \times \mathcal{L}$ which is defined with the help of the following formulas:

$$\mathtt{val}(t) = t \wedge \bigwedge_{t_1 \in \mathscr{T} \setminus \{t\}} \neg t_1 \wedge \bigwedge_{p \in pre(t)} \begin{cases} \neg p & \text{if } (p,t) \in \mathscr{F}_I \\ p & \text{otherwise} \end{cases},$$

$$\mathtt{noT} = \bigwedge_{t \in \mathscr{T}} \neg \mathtt{val}(t),$$

$$\mathtt{succ}(p) = (\mathtt{noT} \rightarrow p) \wedge (\neg \mathtt{noT} \rightarrow \bigwedge_{t \in \mathscr{T}} (t_o \rightarrow \begin{cases} p & \text{if } p \notin pre(t) \wedge p \notin post(t) \\ 0 & \text{if } p \in pre(t) \wedge p \notin post(t) \\ 1 & \text{otherwise} \end{cases})).$$

The formula $\texttt{val}(t)$ for a $t \in \mathscr{T}$ states the validity of t, i.e., t is set as input but no other transition is set and t is enabled by the current state of the latches. The formula $\texttt{noT}$ is true iff no transition is valid and the formula $\texttt{succ}(p)$ for a place $p \in \mathscr{P}$ defines the successor value for p. If there is no valid input we keep the same marking. Otherwise, the marking is the successor marking of the current output transition t_o and the current marking. Therewith, the conjuncts of $\mathscr{F}$ are defined as follows:

$$
\begin{aligned}
\texttt{out}_\texttt{P} &= \bigwedge_{p \in \mathscr{P}} (p_o \leftrightarrow (\neg \texttt{i} \rightarrow p') \wedge (\texttt{i} \rightarrow p)),\\
\texttt{out}_\texttt{T} &= \bigwedge_{t \in \mathscr{T}} (t_o \leftrightarrow \texttt{val}(t)),\\
\texttt{out}_\texttt{e} &= e_o \leftrightarrow \texttt{e},\\
\texttt{latch}_\texttt{e} &= e' \leftrightarrow \texttt{i} \wedge \texttt{noT},\\
\texttt{latch}_\texttt{i} &= i' \leftrightarrow \mathit{true},\\
\texttt{latch}_P &= \bigwedge_{p \in \mathscr{P}} (p' \leftrightarrow \begin{cases} \texttt{i} \rightarrow \texttt{succ}(p) & \text{if } p \in \mathit{In} \\ \texttt{i} \wedge \texttt{succ}(p) & \text{otherwise} \end{cases}).
\end{aligned}
$$

In all states but the initial one, the outputs corresponding to places are the current values of the latches. The outputs corresponding to the transitions are at most one valid transition. The new value for the latches corresponding to places are initially the initial marking of $\mathscr{N}$. Otherwise, if no valid input is applied, the current values of the latches are copied to the new values and if there is a valid transition, the successor marking of firing this transition in the current values is used for the new values. ◀

6.3.2 Transformation of the Formula and the Correctness

The second step of the reduction is the translation of the input LTL formula ψ to an LTL formula ψ'. The atomic propositions of ψ are $AP = \mathscr{T} \cup \mathscr{P}$, but formulas satisfied by the circuit $\boldsymbol{C}_{\mathscr{N}}$ can only range over the output variables of $\boldsymbol{C}_{\mathscr{N}}$. Hence, the LTL formula $\widetilde{\psi}$ is obtained from ψ by replacing every place and transition atom with the corresponding output variable:

$$\widetilde{\psi} = \psi\,[p_{1_o}/p_1, \ldots, p_{n_o}/p_n, t_{1_o}/t_1, \ldots, t_{m_o}/t_m]$$

for $\mathscr{P} = \{p_1, \ldots, p_n\}$, $\mathscr{T} = \{t_1, \ldots, t_m\}$, and the operator $[\phi'_1/\phi_1, \ldots, \phi'_k/\phi_k]$ on formulas for the simultaneous substitution of formula ϕ_j by formula ϕ'_j for $j \in \{1, \ldots, k\}$. Now we define the LTL formula ψ' by skipping the initialization step of the circuit and by focusing on the valid traces:

$$\psi' = \bigcirc(\square(e_o \rightarrow \square e_o) \rightarrow \widetilde{\psi}).$$

By tolerating invalid inputs only at the end of a trace, we allow for finite firing sequences of a run without enforcing any maximality constraints on the firing sequences. These could be stated in the formula ψ as defined in Example 1 with the additional next operator for the interleaving-maximality for the outgoing semantics.

These constructions yield a circuit with a linear number of latches and a quadratic number of gates (i.e., conjunctions and disjunctions in $\mathcal{F}$) for the input Petri net $\mathcal{N}$ and an LTL formula of linear size of the input formula ψ.

▶ **Lemma 16 (Sizes of the Transformation).** *For a safe P/T Petri net with inhibitor arcs $\mathcal{N}$ and an LTL formula ψ, the constructed circuit $\mathcal{C}_\mathcal{N}$ has $|\mathcal{P}| + 2$ latches and $\mathcal{O}(|\mathcal{N}|^2)$ gates, and the constructed formula ψ' is of size $\mathcal{O}(|\psi|)$.* ◀

Proof. The number of latches and gates directly follows from Definition 37. For the gates we can see that none of the subformulas $\texttt{out}_P$, $\texttt{out}_T$, $\texttt{out}_\texttt{e}$, $\texttt{latch}_\texttt{e}$, $\texttt{latch}_\texttt{i}$, $\texttt{latch}_P$ has more than two nesting conjunctions over $\mathcal{P}$ or $\mathcal{T}$. □

Putting this all together, we can state the final result and show the correctness of this reduction resulting in a single-exponential time algorithm for solving safe Petri nets with inhibitor arcs against LTL formulas via circuits.

▶ **Theorem 5.** *For a safe P/T Petri net with inhibitor arcs $\mathcal{N}$ and an LTL formula ψ, the constructed circuit $\mathcal{C}_\mathcal{N}$ and the constructed formula ψ' yield $\mathcal{N} \models_{\text{LTL}} \psi$ iff $\mathcal{C}_\mathcal{N} \models \psi'$. The model checking problem for the circuit and the LTL formula can be solved in single-exponential time in the size of $\mathcal{N}$ and ψ.* ◀

Proof Sketch. The correctness of the construction is proven using the Kripke structure $\mathcal{K}_{\mathcal{C}_\mathcal{N}}$ of Definition 42 corresponding to the circuit $\mathcal{C}_\mathcal{N}$. We can show the contraposition $\mathcal{N} \not\models_{\text{LTL}} \psi$ iff $\mathcal{K}_{\mathcal{C}_\mathcal{N}} \not\models \psi'$ by straightforwardly transforming the counterexample firing sequence ζ covering a run of $\mathcal{N}$ to a path π in the Kripke structure $\mathcal{K}_{\mathcal{C}_\mathcal{N}}$ and vice versa.

The single-exponential time bound stems from the polynomial size of the circuit and the linear size of the formula (Lemma 16), and the exponential size of the Kripke structure (Definition 42). Which in turn can be checked in linear time in the size of the Kripke structure and in exponential time in the size of the formula [CHVB18]. For more details see the proof on page 120. □

6.4 Proofs and Formal Constructions

In this section, we provide details to the previous two sections. Particularly, we give in Sec. 6.4.1 a formal definition of the construction of the Petri net with inhibitor arcs $\mathcal{N}_{\gg}^{>}$ from a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ described in Sec. 6.2.1 and give in Sec. 6.4.2 the formal definition for the corresponding transformation of φ to an LTL formula $\varphi_{\gg}^{>}$ described in Sec. 6.2.1. Furthermore, the correctness proofs for these transformations are given in Sec. 6.4.3. In Sec. 6.4.4 we show the correctness of the circuit construction of Sec. 6.3 for reducing the model checking problem of a safe Petri net with inhibitor arcs and an LTL formula to a hardware model checking problem. Finally, the formal definition of the construction of the Petri net with inhibitor arcs $\mathcal{N}_{\|}^{>}$ from a Petri net with transits $\mathcal{N}_T$ and a Flow-LTL formula φ described in Sec. 6.2.2 is given in Sec. 6.4.5.

6.4.1 Formal Construction of the Petri Net $\mathcal{N}_{\gg}^{>}$

In this section we give a formal definition for the construction described in Sec. 6.2.1 for creating the Petri net with inhibitor arcs $\mathcal{N}_{\gg}^{>}$ given a Petri net with transits $\mathcal{N}_T$ and the number $n \in \mathbb{N}$ of flow subformulas of a Flow-LTL formula φ.

We introduce a set of identifiers $\mathtt{ID}$ and an injective naming function $\nu_{\mathcal{N}} : \mathcal{P} \cup \mathcal{T} \to \mathtt{ID}$ for every Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ (and all of its extensions) which uniquely identifies every place and transition of a given net. If the net $\mathcal{N}$ is clear from the context, we omit the subscript and only write ν. Furthermore, we often omit the predicate $p \in \mathcal{P} \wedge \nu(p) = \mathit{identifier}$ and $t \in \mathcal{T} \wedge \nu(t) = \mathit{identifier}$, respectively, in formulas and only use *identifier* instead of p or t, respectively, within the formula to keep the presentation short.

The construction of a Petri net with transits to a standard P/T Petri net with inhibitor arcs is given by the following definition.

▶ **Definition 38 (Petri Net with Transits to a P/T Petri Net).** Given a safe Petri net with transits $\mathcal{N}_T = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ and a Flow-LTL formula φ with $n \in \mathbb{N}$ subformulas $\mathbb{A}\,\psi_i$, for $i = 1, \ldots, n$. We define a P/T Petri net with inhibitor arcs $\mathcal{N}_{\gg}^{>} = (\mathcal{P}^{>}, \mathcal{T}^{>}, \mathcal{F}^{>}, \mathcal{F}_I^{>}, In^{>})$, with

$$\mathcal{P}^{>} = \mathcal{P}_o^{>} \uplus \biguplus_{i \in \{1,\ldots,n\}} \mathcal{P}_i^{>}, \qquad \mathcal{T}^{>} = \mathcal{T}_o^{>} \uplus \biguplus_{i \in \{1,\ldots,n\}} \mathcal{T}_i^{>}$$

and a partial function $\lambda : \mathcal{T}^{>} \cup \mathcal{P}^{>} \to \mathcal{T} \cup \mathcal{P}$ which maps the elements to its corresponding original ones. The smallest sets $\mathcal{P}_o^{>}, \mathcal{P}_i^{>}, \mathcal{T}_o^{>}, \mathcal{T}_i^{>}, \mathcal{F}^{>}$, and $\mathcal{F}_I^{>}$ fulfilling the following constraints define the net $\mathcal{N}_{\gg}^{>}$.

By constraint **(o)** it is ensured that all places, transitions, and flows of the original net $\mathcal{N}_T$ are also existent in $\mathcal{N}_{\gg}^{>}$. The labeling and the identifiers are copied.

(o) $\mathcal{P}_o^{>} \supset \mathcal{P} \wedge \mathcal{T}_o^{>} = \mathcal{T} \wedge \mathcal{F}^{>} \supset \mathcal{F} \wedge \forall p^{>} \in \mathcal{P} : \lambda(p^{>}) = p \wedge \forall t^{>} \in \mathcal{T} : \lambda(t^{>}) = t \wedge \forall p^{>} \in \mathcal{P} : \nu(p^{>}) = \nu(p) \wedge \forall t^{>} \in \mathcal{T} : \nu(t^{>}) = \nu(t)$

With the following five constraints, the additional places, transitions, and flows of the subnets are defined for each subformula $\mathbb{A}\,\psi_i$. Let $I = \{1, \ldots, n\}$. In **(s1)**, a copy of every original place for each subnet is required for tracking the flow chains.

(s1) $\forall i \in I : \forall p \in \mathcal{P} : \exists p^{>} \in \mathcal{P}_i^{>} : \lambda(p^{>}) = p \wedge \nu(p^{>}) = [\nu(p)]_i$

To check whether any chain has been tracked, an initial place $[\iota]_i$ for every subnet is defined via constraint **(s2)**.

(s2) $\forall i \in I : \exists p^{>} \in \mathcal{P}_i^{>} : \nu(p^{>}) = [\iota]_i$

Constraint **(s3)** ensures the existence of transitions simulating the creation of a flow chain during the run. Hence, for every starting flow chain there is a transition in each subnet, which takes the initial token from $[\iota]_i$ and moves it according to the corresponding transit.

(s3) $\forall i \in I : \forall t \in \mathscr{T} : \forall (\triangleright, q) \in \Upsilon(t) : \exists t^{>} \in \mathscr{T}_i^{>} : ([\iota]_i, t^{>}) \in \mathscr{F}^{>} \wedge (t^{>}, [\nu(q)]_i) \in \mathscr{F}^{>} \wedge \lambda(t^{>}) = t \wedge \nu(t^{>}) = \nu(t)_{\nu(q)_i}$

In constraint **(s4)**, this is similarly done for each transit of each transition.

(s4) $\forall i \in I : \forall t \in \mathscr{T} : \forall (p, q) \in \Upsilon(t) : \exists t^{>} \in \mathscr{T}_i^{>} : ([\nu(p)]_i, t^{>}) \in \mathscr{F}^{>} \wedge (t^{>}, [\nu(q)]_i) \in \mathscr{F}^{>} \wedge \lambda(t^{>}) = t \wedge \nu(t^{>}) = \nu(t)_{(\nu(p),\nu(q))_i}$

Constraint **(s5)** treats the situation that the currently tracked flow chain is independent of the last fired transition $t \in \mathscr{T}_o^{>}$. Thus, in each subnet a skipping transition is required for every original transition $t \in \mathscr{T}$ which is only allowed to fire when the considered chain of the subnet is not extended by the transits of t, i.e., no corresponding place in the preset of t is occupied.

(s5) $\forall i \in I : \forall t \in \mathscr{T} : \exists t^{>} \in \mathscr{T}_i^{>} : \forall p \in pre(t) : ([\nu(p)]_i, t^{>}) \in \mathscr{F}_I^{>} \wedge \lambda(t^{>}) = t \wedge \nu(t^{>}) = \nu(t)_{\Rrightarrow_i}$

The following four constraints are used to connect the components sequentially. Constraint **(a)** ensures the existence of one activation place $\overrightarrow{o}$ for the original part of the net, and one activation place $\overrightarrow{\nu(t)}_i$ for every original transition $t \in \mathscr{T}$ for each subnet.

(a) $\exists p^{>} \in \mathscr{P}_o^{>} : \nu(p^{>}) = \overrightarrow{o} \wedge \forall i \in I : \forall t \in \mathscr{T} : \exists p^{>} \in \mathscr{P}_i^{>} : \nu(p^{>}) = \overrightarrow{\nu(t)}_i$

Constraint **(mO)** lets every original transition $t \in \mathscr{T}_o^{>}$ take the activation token from $\overrightarrow{o}$ and moves it to the activation place $\overrightarrow{\nu(t)}_0$ for the transitions of the first subnet which are labeled with t to activate this subnet.

(mO) $\forall t \in \mathscr{T}_o : (\overrightarrow{o}, t) \in \mathscr{F}^{>} \wedge (t, \overrightarrow{\nu(t)}_0) \in \mathscr{F}^{>}$

With the constraints **(mSi)** and **(mSn)**, we move the activation token through the subnets, or back to the original part of the net, respectively. Therefore, we let all equally labeled transitions of the subnet take their corresponding activation token from the place $\overrightarrow{\nu(t)}_i$ for a label $t \in \mathscr{T}$ and move it to the next subnet, i.e., place $\overrightarrow{\nu(t)}_{i+1}$ or $\overrightarrow{o}$, respectively.

(mSi) $\forall i \in \{1, \ldots, n-1\} : \forall t \in \mathscr{T} :$
$\forall t^{>} \in \mathscr{T}_i^{>} : \lambda(t^{>}) = t \implies ((\overrightarrow{\nu(t)}_i, t^{>}) \in \mathscr{F}^{>} \wedge (t^{>}, \overrightarrow{\nu(t)}_{i+1}) \in \mathscr{F}^{>})$

(mSn) $\forall t \in \mathscr{T} : \forall t^{>} \in \mathscr{T}_n^{>} : \lambda(t^{>}) = t \implies ((\overrightarrow{\nu(t)}_n, t^{>}) \in \mathscr{F}^{>} \wedge (t^{>}, \overrightarrow{o}) \in \mathscr{F}^{>})$

The initial marking of $\mathscr{N}_{\gg}^{>}$ is defined by constraint **(in)**. We only activate the original part of the net and allow all subnets to track a chain.

(in) $In^{>} = \{\overrightarrow{o}\} \cup \{[\iota]_i \mid i \in I\} \cup In$.

Newly introduced identifiers, e.g., ι and $\overrightarrow{o}$, are unique and do not occur in $\mathscr{N}_T$. ◀

Note that in case $n = 0$ flow subformulas exist, we just use the input Petri net with transits $\mathcal{N}_T$ and omit the transits.

This definition directly yields the number of places and transitions of the constructed Petri net $\mathcal{N}_{\gg}^{>}$ stated in Lemma 10 on page 91.

Proof (Lemma 10: Size of the Constructed Net): The injectivity of ν yields that each required place or transition is a unique element of $\mathscr{P}^{>}$ or $\mathscr{T}^{>}$ respectively. That we demand that the smallest sets $\mathscr{P}^{>}$ and $\mathscr{T}^{>}$ fulfill the constraints allows us to only consider the explicitly stated elements.

Therewith, constraint **(o)** together with constraint **(a)** yield $|\mathscr{P}_o^{>}| = |\mathscr{P}|+1$. Constraint **(s1)** requires $|\mathscr{P}|$ places for each subnet, constraint **(s2)** requires one initial place for each subnet, and constraint **(a)** requires one activation place for each original transition for every subnet. Hence, $|\bigcup_{i=1,\dots,n} \mathscr{P}_i^{>}| = n \cdot |\mathscr{P}| + n + n \cdot |\mathscr{T}|$ and so $|\mathscr{P}^{>}| = n \cdot (|\mathscr{P}| + |\mathscr{T}| + 2) + |\mathscr{P}| + 1$.

For the transitions of the original part of the net, constraint **(o)** directly yields $|\mathscr{T}_o^{>}| = |\mathscr{T}|$. Constraint **(s3)** requires a transition for every newly created flow chain of the net, i.e., $|\{(t,p) \in \mathscr{T} \times \mathscr{P} \mid (\triangleright, p) \in \Upsilon(t)\}|$ many transitions, which are at most $|\mathscr{T}| \cdot |\mathscr{P}|$ transitions for each subnet. Constraint **(s4)** does the same for every transit of the net, i.e., $|\{(p,t,q) \in \mathscr{P} \times \mathscr{T} \times \mathscr{P} \mid (p,q) \in \Upsilon(t)\}|$ many transitions, which are at most $|\mathscr{P}| \cdot |\mathscr{T}| \cdot |\mathscr{P}|$ transitions for each subnet. Constraint **(s5)** requires one transition for skipping and directly moving the active token to the next subnet for each transition $t \in \mathscr{T}$. Hence, $|\bigcup_{i=1,\dots,n} \mathscr{T}_i^{>}| = n \cdot |\mathscr{T}| \cdot |\mathscr{P}| + n \cdot |\mathscr{P}| \cdot |\mathscr{T}| \cdot |\mathscr{P}| + n \cdot |\mathscr{T}|$ and thus, $\mathscr{T}^{>}$ contains $n \cdot (|\mathscr{P}|^2 \cdot |\mathscr{T}| + |\mathscr{P}| \cdot |\mathscr{T}| + |\mathscr{T}|) + |\mathscr{T}|$ transitions. □

6.4.2 Formal Construction of the Formula $\varphi_{\gg}^{>}$

In this section we formally introduce the transformation of a Flow-LTL formula φ to an LTL formula $\varphi_{\gg}^{>}$ presented in Sec. 6.2.1. We use the operator $[\phi_1'/\phi_1, \dots, \phi_m'/\phi_m]$ on formulas for the simultaneous substitution of ϕ_j by ϕ_j'. To substitute formulas from the inner- to the outermost, we utilize the function d, which calculates the depth of a formula. The *depth-function* d is inductively defined: $d(a) = 0$ for every atomic proposition a and $d(\circ\, \phi_1) = 1 + d(\phi_1)$, $d(\phi_1 \mathbin{\widetilde{\circ}} \phi_2) = 1 + max\{d(\phi_1), d(\phi_1)\}$ for every unary operator $\circ$, binary operator $\widetilde{\circ}$, and formulas ϕ_1 and ϕ_2.

▶ **Definition 39 (Flow-LTL to LTL).** Let $\mathcal{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ be a Petri net with transits, φ a Flow-LTL formula with $n \in \mathbb{N}$ subformulas $\mathbb{A}\,\psi_i$, for $i = 1, \dots, n$, and $\mathcal{N}_{\gg}^{>} = (\mathscr{P}^{>}, \mathscr{T}^{>}, \mathscr{F}^{>}, \mathscr{F}_I^{>}, In^{>})$ the with Definition 38 created P/T Petri net with inhibitor arcs. The corresponding LTL formula $\varphi_{\gg}^{>}$ is created by the following steps:

Flow Part: For each flow formula $\mathbb{A}\,\psi_i$, for $i = 1, \dots, n$, we create a new formula $\psi_i^{\mathscr{P}\mathscr{T}\mathscr{X}_{d_{\max}}}$ which adequately copes with the different timelines of the corresponding flow chains. Since in our approach each flow formula is checked on the corresponding subnet $\mathcal{N}_i^{>}$, the places and transitions of the other components are ignored.

The atomic propositions $p \in \mathscr{P}$ are substituted with the corresponding places of the subnet:

(pF) $\psi_i^{\mathscr{P}} = \psi_i \left[\begin{array}{c} [\nu(p_1)]_i/p_1, \\ \ldots, \\ [\nu(p_m)]_i/p_m \end{array} \right]$ for $\mathscr{P} = \{p_1, \ldots, p_m\}$.

For the atomic propositions $t \in \mathscr{T}$, we skip all transitions not concerning the extension of the current flow chain via an until operator. That means the firing of *unrelated* transitions $O_i = (\mathscr{T}^> \setminus \mathscr{T}_i^>) \cup \{\nu(t)_{\Rrightarrow_i} \in \mathscr{T}_i^> \mid t \in \mathscr{T}\}$, i.e., transitions of the other components or own skipping transitions, is skipped until a *related* transition, i.e., one of the set of transitions extending the flow chain $M_i(t) = \{t^> \in \mathscr{T}_i^> \setminus \{\nu(t)_{\Rrightarrow_i}\} \mid \lambda(t^>) = t\}$, is fired.

(tF) $\psi_i^{\mathscr{P}\mathscr{T}} = \psi_i^{\mathscr{P}} \left[\begin{array}{c} (\bigvee_{t_o \in O_i} t_o)\, \mathcal{U} (\bigvee_{t \in M_i(t_1)} t)/t_1, \\ \ldots, \\ (\bigvee_{t_o \in O_i} t_o)\, \mathcal{U} (\bigvee_{t \in M_i(t_{m'})} t)/t_{m'} \end{array} \right]$ for $\mathscr{T} = \{t_1, \ldots, t_{m'}\}$.

The next operator is treated similarly. Let $\psi_i^{\mathscr{P}\mathscr{T}}$ contain m_1 subformulas $\bigcirc \psi'_j$ for $j = 1, \ldots, m_1$. In this case, the *related* transitions are all transitions of the subnet $\mathscr{T}_i^>$ except for the skipping transitions: $M_i = \mathscr{T}_i^> \setminus \{\nu(t)_{\Rrightarrow_i} \in \mathscr{T}_i^> \mid t \in \mathscr{T}\}$. We define the disjunction of all related transitions by $\widetilde{M}_i = \bigvee_{t \in M_i} t$ and apply the same ideas as in the previous case. To adequately cope with situations where no related transition $t \in M_i$ would ever fire again ($\Box(\neg \widetilde{M}_i)$), i.e., the stuttering at the end of a chain, we require the immediate satisfaction. To replace the formulas from the inner- to the outermost, we organize the formulas in groups $\{\bigcirc \psi_1^l, \ldots, \bigcirc \psi_{k_l}^l\}$ according to their depth:

(nF) Let $\{\bigcirc \psi_1^l, \ldots, \bigcirc \psi_{k_l}^l\} = \{\bigcirc \psi'_j \mid j \in \{1, \ldots, m_1\} \wedge d(\bigcirc \psi'_j) = l\}$ and $d \in \{2, \ldots, d_{\mathtt{max}}\}$ with $d_{\mathtt{max}} = \mathtt{max}\{d(\bigcirc \psi'_j) \mid j \in \{1, \ldots, m_1\}\}$. Then, the substitutions of the next operators is given by

$$\psi_i^{\mathscr{P}\mathscr{T}\mathscr{X}_1} = \psi_i^{\mathscr{P}\mathscr{T}} \left[\begin{array}{c} ((\bigvee_{t \in O_i} t)\, \mathcal{U} (\widetilde{M}_i \wedge \bigcirc \psi_1^1)) \vee (\Box(\neg \widetilde{M}_i) \wedge \psi_1^1)/\bigcirc \psi_1^1, \\ \ldots, \\ ((\bigvee_{t \in O_i} t)\, \mathcal{U} (\widetilde{M}_i \wedge \bigcirc \psi_{k_1}^1)) \vee (\Box(\neg \widetilde{M}_i) \wedge \psi_{k_1}^1)/\bigcirc \psi_{k_1}^1 \end{array} \right]$$

and

$$\psi_i^{\mathscr{P}\mathscr{T}\mathscr{X}_d} = \psi_i^{\mathscr{P}\mathscr{T}\mathscr{X}_{d-1}} \left[\begin{array}{c} ((\bigvee_{t \in O_i} t)\, \mathcal{U} (\widetilde{M}_i \wedge \bigcirc \psi_1^d)) \vee (\Box(\neg \widetilde{M}_i) \wedge \psi_1^d)/\bigcirc \psi_1^d, \\ \ldots, \\ ((\bigvee_{t \in O_i} t)\, \mathcal{U} (\widetilde{M}_i \wedge \bigcirc \psi_{k_d}^d)) \vee (\Box(\neg \widetilde{M}_i) \wedge \psi_{k_d}^d)/\bigcirc \psi_{k_d}^d \end{array} \right]$$

Run part: They same ideas are applied for the run part of the formula. Here the atomic propositions $p \in \mathscr{P}$ do not need to be substituted since $\mathscr{P} \subset \mathscr{P}_o^>$ holds. For the atomic propositions $t \in \mathscr{T}$ however, the skipping procedure has to be applied as well. The *unrelated* transitions in this case are all transitions of the subnet $O = \mathscr{T}^> \setminus \mathscr{T}$. To only substitute occurrences in the run part of the formula, we introduce the substitution operator $/_{\bar{\mathbb{A}}}$ which does not change anything within the scope of the flow operator $\mathbb{A}$.

(tR) $\varphi^{\mathcal{T}} = \varphi \begin{bmatrix} (\bigvee_{t\in O} t)\,\mathcal{U}\, t_1/_{\bar{\mathbb{A}}} t_1, \\ \ldots, \\ (\bigvee_{t\in O} t)\,\mathcal{U}\, t_m/_{\bar{\mathbb{A}}} t_m \end{bmatrix}.$

For the next operator in the run part of the formula, the *related* transition are all transitions of $\mathcal{N}_O^{>}$, i.e., $\mathcal{T}$. We define the disjunction of these transitions by $T = \bigvee_{t\in\mathcal{T}} t$. To adequately cope with situations where no transition $t \in \mathcal{T}$ would ever fire again ($\Box(\neg T)$), i.e., the stuttering in the traces for finite firing sequences, we require the immediate satisfaction. Let the run part of $\varphi^{\mathcal{T}}$ contain m_2 subformulas $\bigcirc\varphi_j$ for $j = 1, \ldots, m_2$. We again organize the formulas according to their depth, to replace them from the inner- to the outermost.

(nR) Let $\{\bigcirc\varphi_1^l, \ldots, \bigcirc\varphi_{k_l}^l\} = \{\bigcirc\varphi_j \mid j \in \{1, \ldots, m_2\} \wedge d(\bigcirc\varphi_j) = l\}$ and $d' \in \{2, \ldots, d'_{\mathtt{max}}\}$ with $d'_{\mathtt{max}} = \mathtt{max}\{d(\bigcirc\varphi_j) \mid j \in \{1, \ldots, m_2\}\}$. Then, the substitutions of the next operators is given by

$$\varphi^{\mathcal{T}\mathcal{X}_1} = \varphi^{\mathcal{T}} \begin{bmatrix} ((\bigvee_{t\in O} t)\,\mathcal{U}(T \wedge \bigcirc\varphi_1^1)) \vee (\Box(\neg T) \wedge \varphi_1^1)/_{\bar{\mathbb{A}}} \bigcirc\varphi_1^1, \\ \ldots, \\ ((\bigvee_{t\in O} t)\,\mathcal{U}(T \wedge \bigcirc\varphi {k_1}^1)) \vee (\Box(\neg T) \wedge \varphi_{k_1}^1)/_{\bar{\mathbb{A}}} \bigcirc\varphi_{k_1}^1 \end{bmatrix}$$

and

$$\varphi^{\mathcal{T}\mathcal{X}_{d'}} = \varphi^{\mathcal{T}\mathcal{X}_{d'-1}} \begin{bmatrix} ((\bigvee_{t\in O} t)\,\mathcal{U}(T \wedge \bigcirc\varphi_1^{d'})) \vee (\Box(\neg T) \wedge \varphi_1^{d'})/_{\bar{\mathbb{A}}} \bigcirc\varphi_1^{d'}, \\ \ldots, \\ ((\bigvee_{t\in O} t)\,\mathcal{U}(T \wedge \bigcirc\varphi_{k_{d'}}^{d'})) \vee (\Box(\neg T) \wedge \varphi_{k_{d'}}^{d'})/_{\bar{\mathbb{A}}} \bigcirc\varphi_{k_{d'}}^{d'} \end{bmatrix}$$

Since flow chains can be created at any point in time during the run, we skip to their creation point, i.e., we skip as long as a token resides in $[\iota]_i$. To also enable to not track any chain, the weak until operator is used.

(𝔸R) $\varphi^{\mathbb{A}} = \varphi^{\mathcal{T}\mathcal{X}_{d'_{\mathtt{max}}}} \begin{bmatrix} [\iota]_1\,\mathscr{W}(\neg[\iota]_1 \wedge \psi_1^{\mathscr{P}\mathcal{T}\mathcal{X}_{d_{\mathtt{max}}}})/\mathbb{A}\,\psi_1, \\ \ldots, \\ [\iota]_n\,\mathscr{W}(\neg[\iota]_n \wedge \psi_n^{\mathscr{P}\mathcal{T}\mathcal{X}_{d_{\mathtt{max}}}})/\mathbb{A}\,\psi_n \end{bmatrix}$

Since we do not want any run or firing sequence to stop in any subnet $\mathcal{N}_i^{>}$, the final formula restricts the considered traces to those which infinitely often visit the activation place $\overrightarrow{o}$ of the original part of the net.

(nSub) $\varphi_{\gg}^{>} = \Box\Diamond\overrightarrow{o} \rightarrow \varphi^{\mathbb{A}}$

The last step **(nSub)** concludes the construction of the LTL formula $\varphi_{\gg}^{>}$. ◀

Note that in the case that no flow subformula occurs in φ, we can just use φ itself.

The proof of the size of the constructed formula $\varphi_{\gg}^{>}$ (Lemma 11 on page 93) directly results from Lemma 10 about the size of the Petri net $\mathcal{N}_{\gg}^{>}$, because the substitution of elements of φ completely depends on the size of $\mathcal{N}_{\gg}^{>}$. This net is quartic in the size of $\mathcal{N}_T$ and the number of flow subformulas in φ.

Proof (Lemma 11: Size of the Constructed Formula): For each occurrence of an atomic proposition $t \in \mathcal{T}$ in the *run part* of φ, we have additionally $1+2\cdot|\mathcal{T}^{>}\setminus\mathcal{T}|-1$ subformulas in $\varphi^{>}_{\gg}$ by constraint **(tR)**. Since $\mathcal{T}^{>}$ is quartic in the size of $\mathcal{N}_T$ and n, $\varphi^{>}_{\gg}$ is already quintic in the size of $\mathcal{N}_T$ and φ. For each occurrence of an atomic proposition $t \in \mathcal{T}$ in a *flow subformula* of φ we have additionally $1+2\cdot(|\mathcal{T}^{>}\setminus\mathcal{T}_i|+|\mathcal{T}|)-1+2\cdot(|M_i(t)|)-1$ subformulas in $\varphi^{>}_{\gg}$ (constraint **(tF)**). Since for $M_i(t)$ there are at most $|\mathcal{P}|^2+|\mathcal{P}|$ transits of a transition, $\mathcal{T}^{>}$ is still the biggest part and we are quintic in the size of $\mathcal{N}_T$ and φ. For each occurrence of a next operator in the *run part* of φ we have additionally $2+1+2\cdot|\mathcal{T}^{>}\setminus\mathcal{T}|-1+3+|\mathcal{T}|+1+3+1+|\mathcal{T}|$ subformulas in $\varphi^{>}_{\gg}$ (constraint **(nR)**). Hence, $\varphi^{>}_{\gg}$ is still quintic in the size of $\mathcal{N}_T$ and φ. For each occurrence of a next operator in a *flow subformula* of φ we have additionally $2+1+2\cdot(|\mathcal{T}^{>}\setminus\mathcal{T}_i|+|\mathcal{T}|)-1+3+|\mathcal{T}_i|-|\mathcal{T}|+1+3+1+|\mathcal{T}_i|-|\mathcal{T}|$ subformulas in $\varphi^{>}_{\gg}$ (constraint **(nF)**). Hence, $\varphi^{>}_{\gg}$ is still quintic in the size of $\mathcal{N}_T$ and φ.□

6.4.3 Correctness Proof of the Reduction Technique

In this section, we prove Lemma 12 stating the correctness of the reduction. We fix a Petri net with transits $\mathcal{N}_T = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \Upsilon, In)$, the corresponding Petri net with inhibitor arcs $\mathcal{N}^{>}_{\gg} = (\mathcal{P}^{>}, \mathcal{T}^{>}, \mathcal{F}^{>}, \mathcal{F}^{>}_I, In^{>})$ and the partial labeling function $\lambda : \mathcal{T}^{>} \cup \mathcal{P}^{>} \to \mathcal{T} \cup \mathcal{P}$ which maps the elements to its corresponding original ones, both created by Definition 38, a Flow-LTL formula φ with $n \in \mathbb{N}$ subformulas $\mathbb{A}\,\psi_i$, for $i = 1, \ldots, n$, and the corresponding LTL formula $\varphi^{>}_{\gg}$ which is created by Definition 39 throughout the section. The general idea is to show the contraposition of the statement:

$$\mathcal{N}_T \not\models \varphi \text{ iff } \mathcal{N}^{>}_{\gg} \not\models_{\text{LTL}} \varphi^{>}_{\gg}.$$

Therefore, we transform the counterexamples mutually by Definition 40 and Definition 41. For Definition 40, we sequentially pump up the firing sequence serving as counterexample for $\mathcal{N}_T \models \varphi$ by one transition for each subnet in every step. If the subnet has to consider a flow chain, i.e., $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\psi_i$ and the original transition transits the flow chain, the corresponding transition of the subnet is used. Otherwise, the corresponding skipping transition is added. The markings are filled with the additional tokens of $\mathcal{N}^{>}_{\gg}$.

▶ **Definition 40 (CEX: From PNwT to PN).** Let $\beta = (\mathcal{N}^R_T, \rho)$ be a run of $\mathcal{N}_T$, $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ be a covering firing sequence, and for every $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\psi_i$ let $\xi^i = t^i_\iota, p^i_0, t^i_0, p^i_1, t^i_1, \ldots$ be the corresponding flow chain with $\sigma_F(\xi^i) \not\models_{\text{LTL}} \psi_i$. We create a pair $\beta^{>} = ({\mathcal{N}^{>}_{\gg}}^R, \rho^{>})$ and a sequence $\zeta^{>} = M^{>}_0[t^{>}_0\rangle M^{>}_1[t^{>}_1\rangle \cdots$ iteratively. We lift the function $\rho^{>}$ to sets X by $\rho^{>}(X) = \{\rho^{>}(x) \mid x \in X\}$.

(i) The marking $M^{>}_0$ corresponds to the initial marking of $\mathcal{N}^{>}_{\gg}$, i.e., $\rho^{>}(M^{>}_0) = In^{>}$.

(ii) Every $(n+1)$st transition is the next transition of ζ. Thus, $t^{>}_{j\cdot(n+1)} = t_j$ with $\rho^{>}(t_{j\cdot(n+1)}) = \rho(t_j)$ for every $j \in \mathbb{N}$ (as long as t_j is existent in ζ).

(iii) Every other transition $t = t^{>}_{j\cdot(n+1)+i}$, for the existing t_j and $i \in \{1, \ldots, n\}$, is a fresh transition. The mapping of t is dependent on the previous original transition

$t_o = \rho(t^{>}_{j\cdot(n+1)}) \in \mathscr{T}_o^{>} = \mathscr{T}$. In the case of $\beta, \sigma_R(\zeta) \models \mathbb{A}\,\psi_i$, no chain has to be considered, thus t is mapped to the corresponding skipping transition (i.e., $\rho^{>}(t) = \nu(t_o)_{\Rrightarrow_i}$). In the case of $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\psi_i$ the mapping is done iteratively according to ξ^i. Before the transition starts, we again map t to the corresponding skipping transition. The first occurrence of $(\rhd, \rho(p_0^i)) \in \Upsilon(t_o)$ yields $\rho^{>}(t) = \nu(t_o)_{\nu(\rho(p_0^i))_i}$. We remember this position by $\Theta_i(\zeta, 0) = j(n+1) + i + 1$. Then, whenever the next transition t_o with $(\rho(p_k^i), \rho(p_{k+1}^i)) \in \Upsilon(t_o)$ occurs, $\rho^{>}(t) = \nu(t_o)_{(\nu(p_k^i),\nu(p_{k+1}^i))_i}$ is used. This position is remembered with $\Theta_i(\zeta, k) = j(n+1) + i + 1$. In all other cases, $\rho^{>}(t) = \nu(t_o)_{\Rrightarrow_i}$ holds again.

(iv) Each marking $M_k^{>}$ of $\zeta^{>}$ for $k \in \mathbb{N} \setminus \{0\}$ is corresponding to a marking $M_k'^{>} = \rho(M_{k-1}^{>}) \setminus pre^{\mathscr{N}_{\gg}^{>}}(\rho(t_{k-1}^{>})) \cup post^{\mathscr{N}_{\gg}^{>}}(\rho(t_{k-1}^{>}))$ (as long as $t_{k-1}^{>}$ has been created), i.e., $\rho^{>}(M_k^{>}) = M_k'^{>}$.

The net $\mathscr{N}_{\gg}^{>R}$ is created iteratively out of the places of the markings $M_j^{>}$ and the transitions $t_j^{>}$ of $\zeta^{>}$ which are connected according to the pre- and postsets of $\rho^{>}(t_j^{>})$. We denote this construction with $\Theta(\zeta) = \zeta^{>}$. ◀

Note that the constructed pair $\beta^{>} = (\mathscr{N}_{\gg}^{>R}, \rho^{>})$ is indeed a run of $\mathscr{N}_{\gg}^{>}$ and the constructed sequence $\zeta^{>}$ is a firing sequence covering $\beta^{>}$.

The firing sequence serving as counterexample for $\mathscr{N}_T \models \varphi$ is gained from a covering $\zeta^{>}$ of a run of $\mathscr{N}_{\gg}^{>}$ by projecting onto the elements of $\mathscr{N}_T$. The flow chains serving as counterexample for $\beta, \sigma_R(\zeta) \models \mathbb{A}\,\psi_i$ are created by iteratively concatenating the corresponding places and the transitions different to the skipping transition of each subnet.

▶ **Definition 41 (CEX: From PN to PNwT).** Let $\beta^{>} = (\mathscr{N}_{\gg}^{>R}, \rho^{>})$ be a run of $\mathscr{N}_{\gg}^{>}$ and $\zeta^{>} = M_0^{>}[t_0^{>}\rangle M_1^{>}[t_1^{>}\rangle \cdots$ be a covering firing sequence.

(i) We create a sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ by projecting onto the elements of $\mathscr{N}_T$, i.e., $M_j = \{p \in M^{>}_{j\cdot(n+1)} \mid \rho^{>}(p) \in \mathscr{P}\}$ and $t_j = t^{>}_{j\cdot(n+1)}$ for all $j \in \mathbb{N}$ (as long as $M^{>}_{j\cdot(n+1)}$ and $t^{>}_{j\cdot(n+1)}$ are existent in $\zeta^{>}$).

(ii) The net $\mathscr{N}_T^R$ is analogously created as in Definition 40 and ρ is defined by $\rho(p) = \rho^{>}(p)$ for all $p \in M_j$ and $\rho(t_j) = \rho^{>}(t^{>}_{j\cdot(n+1)})$ for all $j \in \mathbb{N}$ (as long as the elements exist).

(iii) The flow chain $\xi^i = t_\iota^i, p_0^i, t_0^i, p_1^i, t_1^i, \ldots$ for a subnet $i \in \{1, \ldots, n\}$ is only created when there is any transition $t_j^{>}$ in $\zeta^{>}$ with $\rho^{>}(t_j^{>}) = t^{>}$ for any $t^{>} \in \mathscr{T}_i^{>}$ with $\nu(t^{>}) \neq \nu(\lambda(t^{>}))_{\Rrightarrow_i}$. In this case, we iteratively collect the corresponding transitions and their, to the transit belonging, corresponding places of the pre- and postset. This means, if $\nu(t^{>}) = \nu(\lambda(t^{>}))_{\nu(q)_i}$ for any place $q \in \mathscr{P}$, we start the chain with t_ι^i, p_0^i such that $t_\iota^i = t^{>}_{j-i} \in \mathscr{T}_o$ (the corresponding transition of the original part of the net) and $p_0^i \in post^{\mathscr{N}_{\gg}^{>R}}(t^{>}_{j-1}) \cap \{p \in \mathscr{P}^{>R} \mid \lambda(p) = q\}$ (the single successor place in which the chain is started). If $\nu(t^{>}) = \nu(\lambda(t^{>}))_{(\nu(p),\nu(q))_i}$ for any $p, q \in \mathscr{P}$, then add t_k^i, p_k^i to the sequence exactly as in the previous case. For each

adding step k, we remember the position in $\zeta^>$ by $\Theta_i^>(\zeta^>, k) = j + 1$. We denote the construction by $\Theta_{\xi^i}^>(\zeta^>) = \xi^i$.

For this construction we denote $\Theta_R^>(\zeta^>) = (\mathcal{N}_T^R, \rho)$ and $\Theta^>(\zeta^>) = \zeta$. ◀

Note that the constructed pair $\beta = (\mathcal{N}_T^R, \rho)$ is indeed a run of $\mathcal{N}_T$, the constructed sequences ξ^j are flow chains of β, and the constructed sequence ζ is a covering firing sequence of β.

We prove Lemma 12 via a nested structural induction over the Flow-LTL formula φ. Therefore, we use the LTL subformulas of the run part and the flow subformulas of φ as induction base for the outer induction, and prove each part separately by structural induction. We again consider both parts of the correctness, the *soundness (s)* and the *completeness (c)*, separately.

▶ **Lemma 17 (LTL Part).** *Given an LTL formula $\psi^>$ created by Definition 39 without condition **(nSub)** from the LTL part ψ (not within the scope of a flow operator $\mathbb{A}$) of a Flow-LTL formula φ.*

(s) Given a run β and a covering firing sequence ζ, with $\sigma_R(\Theta(\zeta)) \models_{\text{LTL}} \square\diamond \overrightarrow{o}$, then $\sigma_R(\zeta) \not\models_{\text{LTL}} \psi \implies \sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \psi^>$ holds.

(c) Given a firing sequence $\zeta^>$ of a run of the net $\mathcal{N}_{\gg}^>$ with $\sigma_R(\zeta^>) \models_{\text{LTL}} \square\diamond \overrightarrow{o}$, then $\sigma_R(\zeta^>) \not\models_{\text{LTL}} \psi^> \implies \sigma_R(\Theta^>(\zeta^>)) \not\models_{\text{LTL}} \psi$ holds. ◀

Proof (via structural induction over ψ). For proving *(s)* and *(c)*, we show the correctness for all corresponding subtraces, i.e., we prove the *soundness* property

$$\forall i \in \mathbb{N} : \sigma_R(\zeta)^i \not\models_{\text{LTL}} \psi \implies \forall j \in \{0, \dots, b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi^>,$$

and the *completeness* property

$$\forall i \in \mathbb{N} : \sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \psi^> \implies \sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1) \rceil} \not\models_{\text{LTL}} \psi$$

with $b(i) = 0$ for $i = 0$ and $b(i) = n$ otherwise. Since at the end this holds for all $i \in \mathbb{N}$, the statements *(s)* and *(c)* of the lemma directly follow with $i = 0$. Let $i \in \mathbb{N}$.

(IB) **Case** $\psi = p \in \mathscr{P}$. Definition 39 yields $\psi^> = p$.
Regarding soundness: Let $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ and $\Theta(\zeta) = M_0^>[t_0^>\rangle M_1^>[t_1^>\rangle \cdots$ with the corresponding mapping functions ρ and $\rho^>$, respectively. The premise of the statement yields $p \notin \sigma_R(\zeta)(i)$. If $i = 0$, condition **(in)** of Definition 38 together with condition (i) of Definition 40 ensures that $\rho(M_0) \subset \rho^>(M_0^>)$ and that there cannot be any other $p' \in M_0^>$ with $\rho^>(p') \in \mathscr{P}$. Hence, $p \notin \sigma_R(\Theta(\zeta))(0)$. If $i > 0$, condition (iv) of Definition 40 yields that all other markings $M_k^>$ are mapped to markings which are created by the firing of transitions of $\mathcal{N}_{\gg}^>$. Definition 38 ensures that tokens residing on places $p \in \mathscr{P}_o^> \cap \mathscr{P}$ of the original part of the net are not moved by transitions of the subnet. Hence, due to $p \notin \sigma_R(\zeta)(i)$, we know that p cannot get occupied while firing any transition of the

subnet. With condition (ii) and (iii) of Definition 40 we know $p \notin \sigma_R(\Theta(\zeta))(i(n+1)-j)$ for all $j \in \{0,\dots,n\}$.

Regarding completeness: Given $\zeta^> = M_0^>[t_0^>\rangle M_1^>[t_1^>\rangle \cdots$ and its transformation $\Theta^>(\zeta^>) = M_0[t_0\rangle M_1[t_1\rangle \cdots$ with the corresponding mapping functions ρ and $\rho^>$, respectively. The premise yields $p \notin \sigma_R(\zeta^>)(i)$. If $i = 0$, condition (i) of Definition 41 yields that $M_0 = \{p^> \in M_0^> \mid \rho^>(p^>) \in \mathscr{P}\}$ and since $p \in \mathscr{P}$ and condition (ii) of the definition ensures that both mapping functions coincide, $p \notin \sigma_R(\Theta^>(\zeta^>))(0)$. If $i > 0$, Definition 38 and the precondition $\sigma_R(\zeta^>) \models_{\text{LTL}} \square\diamond \overrightarrow{o}$, ensures that the firing sequence $\zeta^>$ of $\mathscr{N}_\gg^>$ repeatedly has an original transition $t_o \in \mathscr{T}_o^>$ and then sequentially one transition for each subnet, i.e., n transitions. Thus, with $\lceil i/(n+1)\rceil$ we obtain the index of the corresponding block. This means for $i = 0$ we have the block only containing the initial marking and the outgoing transition, and for all other blocks we have first the marking and outgoing transition of the first subnet $\mathscr{N}_1^>$ and last the next marking where $\mathscr{N}_O^>$ is again activated and the corresponding outgoing transition $t \in \mathscr{T}$ (or no transition at all). Since Definition 38 ensures that no transition of any subnet moves a token of the original net (apart from $\overrightarrow{o}$) the places $p \in \mathscr{P}$ of the markings of each block stay the same. Since Definition 41 creates the markings for $\Theta^>(\zeta^>)$ exactly by choosing those corresponding places and maps them accordingly, $p \notin \sigma_R(\Theta^>(\zeta^>))(\lceil i/(n+1)\rceil)$.

Case $\psi = t \in \mathscr{T}$. Definition 39 yields $\psi^> = \bigvee_{t' \in \mathscr{T}^> \setminus \mathscr{T}} t' \,\mathcal{U}\, t$.

Regarding soundness: The premise yields $t \notin \sigma_R(\zeta)(i)$. Since condition (ii) of Definition 40 copies every $(n+1)$st transition we know that $t \notin \sigma_R(\Theta(\zeta))(i(n+1))$. Furthermore, with Definition 32, $\sigma_R(\Theta(\zeta))(i(n+1)) \cap (\mathscr{T}^> \setminus \mathscr{T}) = \emptyset$ holds. Hence, $\sigma_R(\Theta(\zeta))^{i(n+1)} \not\models_{\text{LTL}} \psi^>$. For $i = 0$ this already yields the conclusion. In the case that $i > 0$, condition (iii) of Definition 40 ensures that for all $j \in \{0,\dots,n\}$ a transition is added which maps to a transition of the subnet $\mathscr{T}^> \setminus \mathscr{T}$. Hence, $\sigma_R(\Theta(\zeta))(i(n+1)-j) \cap \mathscr{T} = \emptyset$ for all $j \in \{0,\dots,n\}$, and so $\sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi^>$.

Regarding completeness: The premise yields $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \bigvee_{t' \in \mathscr{T}^> \setminus \mathscr{T}} t' \,\mathcal{U}\, t$. By Definition 38 and precondition $\sigma_R(\zeta^>) \models_{\text{LTL}} \square\diamond \overrightarrow{o}$, we know that every trace of $\mathscr{N}_\gg^>$ must contain at position $i(n+1)$ a transition $t_o \in \mathscr{T}_o^> = \mathscr{T}$ or no transition at all. Thus, if $i = 0$, we know from the premise and Definition 32, $t \notin \sigma_R(\zeta^>)(0)$ holds. Definition 41 keeps for all $j(n+1)$ positions the transitions and the mapping for $\Theta^>(\zeta^>)$. Hence, $t \notin \sigma_R(\Theta^>(\zeta^>))(0)$. If $i > 0$, we again consider the block with index $\lceil i/(n+1)\rceil$ of $\sigma_R(\zeta^>)$. Due to Definition 38 we know that the block first contains n transitions $t \in \mathscr{T}^> \setminus \mathscr{T}$ of the subnets for the flow formulas and then one transition $t \in \mathscr{T}$ for the original part of the net (or no transition at all). Thus, due to the premise $t \notin \sigma_R(\zeta^>)(\lceil i/(n+1)\rceil \cdot (n+1))$ holds. Again, Definition 41 yields the conclusion, i.e., $t \notin \sigma_R(\Theta^>(\zeta^>))(\lceil i/(n+1)\rceil)$.

(IS) Let $\psi_1^>$ and $\psi_2^>$ be LTL formulas created from the LTL parts ψ_1 and ψ_2 of Flow-LTL formulas by Definition 39 without **(nSub)**.

Case $\psi = \neg\psi_1$. Since Definition 39 does not concern the negation, $\psi^> = \neg\psi_1^>$.

Regarding soundness: The premise yields $\sigma_R(\zeta)^i \models_{\text{LTL}} \psi_1$ and because $\Theta^>(\Theta(\zeta)) = \zeta$, this can also be stated as $\sigma_R(\Theta^>(\Theta(\zeta)))^i \models_{\text{LTL}} \psi_1$. Consider the contraposition of the *completeness* property: $\sigma_R(\Theta^>(\zeta^>))^{\lceil i'/(n+1)\rceil} \models_{\text{LTL}} \psi \implies \sigma_R(\zeta^>)^{i'} \models_{\text{LTL}} \psi^>$. For all

$j \in \{0,\dots,b(i)\}$, we have $\lceil (i(n+1)-j)/(n+1)\rceil = \lceil (i - j/(n+1)\rceil = i$. Thus, the induction hypothesis yields $\sigma_R(\Theta^>(\zeta^>))^{i(n+1)-j} \models_{\text{LTL}} \psi_1^>$. This means, the conclusion $\forall j \in \{0,\dots,b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \neg\psi_1^> = \psi^>$ holds.

Regarding completeness: The premise yields $\sigma_R(\zeta^>)^i \models_{\text{LTL}} \psi_1^>$. Consider the contraposition of the *soundness* property: $\exists j' \in \{0,\dots,b(i)\} : \sigma_R(\Theta(\zeta))^{i'(n+1)-j'} \models_{\text{LTL}} \psi^> \implies \sigma_R(\zeta)^{i'} \models_{\text{LTL}} \psi$. There is a $j \in \{0,\dots,b(i)\}$ such that the sum with i is the next multiple of $n+1$, i.e., $i+j = i'(n+1)$ for some $i' \in \mathbb{N}$. Thus, with $i = i'(n+1)-j$, the induction hypothesis yields $\sigma_R(\Theta^>(\zeta^>))^{i'} \models_{\text{LTL}} \psi_1$ because $\Theta(\Theta^>(\zeta^>)) = \zeta^>$. Since $i' = (i+j)/(n+1)$, we have $\sigma_R(\Theta^>(\zeta^>))^{(i+j)/(n+1)} \models_{\text{LTL}} \psi_1$ and so $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \not\models_{\text{LTL}} \psi$.

Case $\psi = \psi_1 \wedge \psi_2$. Since Definition 39 does not concern the conjunction, $\psi^> = \psi_1^> \wedge \psi_2^>$.
Regarding soundness: The premise and the induction hypothesis for *soundness* yield that either $\forall j \in \{0,\dots,b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi_1^>$ or $\forall j \in \{0,\dots,b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi_2^>$ holds. The universal quantifier can be moved to the outer level. Hence, $\forall j \in \{0,\dots,b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi_1^> \wedge \psi_2^>$.

Regarding completeness: The premise states that $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \psi_1^>$ or $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \psi_1^>$ holds. Due to the induction hypothesis we know that $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \not\models_{\text{LTL}} \psi_1$ or $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \not\models_{\text{LTL}} \psi_2$ holds. Hence, $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \not\models_{\text{LTL}} \psi_1 \wedge \psi_2$.

Case $\psi = \bigcirc\psi_1$. Definition 39 yields $\psi^> = (\bigvee_{t' \in \mathscr{T}^> \setminus \mathscr{T}} t')\,\mathscr{U}(\bigvee_{t\in\mathscr{T}} t \wedge \bigcirc\psi_1^>) \vee (\Box(\neg\bigvee_{t\in\mathscr{T}} t) \wedge \psi_1^>)$.
Regarding soundness: The premise yields $\sigma_R(\zeta)^i \not\models_{\text{LTL}} \bigcirc\psi_1$, i.e., $\sigma_R(\zeta)^{i+1} \not\models_{\text{LTL}} \psi_1$. The induction hypothesis ensures that the statement $(\star)$ $\sigma_R(\Theta(\zeta))^{(i+1)(n+1)-j} \not\models_{\text{LTL}} \psi_1^>$ holds for all $j \in \{0,\dots,b(i+1)\}$. We show the conclusion by contradiction. Assume: $\sigma_R(\Theta(\zeta))^{i(n+1)-j'} \models_{\text{LTL}} \psi^>$ for some $j' \in \{0,\dots,b(i)\}$. Thus, the first or the second disjunct is satisfied. In the case that the first disjunct is satisfied, condition (iii) of Definition 40 ensures that if $j' > 0$, $\sigma_R(\Theta(\zeta))^{i(n+1)-j'} \cap \mathscr{T} = \emptyset$ holds, because at most transitions of the subnets for the flow part could appear. Since these steps are skipped by the until operator we know $\sigma_R(\Theta(\zeta))^{i(n+1)} \models_{\text{LTL}} \bigvee_{t\in\mathscr{T}} t \wedge \bigcirc\psi_1^>$ has to hold because at position $i(n+1)$ condition (ii) of Definition 40 ensures that at most $t \in \mathscr{T}$ can occur. This is directly given if $j' = 0$. Hence, $\sigma_R(\Theta(\zeta))^{i(n+1)+1} \models_{\text{LTL}} \psi_1^>$ which is a contradiction to $(\star)$ for $j = n$. If the second disjunct is satisfied, we know by $\Box(\neg\bigvee_{t\in\mathscr{T}} t)$ that never an original transition will occur in the future. Because of Definition 40 and Definition 32 this is only possible if ζ is finite, and therewith also $\Theta(\zeta)$. Thus, from $i(n+1)-j'$ on, the trace of $\Theta(\zeta)$ is stuttering and so all atomic propositions stay the same in the future, i.e., $\sigma_R(\Theta(\zeta))^{(i(n+1)-j')} = \sigma_R(\Theta(\zeta))^{(i(n+1)-j')+x}$ for all $x \in \mathbb{N}$. This is a *contradiction* to $(\star)$, since $\psi_1^>$ currently holds and therewith for the whole future.

Regarding completeness: The premise states $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} (\bigvee_{t' \in \mathscr{T}^> \setminus \mathscr{T}} t')\,\mathscr{U}(\bigvee_{t\in\mathscr{T}} t \wedge \bigcirc\psi_1^>) \vee (\Box(\neg\bigvee_{t\in\mathscr{T}} t) \wedge \psi_1^>)$. Thus, (a) $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} (\bigvee_{t' \in \mathscr{T}^> \setminus \mathscr{T}} t')\,\mathscr{U}(\bigvee_{t\in\mathscr{T}} t \wedge \bigcirc\psi_1^>)$ *and* (b) $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \Box(\neg\bigvee_{t\in\mathscr{T}} t) \wedge \psi_1^>$ holds. We show $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \not\models_{\text{LTL}} \psi$ holds by contradiction. Assume: $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil} \models_{\text{LTL}} \psi = \bigcirc\psi_1$ holds, i.e., $\sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1)\rceil+1} \models_{\text{LTL}} \psi_1$. The contraposition of the induction hypothesis yields $(\star)$ $\sigma_R(\zeta^>)^{i+n+1} \models_{\text{LTL}} \psi_1^>$ because $\lceil i/(n+1)\rceil+1 = \lceil (i+n+1)/(n+1)\rceil$. Case $\sigma_R(\zeta^>)^i(0) \subseteq \mathscr{P}$: Due to Definition 32 this case is only possible when $\zeta^>$ is finite. Thus, we are in

the stuttering mode and no step of $\sigma_R(\zeta^>)^i$ will ever contain a transition again. So especially, $\sigma_R(\zeta^>)^i \models_{\text{LTL}} \Box(\neg \bigvee_{t \in \mathcal{T}} t)$. Condition (b) yields that $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \psi_1^>$ and due to the stuttering $\sigma_R(\zeta^>)^{i+n+1} \not\models_{\text{LTL}} \psi_1^>$. This is a contradiction to $(\star)$. Case $\sigma_R(\zeta^>)^i(0) \cap \mathcal{T} \neq \emptyset$: Due to Definition 32 $\sigma_R(\zeta^>)^i(0) \cap \mathcal{T}^> \setminus \mathcal{T} = \emptyset$ we know from (a) that $\sigma_R(\zeta^>)^i \not\models_{\text{LTL}} \bigcirc \psi_1^>$, i.e., $\sigma_R(\zeta^>)^{i+1} \not\models_{\text{LTL}} \psi_1^>$. Definition 38 ensures that after an original transition $t \in \mathcal{T}$ is fired, first n transitions $t^> \in \mathcal{T}^> \setminus \mathcal{T}$ of the subnets concerning the flow subformulas are fired. Due to the precondition $\sigma_R(\zeta^>) \models_{\text{LTL}} \Box\Diamond \overrightarrow{o}$, all of these transitions are actually taken into account. Definition 38 ensures that those transitions do not change the tokens in places of the original part (apart from $\overrightarrow{o}$). Hence, $\sigma_R(\zeta^>)^{i+1+n} \not\models_{\text{LTL}} \psi_1^>$. This is a *contradiction* to $(\star)$. Case $\sigma_R(\zeta^>)^i(0) \cap (\mathcal{T}^> \setminus \mathcal{T}) \neq \emptyset$: If there is a transition $t \in \mathcal{T}$ at some point in $\sigma_R(\zeta^>)^i$, say after j steps there is the first, i.e., $\sigma_R(\zeta^>)^{i+j} \cap \mathcal{T} \neq \emptyset$, then Definition 38 yields $\sigma_R(\zeta^>)^i(k) \cap (\mathcal{T}^> \setminus \mathcal{T}) \neq \emptyset$ for all $k \in \{0, \ldots, j-1\}$, i.e., $\sigma_R(\zeta^>)^{i+k} \models_{\text{LTL}} \bigvee_{t' \in \mathcal{T}^> \setminus \mathcal{T}} t'$. Thus, we can proceed as in the second case. If $\sigma_R(\zeta^>)^i$ never contains such a transition $t \in \mathcal{T}$ again, this means we can fire at most n transitions $t^> \in (\mathcal{T}^> \setminus \mathcal{T})$ due to Definition 38 and then have to stutter, then we can proceed as in the first case. Due to Definition 32 these are all possible cases.

Case $\psi = \psi_1 \,\mathcal{U}\, \psi_2$. Since Definition 39 does not concern the until operator, $\psi^> = \psi_1^> \,\mathcal{U}\, \psi_2^>$.
Regarding soundness: The premise yields $\forall k \geq 0 : \sigma_R(\zeta)^{i+k} \not\models_{\text{LTL}} \psi_2 \vee \exists 0 \leq l < k : \sigma_R(\zeta)^{i+l} \not\models_{\text{LTL}} \psi_1$. The induction hypothesis applied for all these ks and ls yields: $\forall k \geq 0 : \forall j \in \{0, \ldots, b(i+k)\} : \sigma_R(\Theta(\zeta))^{(i+k)(n+1)-j} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \forall j \in \{0, \ldots, b(i+l)\} : \sigma_R(\Theta(\zeta))^{(i+l)(n+1)-j} \not\models_{\text{LTL}} \psi_1^>$. By rewriting the indices $(i+k)(n+1)-j = i(n+1)-j+k(n+1)$ and $(i+l)(n+1)-j = i(n+1)-j+l(n+1)$ we can see that k and l are used to jump in $n+1$ steps. The js ensure that the formulas are also not satisfied for all $n+1$ steps in between. Since this holds for all $k > 0$, we can shift the indices while moving the universal quantifier of the j to the outside: $\forall j \in \{0, \ldots, b(i)\} : \forall k \geq 0 : \sigma_R(\Theta(\zeta))^{i(n+1)+k-j} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \sigma_R(\Theta(\zeta))^{i(n+1)+l-j} \not\models_{\text{LTL}} \psi_1^>$. This is the semantical definition of the until operator for $\psi_1^>$ and $\psi_2^>$. Hence, $\forall j \in \{0, \ldots, b(i)\} : \sigma_R(\Theta(\zeta))^{i(n+1)-j} \not\models_{\text{LTL}} \psi_1^> \,\mathcal{U}\, \psi_2^>$.
Regarding completeness: The premise yields that $\forall k \geq 0 : \sigma_R(\zeta^>)^{i+k} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \sigma_R(\zeta^>)^{i+l} \not\models_{\text{LTL}} \psi_1^>$. The induction hypothesis for all ks and ls gives $\forall k \geq 0 : \sigma_R(\Theta^>(\zeta^>))^{\lceil (i+k)/(n+1) \rceil} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k : \sigma_R(\Theta^>(\zeta^>))^{\lceil (i+l)/(n+1) \rceil} \not\models_{\text{LTL}} \psi_1^>$. Since this holds for every k and thus, especially for every $k(n+1)$ we know $\forall k' \geq 0 : \sigma_R(\Theta^>(\zeta^>))^{\lceil (i/(n+1)+k' \rceil} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l < k'(n+1) : \sigma_R(\Theta^>(\zeta^>))^{\lceil (1/(n+1)+l/(n+1)) \rceil} \not\models_{\text{LTL}} \psi_1^>$ holds. With $l' = l/(n+1)$ we have $\forall k' \geq 0 : \sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1) \rceil + k'} \not\models_{\text{LTL}} \psi_2^> \vee \exists 0 \leq l' < k') : \sigma_R(\Theta^>(\zeta^>))^{\lceil i/(n+1) \rceil + l'} \not\models_{\text{LTL}} \psi_1^>$, which is the semantical definition of the conclusion. □

So far we proved the correctness of the LTL formulas in the run part of the formula transformation. Now we consider a single flow part and proceed quite analogously for the *soundness (s)* and *completeness (c)* of the construction.

▶ **Lemma 18 (Flow Part).** *Given an LTL formula $\psi^{>}$ created by Definition 39 without condition **(nSub)** from a flow formula $\mathbb{A}\,\psi_i$ of a Flow-LTL formula φ.*

(s) Given a run β and a covering firing sequence ζ, with $\sigma_R(\Theta(\zeta)) \models_{\text{LTL}} \square\diamond \overrightarrow{\sigma}$, then $\beta, \sigma_R(\zeta) \not\models \mathbb{A}\,\psi_i \implies \sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \psi^{>}$ holds.

(c) Given a firing sequence $\zeta^{>}$ of a run of the Petri net $\mathcal{N}_{\gg}^{>}$ with $\sigma_R(\zeta^{>}) \models_{\text{LTL}} \square\diamond \overrightarrow{\sigma}$, then $\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} \psi^{>} \implies \Theta_R^{>}(\zeta^{>}), \sigma_R(\Theta^{>}(\zeta^{>})) \not\models \mathbb{A}\,\psi_i$ holds. ◀

Proof (via structural induction over ψ_i). Definition 39 yields $\psi^{>} = [\iota]_i \mathscr{W} (\neg[\iota]_i \wedge \psi_i^{>})$ where $\psi_i^{>}$ is created from ψ_i by the constraints **(pF)**, **(tF)**, and **(nF)**. Since the operator $\mathscr{W}$ is an abbreviation, we know $\psi^{>} = ([\iota]_i \,\mathscr{U} (\neg[\iota]_i \wedge \psi_i^{>})) \vee \square[\iota]_i$.

Regarding soundness. The premise yields that there is a flow chain $\xi^i = t_\iota^i, p_0^i, t_0^i, p_1^i, t_1^i, \ldots$ such that $\sigma_F(\xi^i) \not\models_{\text{LTL}} \psi_i$. Since there is a chain, condition (iii) of Definition 40 yields the existence of a transition t in $\Theta(\zeta)$ at position $\Theta_i(\zeta, 0) - 1$ starting the chain, i.e., $\rho^{>}(t) = \nu(t_o)_{\nu(\rho(p_0^i))_i}$ for an original transition $t_o \in \mathscr{T}_o^{>} = \mathscr{T}$ with $(\triangleright, \rho(p_0^i)) \in \Upsilon(t_o)$. Condition **(s3)** of Definition 38 ensures that at position $\Theta_i(\zeta, 0)$ the place $[\iota]_i$ is unoccupied because the transition starting the chain just fired. Hence, $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \square[\iota]_i$. Furthermore, from condition **(in)** of Definition 38 follows that $[\iota]_i$ is initially marked. Definition 40 states that t is the first occurrence of such kind and Definition 38 ensures that no other kind of transition takes a token from $[\iota]_i$. Thus, $[\iota]_i$ is satisfied until position $\Theta_i(\zeta, 0)$. Hence, we have to show that in such situations $\sigma_R(\Theta(\zeta))^{\Theta_i(\zeta,0)} \not\models_{\text{LTL}} \psi_i^{>}$ holds. This is done by a structural induction together with the *(c)* part.

Regarding completeness. The premise states that $\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} ([\iota]_i \,\mathscr{U} (\neg[\iota]_i \wedge \psi_i^{>})) \vee \square[\iota]_i$ holds. Since $\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} \square[\iota]_i$, Definition 38 yields the existence of a transition t in $\zeta^{>}$ with $\rho^{>}(t) = \nu(t_o)_{\nu(q)_i}$ for an original transition $t_o \in \mathscr{T}^{>} = \mathscr{T}$ and a place $q \in \mathscr{P}$ with $(\triangleright, q) \in \Upsilon(t_o)$. Thus, condition (iii) of Definition 41 yields the existence of a flow chain $\Theta_{\xi^i}^{>}(\zeta^{>})$ and that the first place of the chain corresponds to index $\Theta_i^{>}(\zeta^{>}, 0)$. Since we know that until this position $[\iota]_i$ is satisfied, the premise yields $\sigma_R(\zeta^{>})^{\Theta_i^{>}(\zeta^{>},0)} \not\models_{\text{LTL}} \psi_i^{>}$. Hence, we have to show that in those situations $\sigma_R(\Theta_{\xi^i}^{>}(\zeta^{>})) \not\models_{\text{LTL}} \psi_i$ holds.

We show these two cases via a common structural induction over ψ_i. Let $n_0 \in \mathbb{N}$ be the last index such that $\Theta_i(\zeta, n_0)$ of Definition 40 is defined or $n_0 = \infty$ if all are defined. We define with $\Delta_0 = \{\Theta_i(\zeta, 0)\}$, $\Delta_k = \{\Theta_i(\zeta, k-1) + 1, \ldots, \Theta_i(\zeta, k)\}$ for $0 < k \leq n_0$, and $\Delta_{k'} = \{\Theta_i(\zeta, n_0)\}$ otherwise, the sets of indices of the firing sequence ζ between the $(k-1)$st and the kth step of the corresponding data flow chain. Similarly, let $n_0^{>} \in \mathbb{N}$ be the last index such that $\Theta_i^{>}(\zeta^{>}, n_0^{>})$ is defined or $n_0^{>} = \infty$ if all are defined. We define a function $\Delta_i^{>}$ with $\Delta_i^{>}(\Theta_i^{>}(\zeta^{>}, 0)) = 0$, for all steps $k > 0$ of the flow chain $\Theta_{\xi^i}^{>}(\zeta^{>})$ we have $\Delta_i^{>}(x) = k$ for all $x \in \{\Theta_i^{>}(\zeta^{>}, k-1) + 1, \ldots, \Theta_i^{>}(\zeta^{>}, k)\}$, and, if the flow chain $\Theta_{\xi^i}^{>}(\zeta^{>})$ is finite, $\Delta_i^{>}(x') = \{n_0^{>}\}$ for all $x' > \Theta_i^{>}(\zeta^{>}, n_0^{>})$. With this function we map each index of the firing sequence $\zeta^{>}$ to the corresponding step of the created flow chain, i.e., all indices are mapped to the element of the trace of the flow chain until the next transition of the subnet transiting the chain has fired.

We show the open cases of property *(s)* and *(c)* by proving the *soundness* property, i.e.,

$$\forall k \in \mathbb{N} : \sigma_F(\xi^i)^k \not\models_{\text{LTL}} \psi_i \implies \forall j \in \Delta_k : \sigma_R(\Theta(\zeta))^j \not\models_{\text{LTL}} \psi_i^>$$

and the *completeness* property, i.e.,

$$\forall k \geq \Theta_i^>(\zeta^>, 0) : \sigma_R(\zeta^>)^k \not\models_{\text{LTL}} \psi_i^> \implies \sigma_F(\Theta_{\xi^i}^>(\zeta^>))^{\Delta_i^>(k)} \not\models_{\text{LTL}} \psi_i.$$

As for the LTL part, the open cases for *(s)* and *(c)* follow for the smallest k, i.e., for $k = 0$ we have $\sigma_F(\xi^i) \not\models_{\text{LTL}} \psi_i \implies \sigma_R(\Theta(\zeta))^{\Theta_i(\zeta,0)} \not\models_{\text{LTL}} \psi_i^>$ and for $k = \Theta_i^>(\zeta^>, 0)$ we have $\sigma_R(\zeta^>)^{\Theta_i^>(\zeta^>,0)} \not\models_{\text{LTL}} \psi_i^> \implies \sigma_F(\Theta_{\xi^i}^>(\zeta^>)) \not\models_{\text{LTL}} \psi_i$, because $\Delta_i^>(\Theta_i^>(\zeta^>, 0)) = 0$. Even though these two properties look different to the ones of the LTL part in the proof of Lemma 17, they follow the exact same structure. It is easy to generate the corresponding sets Δ_k and the function $\Delta_i^>$ for the LTL part because the relevant bounds are just multiples of $n + 1$. For a lighter weight access we decided to use the easier notation in the proof of Lemma 17, but none the less the arguments of the structural induction can be completely adopted for the flow part.

If we consider the overview of the replacements for the next operator and atomic propositions $t \in \mathcal{T}$ in Tab. 6.2, we see that the replacements mainly only differ in the related and unrelated transitions. That for the replacement of a transition in the flow part any of the corresponding transitions extending the flow chain suffices for the second argument of the until operator, is due to Definition 38 introducing one transition for each transit, but does not complicate the arguments. For the LTL part the unrelated transitions are $\mathcal{T}^> \setminus \mathcal{T}$ and the related $\mathcal{T}$. Thus, due to Definition 38 we could use the blocks of length $n + 1$ to find the next related situation after skipping the unrelated ones for the LTL part of the formula. This is exactly the same for the flow part. The related transitions are those of the subnet transiting the data flow and the unrelated are all others. Constraints **(tF)** and **(nF)** of Definition 39 skip those exactly as for the LTL part. The difference in this case is that not after exactly $n + 1$ steps the entry concern the current part of the formula but there could be more rounds not concerning the considered flow chain. With the help of the function Θ_i of Definition 40 and $\Theta_i^>$ of Definition 41 we can identify the relevant blocks as for the LTL part. Since due to Definition 38 also no token (apart from the activation token) of a subnet $\mathcal{N}_i^>$ is moved by any unrelated transition and the skipping transitions can only be used when no flow can be extended, the structural induction for the flow part of the formula is analog to the LTL part of the formula. □

With these two lemmas showing the soundness and completeness of the counterexample constructions for the LTL subformulas of the run part and the flow subformulas of an Flow-LTL formula, we can now straightforwardly show that for every counterexample for the satisfaction of a Flow-LTL formula φ the constructed counterexample indeed does not satisfy the constructed formula $\varphi_{\gg}^>$.

▶ **Lemma 19 (Soundness).** *Given a run β, a covering firing sequence ζ, and an LTL formula $\varphi_{\gg}^>$ created by Definition 39 from a Flow-LTL formula φ. Then:*

$$\beta, \sigma_R(\zeta) \not\models \varphi \implies \sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_{\gg}^>$$

holds. ◀

Proof (via structural induction over φ). Definition 39 yields that every transformed formula $\varphi_{\gg}^{>}$ is of the form $\square\diamond\overrightarrow{o} \to \varphi^{\mathbb{A}}$. Since Definition 40 adds for every existing transition in ζ also the n transitions (one for each subnet) to $\Theta(\zeta)$ (condition (iii)) and by condition **(mSn)** of Definition 38 every transition of the last subnet puts a token onto $\overrightarrow{o}$, the statement $\sigma_R(\Theta(\zeta)) \models_{\text{LTL}} \square\diamond\overrightarrow{o}$ is satisfied. This also holds for finite firing sequences, because Definition 40 ensures that the last added transition is one of the last subnet. The stuttering then yields the satisfaction. Thus, we only have to show that $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi^{\mathbb{A}}$. Let $\varphi^{>}$ be such a subformula $\varphi^{\mathbb{A}}$.

(IB) **Case** $\varphi = \psi$, for a standard LTL formula ψ. Thus, Definition 35 and the premise yield $\sigma_R(\zeta) \not\models_{\text{LTL}} \psi$ and so, Lemma 17 directly yields $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi^{>}$.
Case $\varphi = \mathbb{A}\,\psi$. Lemma 18 proves exactly this case.

(IS) **Case** $\varphi = \varphi_1 \wedge \varphi_2$. Since Definition 39 does not concern the conjunction operator of the run part, there are subformulas $\varphi_1^{>}$ and $\varphi_2^{>}$ with $\varphi^{>} = \varphi_1^{>} \wedge \varphi_2^{>}$, which are created from the corresponding φ_1 and φ_2, respectively. Due to the premise $\beta, \sigma_R(\zeta) \not\models \varphi_1 \wedge \varphi_2$, we know $\beta, \sigma_R(\zeta) \not\models \varphi_1$ or $\beta, \sigma_R(\zeta) \not\models \varphi_2$. The induction hypothesis yields that $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_1^{>}$ or $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_2^{>}$, thus $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_1^{>} \wedge \varphi_2^{>}$.
Case $\varphi = \varphi_1 \vee \varphi_2$. Since Definition 39 also does not concern the disjunction operator of the run part, this case is analog to the previous case.
Case $\varphi = \psi \to \varphi_2$. Since Definition 39 also does not concern the implication operator there are subformulas $\varphi_1^{>}$ and $\varphi_2^{>}$ with $\varphi^{>} = \varphi_1^{>} \to \varphi_2^{>}$, which are created from ψ and φ_2, respectively. The premise $\beta, \sigma_R(\zeta) \not\models \neg\psi \vee \varphi_2$ yields $\beta, \sigma_R(\zeta) \models \psi$, thus $\beta, \sigma_R(\zeta) \not\models \neg\psi$, and $\beta, \sigma_R(\zeta) \models \neg\varphi_2$, thus $\beta, \sigma_R(\zeta) \not\models \varphi_2$. Since $\neg\psi$ is still a standard LTL formula, Lemma 17 yields $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \neg\varphi_1^{>}$ (the Definition 39 does not concern the negation). The induction hypothesis ensures $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_2^{>}$, and so $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_1^{>} \to \varphi_2^{>}$.□

The completeness proof, meaning that also every counterexample for the satisfaction of the constructed formula $\varphi_{\gg}^{>}$ implies that the constructed elements of Definition 41 are indeed a counterexample for the formula φ, can be done completely analogously by using the completeness part of Lemma 17 and Lemma 18.

▶ **Lemma 20 (Completeness).** *Given a firing sequence $\zeta^{>}$ of a run of $\mathcal{N}_{\gg}^{>}$, and an LTL formula $\varphi_{\gg}^{>}$ created by Definition 39 from a Flow-LTL formula φ. Then:*

$$\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} \varphi_{\gg}^{>} \implies \Theta_R^{>}(\zeta^{>}), \sigma_R(\Theta^{>}(\zeta^{>})) \not\models \varphi$$

holds. ◀

Proof (via structural induction over φ). Again, every $\varphi_{\gg}^{>}$ is of the form $\square\diamond\overrightarrow{o} \to \varphi^{\mathbb{A}}$. The premise of the statement therewith yields $\sigma_R(\zeta^{>}) \models_{\text{LTL}} \square\diamond\overrightarrow{o}$ and $\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} \varphi^{\mathbb{A}}$. Every other argument is analog to the arguments of the proof of Lemma 19. □

Finally, putting all this together, we are able to prove Lemma 12 on page 94, stating the correctness of the construction presented in Sec. 6.2.1, i.e., $\mathcal{N}_T \models \varphi$ iff $\mathcal{N}_{\gg}^{>} \models_{\text{LTL}} \varphi_{\gg}^{>}$.

Proof (Lemma 12: Correctness of the Transformation):
Regarding soundness: We show the contraposition of the statement: $\mathcal{N}_T \not\models \varphi \implies \mathcal{N}_{\gg}^{>} \not\models_{\text{LTL}} \varphi_{\gg}^{>}$. Hence, there is a run β of $\mathcal{N}_T$ and a covering firing sequence ζ such that $\beta, \sigma_R(\zeta) \not\models \varphi$. Lemma 19 yields that the firing sequence $\Theta(\zeta)$ fulfills $\sigma_R(\Theta(\zeta)) \not\models_{\text{LTL}} \varphi_{\gg}^{>}$. Thus, there exists a run $\beta^{>}$ (created from $\Theta(\zeta)$ by iteratively adding the places of the markings and the transitions of the firing sequence and connecting them according to their corresponding places and transitions in $\mathcal{N}_{\gg}^{>}$) which is covered by $\Theta(\zeta)$, such that $\beta^{>} \not\models_{\text{LTL}} \varphi_{\gg}^{>}$, and thus $\mathcal{N}_{\gg}^{>} \not\models_{\text{LTL}} \varphi_{\gg}^{>}$.
Regarding completeness. We analogously show the contraposition of the statement: $\mathcal{N}_{\gg}^{>} \not\models_{\text{LTL}} \varphi_{\gg}^{>} \implies \mathcal{N}_T \not\models \varphi$. Hence, there is a run $\beta^{>}$ of $\mathcal{N}_{\gg}^{>}$ and a covering firing sequence $\zeta^{>}$ such that $\sigma_R(\zeta^{>}) \not\models_{\text{LTL}} \varphi_{\gg}^{>}$. Lemma 20 yields that the firing sequence $\Theta^{>}(\zeta^{>})$ fulfills $\Theta_R^{>}(\zeta^{>}), \sigma_R(\Theta^{>}(\zeta^{>})) \not\models \varphi$. Hence, $\mathcal{N}_T \not\models \varphi$. □

6.4.4 Correctness Proof of the Reduction to the Hardware Model Checking Problem

In this section we formally prove the correctness of the reduction method presented in Sec. 6.3 for checking the constructed circuit $\boldsymbol{C}_{\mathcal{N}}$ against the constructed formula ψ' to answer the question whether a safe P/T Petri net with inhibitor arcs $\mathcal{N} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \mathscr{F}_I, In)$ satisfies an LTL formula ψ.

First, we define the Kripke structure $\mathscr{K}_{\boldsymbol{C}}$ corresponding to the constructed circuit $\boldsymbol{C}_{\mathcal{N}}$. For all subsets $P \subseteq \mathscr{P}$ and $T \subseteq \mathscr{T}$, we define with $P_o = \{p_o \in \boldsymbol{O} \mid p \in P\}$ and $T_o = \{t_o \in \boldsymbol{O} \mid t \in T\}$ the respective sets of the output variables.

▶ **Definition 42 (Circuit to Kripke Structure).** The *Kripke structure* $\mathscr{K}_{\boldsymbol{C}_{\mathcal{N}}}$ of the constructed circuit $\boldsymbol{C}_{\mathcal{N}} = (\boldsymbol{\mathcal{I}}, \boldsymbol{O}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{F}})$ is defined by $\mathscr{K}_{\boldsymbol{C}_{\mathcal{N}}} = (A, S, S_0, \ell, \rightarrow)$, with the set of *atoms* $A = \boldsymbol{O}$, the set of *states* $S = 2^{\boldsymbol{\mathcal{I}}} \times 2^{\boldsymbol{\mathcal{L}}} \times 2^{\boldsymbol{O}} \times 2^{\boldsymbol{\mathcal{L}}}$, the set of *initial states* $S_0 = \{(I, \emptyset, In_o, In' \cup \{\mathtt{i}'\}) \mid I \subseteq \boldsymbol{\mathcal{I}}\}$, the *labeling function* $\ell = \{((I, L, O, L'), O) \in S \times 2^{\boldsymbol{O}}\}$, and the *transition relation* $\rightarrow = \{((I_1, L_1, O_1, L_1'), (I_2, L_2, O_2, L_2')) \in S \times S \mid L_1' = L_2 \wedge (I_2, L_2, O_2, L_2') \models \boldsymbol{\mathcal{F}}\}$. ◀

Thus, the states of the Kripke structure correspond to the configurations of the circuit and are labeled with its output. The edges of the Kripke structure correspond to the configuration change of the circuit after a clock pulse. We start with the initialization such that decorated latches contain the initial marking and the initialized flag i. Note that the initial state is skipped by the formula anyhow, so we could have also used any arbitrary set $X \in \boldsymbol{O}$ for In_o.

We show the correctness of the construction of the circuit $\boldsymbol{C}_{\mathcal{N}}$ and the corresponding Kripke structure $\mathscr{K}_{\boldsymbol{C}_{\mathcal{N}}}$ by proving $\mathcal{N} \not\models_{\text{LTL}} \psi$ iff $\mathscr{K}_{\boldsymbol{C}_{\mathcal{N}}} \not\models \psi'$. For this purpose, we mutually transform the respective counterexamples and show their correspondence on the atomic propositions of ψ and ψ', respectively. For that we first introduce a correspondence between the elements of a path of a Kripke structure and a trace of a covering firing sequence of a run of a Petri net.

▶ **Definition 43 (Correspondence of Paths and Firing Sequences).** Given a trace $\sigma_R(\zeta)$ of a firing sequence ζ covering a run $\beta = (\mathcal{N}^R, \rho)$ of a safe P/T Petri net with inhibitor arcs $\mathcal{N}$ and a path $\pi = s_0 s_1 \cdots$ of the corresponding Kripke structure $\mathscr{K}_{C_\mathcal{N}} = (A, S, S_0, \ell, \rightarrow)$. We say an entry $z_i = \sigma_R(\zeta)(i) \in 2^{\mathscr{P} \cup \mathscr{T}}$ of the trace and an element $s_j = (I_j, L_j, O_j, L'_j) \in S$ of the path *coincide*, denoted by $z_i \sim s_j$, iff $z_{i|\mathscr{T}} = \{t \in \mathscr{T} \mid t_o \in O_j \wedge e_o \notin O_j\}$ and $z_{i|\mathscr{P}} = \{p \in \mathscr{P} \mid p_o \in O_j\}$. Where $z_{i|\mathscr{T}}$ and $z_{i|\mathscr{P}}$ are the projections onto the respective sets. ◀

Now we can show Theorem 5 on page 104 stating the correctness of the reduction, i.e., $\mathcal{N} \models_{\mathtt{LTL}} \psi$ iff $C_\mathcal{N} \models \psi'$ and the single-exponential time bound for the algorithm for model checking safe Petri nets with inhibitor arcs against LTL formulas via circuits.

Proof (Theorem 5): The correctness is proven on the corresponding Kripke structure $\mathscr{K}_{C_\mathcal{N}}$ via contraposition. We show $\mathcal{N} \not\models_{\mathtt{LTL}} \psi$ iff $\mathscr{K}_{C_\mathcal{N}} \not\models \psi'$ by transforming the counterexamples.
Soundness: Let $\mathcal{N} \not\models_{\mathtt{LTL}} \psi$. Thus, there is a run $\beta = (\mathcal{N}^R, \rho)$ and a covering firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ such that $\sigma(\zeta) \not\models_{\mathtt{LTL}} \psi$. We create a path $\pi = \pi_0 \pi_1 \cdots$ of $\mathscr{K}_{C_\mathcal{N}}$ by $\pi_0 = (\emptyset, \emptyset, \rho(M_0)_o, \rho(M_0)' \cup \{\mathtt{i}'\})$ and if ζ is infinite

$$\pi_i = (\{\rho(t_{i-1})\}, \rho(M_{i-1}) \cup \{\mathtt{i}\}, \{\rho(t_{i-1})_o\} \cup \rho(M_{i-1})_o, \rho(M_i)' \cup \{\mathtt{i}'\})$$

for all $i \in \mathbb{N} \setminus \{0\}$. If $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots [t_{n-1}\rangle M_n$ is finite the first n states are created as above and we define

$$\pi_{n+1} = (\emptyset, \rho(M_n) \cup \{\mathtt{i}\}, \rho(M_n)_o, \rho(M_n)' \cup \{\mathtt{i}', \mathtt{e}'\})$$

and for all other $j > n+1$ we define π_j equally apart from the current values of the latches also contain $\mathtt{e}$ and e_o is also set for the outputs:

$$\pi_j = (\emptyset, \rho(M_n) \cup \{\mathtt{i}, \mathtt{e}\}, \rho(M_n)_o \cup \{e_o\}, \rho(M_n)' \cup \{\mathtt{i}', \mathtt{e}'\}).$$

We show that the sequence π is indeed an initial path of $\mathscr{K}_{C_\mathcal{N}}$. Since $\rho(M_0) = In$ (because ζ covers a run of $\mathcal{N}$), directly $\pi_0 \in S_0$ holds. We show that for all $i \in \mathbb{N}$: $(\pi_i, \pi_{i+1}) \in \rightarrow$, i.e., $L'_i = L_{i+1}$ and $(I_{i+1}, L_{i+1}, O_{i+1}, L'_{i+1}) \models \mathcal{F}$ for $\pi_i = (I_i, L_i, O_i, L'_i)$ and $\pi_{i+1} = (I_{i+1}, L_{i+1}, O_{i+1}, L'_{i+1})$. The first clause can directly be seen by the definition of π. For the second clause, we show that all of the defined π_i in the finite as well as the infinite case satisfy $\mathcal{F}$ by checking each of the conjuncts: The conjunct $\mathtt{out}_P$ is satisfied, because the current values p are set as output ($\mathtt{i}$ is true) and this fits to the definition of π_i. The conjunct $\mathtt{out}_T$ is satisfied, because transition t_{i-1} is enabled in M_{i-1} due to ζ and no other transition is set in π_i. The conjunct $\mathtt{latch}_P$ is satisfied, because $\mathtt{succ}(p)$ yields the marking resulting from firing transition t_{i-1} in the current values of the latches (here $\rho(M_{i-1})$) because $\mathtt{noT}$ is not satisfied. Due to ζ, the resulting marking is $\rho(M_i)$, which fits π_i. The other conjuncts are directly satisfied by the construction. In the case of a finite firing sequence the defined π_j also satisfy $\mathcal{F}$: The conjunct $\mathtt{out}_P$ is satisfied, with the same arguments as in the previous case. The conjunct $\mathtt{latch}_P$ is satisfied, because $\mathtt{noT}$ is satisfied and therewith the current values of the latches are

taken and $\texttt{out}_T$ is also satisfied, because no transition is applied to the input. Also because of `noT` the conjunct $\texttt{latch}_\texttt{e}$ is satisfied and with the additional constraints for all but the first $j > n$ the conjunct $\texttt{out}_\texttt{e}$ is satisfied for all $i \in \mathbb{N}$. The subpath π^1 satisfies $\square(e_o \rightarrow \square e_o)$, because no e_o is ever set by the construction in the case of an infinite ζ and in the other case e_o is set for every $j + 1 > n$. Since by the construction $\sigma(\zeta)(i) \sim \phi_{i+1}$ directly holds, the path π does not satisfy $\bigcirc(\square(e_o \rightarrow \square e_o) \rightarrow \widetilde{\psi})$. Hence, $\mathscr{K}_{C_\mathscr{N}} \not\models \psi'$.

Completeness: Let $\mathscr{K}_{C_\mathscr{N}} \not\models \psi'$. Thus, there is an initial path $\pi = \pi_0\pi_1 \cdots$ of $\mathscr{K}_{C_\mathscr{N}}$ not satisfying $\bigcirc(\square(e_o \rightarrow \square e_o) \rightarrow \widetilde{\psi})$. Hence, the subpath π^1 satisfies $\square(e_o \rightarrow \square e_o)$ and not $\widetilde{\psi}$. In the case that $e_o \notin O_i$ for all $i \in \mathbb{N}$ holds, we create a infinite firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots$ with $\rho(M_i) = \{p \in \mathscr{P} \mid p_o \in O_{i+1}\}$ and $\rho(t_i) \in \{t \in \mathscr{T} \mid t_o \in O_{i+1}\}$. Otherwise, if there is an $i \in \mathbb{N}$ with $e_o \notin O_i$, say $i = n + 1$ is the first of such occurrences, we create a finite firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle \cdots [t_{n-1}\rangle M_n$ for the first $n - 1$ steps as before and for the nth step we define $\rho(M_n) = \{p \in \mathscr{P} \mid p_o \in O_{n+1}\}$ as above.

This is indeed a firing sequence, because $\rho(M_0) = In$ holds by the definition of $\mathscr{K}_{C_\mathscr{N}}$, the construction of ζ, and since π is initial. All transitions are firable and yield the respective successor marking because of $\mathscr{F}$, since π is a path. This is because the error flag is maximally set in the last step, and thus, there must be exactly one enabled transition applied to the inputs. Therewith $\texttt{succ}(p)$ yields the correct successor marking and this is the output of the next step. Since π satisfies $\bigcirc(\square(e_o \rightarrow \square e_o)$, we know that also in the finite case of ζ it holds $\sigma_R(\zeta)(i) \sim \phi_{i+1}$, since in π the output markings stay the same in situations where e_o is set, because this is only possible if `notT` is true. Since π^1 does not satisfy $\widetilde{\psi}$, $\sigma_R(\zeta) \not\models_{\texttt{LTL}} \psi$. Constructing the corresponding run yields the conclusion.

The single-exponential time bound directly follows from Lemma 16 with the polynomial size of the circuit and the linear size of the formula and Definition 42 creating the Kripke structure of exponential size. These can be checked in linear time in the size of the state space and in exponential time in the size of the formula [CHVB18]. □

6.4.5 Formal Construction of the Petri Net $\mathscr{N}_{\|}^{>}$

Let `ID` be a set of unique identifiers and $\nu_\mathscr{N} : \mathscr{P} \cup \mathscr{T} \rightarrow \texttt{ID}$ an injective naming function which uniquely identifies every place and transition of a given Petri net $\mathscr{N}$ (or of a Petri net with transits). We omit the subscript if the net is clear from the context. To keep the presentation clear, we often directly use *identifier* for a node $n \in \mathscr{P} \cup \mathscr{T}$ with $\nu(n) = \textit{identifier}$.

The construction of a Petri net with transits to a standard P/T Petri net with inhibitor arcs is given by the following definition.

▶ **Definition 44 (Petri Net with Transits to a P/T Petri Net).** For a Petri net with transits $\mathscr{N}_T = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and a Flow-LTL formula φ with $n > 0$ flow subformulas, a Petri net $\mathscr{N}_{\|}^{>} = (\mathscr{P}^{>}, \mathscr{T}^{>}, \mathscr{F}^{>}, \mathscr{F}_I^{>}, In^{>})$ with inhibitor arcs and a labeling function $\lambda : \mathscr{T}^{>} \rightarrow \mathscr{T}$ are defined as follows:

(p) The places of the original net $\mathcal{N}_T$ are copied $n+1$ times:

$$\mathscr{P}^{>} = \mathscr{P} \cup \bigcup_{\{1,\dots,n\}} (\{[\iota]_i\} \cup \{[p]_i \mid p \in \mathscr{P}\})$$

(t) For every transition $t \in \mathscr{T}$ and every combination of which subnet is tracking which transit (or no transit with marker $\circ$), there is one transition in $\mathcal{N}_{\|}^{>}$. Each transition is connected to the original part of the net according to t. For the subnet part, it either a) moves the token from the initial place according to the transit, or b) moves the token from the corresponding ingoing place of the transit according to the transit, or, in the case that the subnet is not involved in any of the transits, c) is connected by inhibitor arcs to all ingoing places of the transition t.

$$\begin{aligned}
&\forall t \in \mathscr{T} : \forall \mathfrak{c} = ((x_1, p_1), \dots, (x_n, p_n)) \in (\Upsilon(t) \cup \{\circ\})^n : \exists t^{>} \in \mathscr{T}^{>} : \\
&\quad \forall (p,t), (t,q) \in \mathscr{F} : (p, t^{>}), (t^{>}, q) \in \mathscr{F}^{>} \wedge \nu(t^{>}) = \nu(t)_{\mathfrak{c}} \wedge \lambda(t^{>}) = \nu(t) \wedge \\
&\quad \forall i \in \{1, \dots, n\} : x_i = \rhd \quad \Longrightarrow\ ([\iota]_i, t^{>}), (t^{>}, [p_i]_i) \in \mathscr{F}^{>} \wedge \\
&\qquad\qquad x_i, p_i \in \mathscr{P} \ \Longrightarrow\ ([x_i]_i, t^{>}), (t^{>}, [p_i]_i) \in \mathscr{F}^{>} \wedge \\
&\qquad\qquad (x_i, p_i) = \circ \ \Longrightarrow\ \forall (p,t) \in \mathscr{F} : ([p_i]_i, t^{>}) \in \mathscr{F}_I^{>}
\end{aligned}$$

(I) The initial marking is given by $In^{>} = In \cup \{[\iota]_i \mid i \in \{1, \dots, n\}\}$.

The sets $\mathscr{T}^{>}, \mathscr{F}^{>}$, and $\mathscr{F}_I^{>}$ are defined as the smallest sets fulfilling condition *(t)*. The identifiers with the square brackets and those with a combination $\mathfrak{c}$ in their index are fresh identifiers. ◀

Note that for $n = 0$ flow subformulas we can just use the Petri net with transits $\mathcal{N}_T$ and only omit the transits. The correctness of the transformation for the parallel approach is already proven in Sec. 6.2.2.

7 AdamMC – A Model Checker for Petri Nets with Transits

In Chap. 6 the sequential and the parallel approach for model checking safe Petri nets with transits against Flow-LTL are introduced. In this chapter we present the tool AdamMC that implements these approaches. AdamMC focuses on the implementation of the single-exponential time algorithms, i.e., the approaches with specifications in Flow-LTL, and the application domain of software-defined networking. However, it also contains algorithms for specifications in Flow-CTL with the automata transformations (cp. Chap. 5) in an early development state. AdamMC is open source[1] (GPL-3.0 License) and is integrated into the Adam framework (cp. Chap. 15). To keep it modular and to allow for an easy reuse even of parts of the procedures, we split AdamMC into several repositories that can be separately built and used as libraries in other projects. For example, there is a repository for the basic data structure for Petri nets with transits and another one for the logics. The tool is based on the publication [FGHO19a] introducing a prototype for the sequential approach and [FGHO20a] implementing the parallel approach and turning AdamMC into full-fledged tool. For both publications we created artifacts with the complete results, the benchmark data, and the corresponding versions of the tool to easily replicate the results of the paper in a corresponding virtual machine [GH19; GH20]. Both artifacts achieved the artifact evaluation badge of the artifact evaluation committee of ATVA and CAV, respectively, stating that the artifact is *consistent*, *easy to use*, *replicable*, *well-documented*, and *complete*. More technical details to the framework can be found in Chap. 15.

In Sec. 7.1, we present the general workflow of AdamMC and briefly introduce the three application areas of AdamMC focusing on Flow-LTL specifications: I) model checking standard properties of software-defined networks with concurrent updates, II) generally model checking Petri nets with transits against Flow-LTL, and III) model checking standard safe Petri nets against LTL with places and transitions as atomic propositions. Section 7.2 gives a brief insight into some optimization techniques in AdamMC that are made available to the user. In Sec. 7.3, we present our benchmarks for the application domain of software-defined networks showing a significant performance increase for the parallel approach in comparison to the sequential one.

This chapter is based on [FGHO20a] and the corresponding full version [FGHO20b].

[1] https://github.com/adamtool/adammc

7.1 Application Areas and Workflow

AdamMC consists of modules for three application areas: checking concurrent updates of software-defined networks against common assumptions and specifications, checking safe Petri nets with transits against Flow-LTL, and checking safe Petri nets against LTL. The general architecture and workflow of the model checking procedure is given in Fig. 12.1.

Software-Defined Networks: In Sec. 2.1, we motivate Petri nets with transits and Flow-LTL with the application domain of concurrent updates of software-defined networks. AdamMC provides dedicated algorithms for this domain. The tool automatically encodes an initially configured network topology and a concurrent update as a Petri net with transits [FGHO20a]. We provide parsers for the *network topology*, the *initial configuration*, the *concurrent update*, and Flow-LTL (Input I). Additionally, the automatic generation of Flow-LTL formulas for *common network properties* like *connectivity*, *loop freedom*, *drop freedom*, and *packet coherence* [FGHO19a] is implemented for correspondingly annotated input elements.

Petri Nets with Transits: The most general input of AdamMC for model checking distributed systems with local data flows is Input II. Here, the tool takes a safe Petri net with transits and a Flow-LTL formula specifying requirements on the global configuration of the system and the local flow of the data. For this, AdamMC extends a parser for Petri nets provided by APT [Uni12] and provides a parser for Flow-LTL. The reduction methods presented in Sec. 6.2.1 and Sec. 6.2.2 to create a safe Petri net and an LTL formula are implemented and several parameters for tweaking the reductions are provided (cp. Sec. 7.2).

Petri Nets As Input III, AdamMC supports the model checking of safe Petri nets against LTL with both places *and* transitions as atomic propositions. In particular, this makes it easy to check specifications under *fairness* and *maximality* assumptions (cp. Example 1). The tool provides both the *interleaving* and the *concurrent* view on the maximality of runs in the system by automatically generating and adding the corresponding formulas. Additionally, AdamMC provides dedicated algorithms to check the system only for *interleaving-maximal* runs. State-of-the-art tools like LoLA [Wol18] and ITS-Tools [Thi15] are restricted to interleaving-maximal runs and places as atomic propositions. For Petri net model checking, we allow for Petri nets in APT [Uni12] and PNML [BCHK+03] format as input and provide a parser for LTL formulas.

The construction of the circuit from a given safe Petri net and an LTL formula is presented in Sec. 6.3. AdamMC automatically generates this circuit in the Aiger format [BHW11]. MCHyper [FRS15] is then used to combine the constructed circuit and the in Sec. 6.3.2 constructed LTL formula into another Aiger circuit. MCHyper is a verification tool for HyperLTL [CFKM+14] that subsumes LTL. The actual model checking is carried out by the hardware model checker ABC [Ber; BM10b].

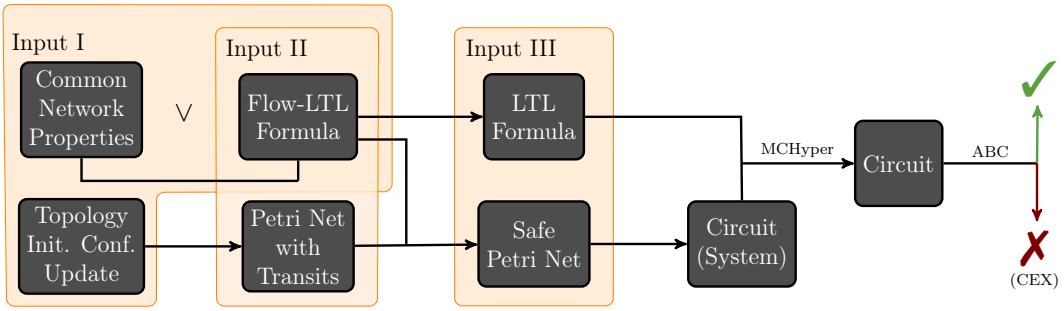


Fig. 7.1: *Overview of the workflow of* AdamMC*: The application areas of the tool are given by three different input domains: software-defined networks / Flow-LTL (Input I), Petri nets with transits / Flow-LTL (Input II), and Petri nets / LTL (Input III).* AdamMC *performs all unlabeled steps in this figure. MCHyper [FRS15] creates the final circuit that ABC [Ber; BM10b] and its large toolbox of verification and falsification algorithms checks to answer the initial model checking question.*

ABC provides a toolbox of state-of-the-art verification and falsification techniques like IC3 [Bra11]/PDR [EMB11], interpolation (INT) [McM03], and bounded model checking [BCRZ99] (BMC, BMC2, BMC3).

As output for all three modules, AdamMC transforms a possible counterexample (CEX) from ABC into a counterexample to the Petri net (with transits). With the web interface presented in Sec. 15.2 we can simulate these counterexamples. Counterexamples for Petri nets with transits can additionally be visualized in a PDF document showing the violating firing sequence and the corresponding violating data flow chains. Furthermore, all inputs can be visualized with Graphviz and the dot language [EGKN+04]. When no counterexample exists, AdamMC verified the input successfully.

7.2 Optimizations

Various optimizations parameters can be applied to the model checking routine to tweak the performance. Table 7.1 gives an overview of the major parameters.

For the reduction step from a Petri net with transits into a standard Petri net, we, on the one hand, allow to switch between the *sequential* approach (cp. Sec. 6.2.1) and the *parallel* approach (cp. Sec. 6.2.2) and, on the other hand, provide for each approach one version using inhibitor arcs (*inhibitor*) and one version using tokens (*act. tokens*, for activation) to de- and activate certain transitions. We found in our experiments that the versions of the sequential and the parallel approach with inhibitor arcs to track flow chains are generally faster than the versions without and that the parallel approach outperforms the sequential one for few local flow requirements [FGHO20b].

Furthermore, we optimized the sequential approach by only using until operators in the replacement of the atomic propositions corresponding to the transitions in the Flow-LTL formula, when the transitions not directly occur in the scope of an eventually operator.

Tab. 7.1: *Overview of the major optimization parameters of* AdamMC*: The three reduction steps depicted in the first column can each be executed by different algorithms. The first step allows us to combine the optimizations of the first and second row.*

1) Petri Net with Transits $\rightsquigarrow$ Petri Net	sequential		parallel	
	inhibitor	act. tokens	inhibitor	act. tokens
2) Petri Net $\rightsquigarrow$ Circuit	explicit		logarithmic	
3) Circuit $\rightsquigarrow$ Circuit	gate optimizations			

This significantly reduces the size of the formula. Additionally, we provide dedicated algorithms when only places (or the fireability of transitions) are used in the formula.

For the reduction step from a Petri net into a circuit, we provide an *explicit* and a *logarithmic* encoding of the transitions. The logarithmic encoding needs less latches, but more gates due to the necessary de- and encoding. The experimental results show that logarithmically encoded transitions often had a better performance than the same step with explicitly encoded transitions for both approaches [FGHO20b].

However, several possibilities to reduce the number of gates of the created circuit (for both the circuit given to and the one returned by MCHyper) worsened the performance of some benchmark families and improved the performance of others. The same is true for various approaches to optimizing the formula construction like encoding the maximality assumptions in the circuit rather than the formula and different kind of formulas preventing the run to get stuck in any subnet. Consequently, all parameters are selectable by the user and a script is provided in the corresponding artifact [GH20] to automatically compare different settings. Furthermore, we preset the parameters in AdamMC to the best settings with respect to our experimental results. An overview of the selectable optimization parameters and more details can be found in the documentation[2]. The main improvement claims can be retraced by the case study in Sec. 7.3.

7.3 Benchmarks

We conduct a case study based on software-defined networks to show the scalability of our algorithms presented in Chap. 6. Especially, we compare the performance of the *sequential* (Sec. 6.2.1) versus the *parallel* approach (Sec. 6.2.2). As a summary of the result, we see that the parallel approach outperforms the sequential one for the case study of software-defined networks. Figure 7.2 shows the difference of the running times for one flow subformula. For one flow subformula the theoretical complexity of the parallel and the sequential approach is the same (cp. Sec. 6.2). Figure 7.3f still shows the superiority of the parallel approach for up to five flow subformulas despite its inferior theoretical complexity. This result can mainly be attributed to the structure of the Petri nets with transits resulting from a software-defined network. The exponentiality of the parallel approach with respect to the number of flow subformulas depends on the number

[2] https://github.com/adamtool/adammc/blob/master/doc/documentation.pdf

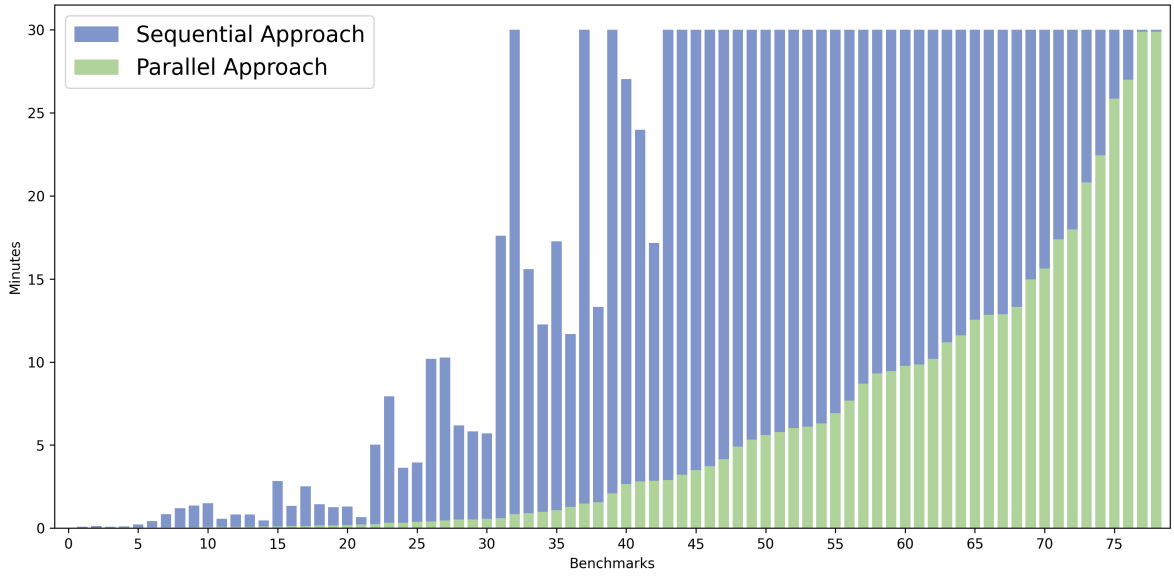


Fig. 7.2: *Comparison of the sequential and the parallel approach with logarithmic encoding for the falsification benchmarks checking the correctness of concurrent updates with respect to the connectivity property in software-defined networks. The benchmarks are sorted by the running times of the parallel approach. A complete bar indicates a timeout of 30 minutes. Only the benchmarks where not both approaches had a timeout are plotted. Each bar is the best result of the BMC2 or BMC3 algorithms.*

of transits the transitions possess.

For the case study we use real-world network topologies from [KNFB+11]. For each network, we choose at random an ingress switch, an egress switch, and a loop- and drop-free initial configuration between the two. Another forwarding table between the two switches is created for the final configuration and an update from the initial to the final configuration is chosen at random. For the first benchmark family (T, for true/verifiable), AdamMC verifies that the update maintained the *connectivity* between the ingress and the egress switch. This means that all packages eventually reach the egress node. For the second benchmark family (F, for false/falsifiable), a switch different to the egress switch is chosen, such that it is ensured that not all packages reach this switch. AdamMC then falsifies the connectivity property for this switch.

We compare the sequential approach with an explicit encoding of the transitions in the circuit (expl. enc.) against the sequential approach with a logarithmic encoding of the transitions (log. enc.), and against the parallel approach with the logarithmic encoding (parallel appr.). Table 7.2 summarizes the results. The results in the upper part of the table belong to the verification of the connectivity requirement, i.e., the specification is satisfied (✓). These results are all calculated with the IC3 algorithm of ABC. The lower part concerns the falsification benchmarks, i.e., the specification is violated (✗), and reports the winner of the bounded model checking algorithms of ABC with dynamic unrolling (BMC2 and BMC3). The last two rows summarizes the results: The

logarithmic encoding with 79.15h accumulated running time and 138 timeouts generally outperforms the explicit encoding with 82.99h and 146 timeouts. More significantly, the parallel approach is the clear winner with 27.62h accumulated running time and 9 timeouts. To this end, we are able to verify concurrent overlapping updates in network topologies with up to 38 switches ($\#Sw$) and falsify specification in networks with up to 82 switches in the given time bound of 30 minutes. For rather small networks, the tool needs only a few seconds to verify and falsify updates.

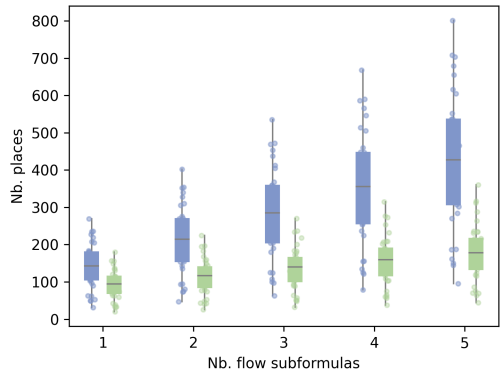
For a second analysis, we increase the number of flow subformulas used in the specification and compare the sequential with the parallel approach both using the logarithmic encoding. Figure 7.3 shows the results for Flow-LTL formulas with up to five flow subformulas. In Fig. 7.3a to Fig. 7.3e the difference in the sizes of the transformed net, the formula, and the circuit that is finally given to ABC are depicted using box plots. For the number of places we can see in Fig. 7.3a that the sequential approach (depicted in blue) needs significantly more places than the parallel approach (depicted in green). Furthermore, the number of places increases more strongly with the number of flow subformulas for the sequential approach. Since the number of latches of the generated circuit is mainly driven by the number of places of the constructed Petri net, we can see the corresponding results in Fig. 7.3d for the latches.

However, the number of transitions for the parallel approach notably increases due to the exponential dependency on the number of flow subformulas from two flow subformulas on. Since the size of the subformula strongly depends on the number of transitions, we can see the same trend in Fig. 7.3c. At this point, an optimization technique of the sequential approach comes into play, which further supports this trend. The sequential approach only replaces a transition in the formula with an until operator over a large subset of all transitions, when the transition is not in the direct scope of an eventually operator. The weak fairness and maximality assumptions guarantee exactly this. Hence, we observe barely any increase of the size of the formula while increasing the number of subformulas for the sequential approach.

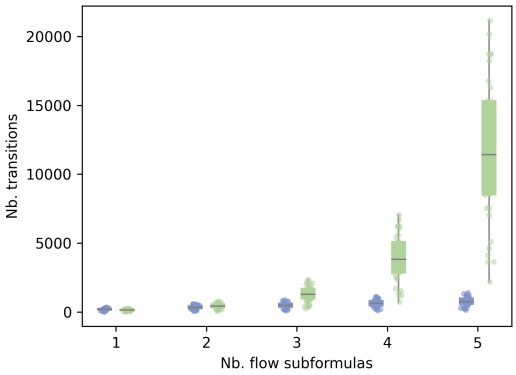
Finally, in Fig. 7.3e we can observe that the number of transitions has a stronger impact on the number of gates than the number of places. While in the beginning the number of gates used in the parallel approach is still smaller than the number of gates needed for the sequential one, the parallel approach overtakes the sequential one from three flow subformulas on. This distance increases considerably when increasing the number of flow subformulas.

Figure 7.3f shows the comparison of the running times of the approaches for an increasing number of flow subformulas. We can see that for five flow subformulas the parallel approach still outperforms the sequential one. The gray hatched areas depict the times the approaches run into a timeout. The gray dashed horizontal line indicates the maximum time available for the approaches. Note that the figure cannot tell us whether the performance of the parallel approach converges to that of the sequential one, since the sequential approach runs into timeouts for a large part of the benchmarks already for one flow subformula.

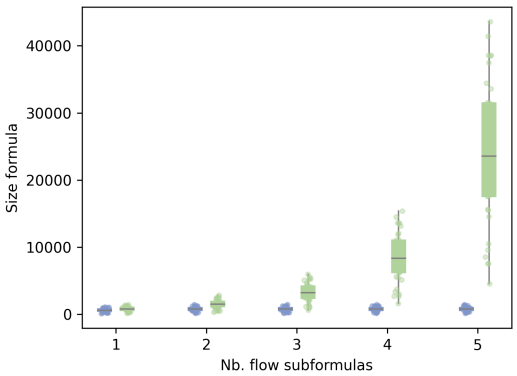
These figures indicate that the main reason for the superiority of the parallel approach for these benchmark suites is the larger number of places and therewith the larger num-



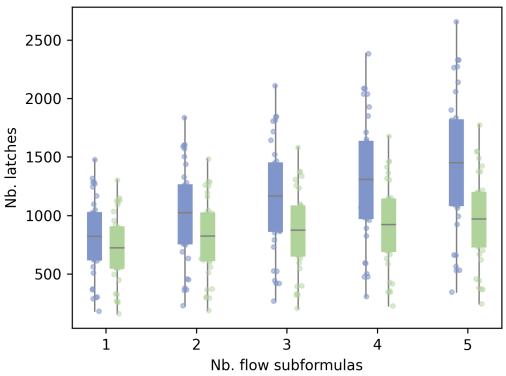
(a) *Comparing the number of* places.



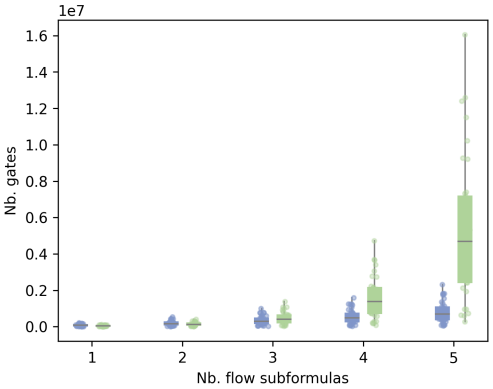
(b) *Comparing the number of* transitions.



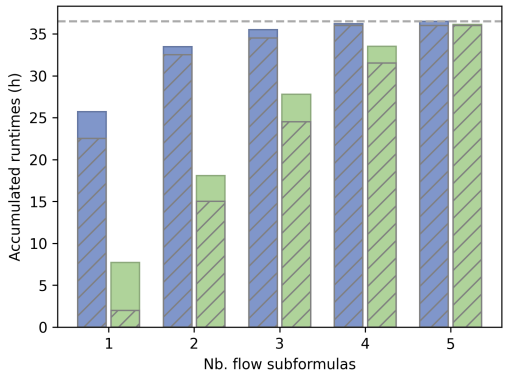
(c) *Comparing the size of the* formulas.



(d) *Comparing the number of* latches.



(e) *Comparing the number of* gates.



(f) *Comparing the* accumulated runtimes.

Fig. 7.3: *Comparison of the sizes of the constructed elements for the* sequential *(blue) versus the* parallel *(green) approach with respect to an increasing number of flow subformulas. In Fig. 7.3a to Fig. 7.3e we use box plots representing the median value with the horizontal gray lines, the area where the median 50% of the data is located as colored boxes, the range of the minimal and maximal values by the vertical gray lines, and the data points itself as small circles. Figure 7.3f compares the accumulated runtimes the approaches needed to solve the complete falsification benchmark suite. The hatched areas mark the timeouts and the gray dashed line the maximum time available for the approaches.*

ber of latches needed by the sequential approach. This supports the observation that in practice the size of the model and not the size of the formula is the source of intractability [DLS06]. Increasing the number of flow subformulas or the amount of transits per transition degrades these benefits, as they have an exponential impact.

The above observation suggests another optimization possibility. For the implementation of the synthesis problem presented in Chap. 13, the BDDs are *token-oriented* rather than *place-oriented*. This means we do not have one variable per place, but encode the ids of the places of a marking logarithmically. Since problems modeled with Petri nets often use considerably more places than tokens, fewer latches would be required, but the complexity of the algorithms and the number of gates would increase. However, by switching to a token-oriented encoding, the performance of the synthesis approach increased significantly.

Tab. 7.2: *We compare the explicit and logarithmic encoding of the sequential approach with the parallel approach for the connectivity requirement. The results are the average over five runs from an Intel i7-2700K CPU with 3.50 GHz, 32 GB RAM, and a timeout (TO) of 30 minutes. The runtimes are given in seconds.*

			expl. enc.			log. enc.			parallel appr.		
T/F	Network	#*Sw*	Alg.	Time	⊨	Alg.	Time	⊨	Alg.	Time	⊨
T	Arpanet196912	4	IC3	12.08	✓	IC3	9.89	✓	IC3	**2.18**	✓
T	Napnet	6	IC3	146.49	✓	IC3	96.06	✓	IC3	**4.75**	✓
T	Epoch	6	IC3	240.57	✓	IC3	214.70	✓	IC3	**6.78**	✓
T	Telecomserbia	6	IC3	1182.43	✓	IC3	912.76	✓	IC3	**12.12**	✓
T	Layer42	6	IC3	133.20	✓	IC3	131.68	✓	IC3	**6.26**	✓
	...			...			...			...	
T	Heanet	7	IC3	806.81	✓	IC3	84.62	✓	IC3	**30.30**	✓
T	HiberniaIreland	7	-	TO	?	-	TO	?	IC3	**26.58**	✓
T	Arpanet19706	9	-	TO	?	IC3	362.21	✓	IC3	**11.33**	✓
T	Nordu2005	9	-	TO	?	-	TO	?	IC3	**12.67**	✓
	...			...			...			...	
T	Fatman	17	-	TO	?	IC3	1543.34	✓	IC3	**162.17**	✓
T	Nextgen	17	-	TO	?	-	TO	?	IC3	**403.33**	✓
T	Nordu2010	18	-	TO	?	-	TO	?	IC3	**50.11**	✓
T	Pacificwave	18	-	TO	?	-	TO	?	IC3	**932.60**	✓
T	Ans	18	-	TO	?	-	TO	?	IC3	**1511.30**	✓
	...			...			...			...	
T	Myren	37	-	TO	?	-	TO	?	IC3	**1309.23**	✓
T	KentmanJan2011	38	-	TO	?	-	TO	?	IC3	**1261.32**	✓
F	Arpanet196912	4	BMC3	2.18	✗	BMC3	1.85	✗	BMC3	**1.20**	✗
F	Napnet	6	BMC2	4.17	✗	BMC2	5.22	✗	BMC3	**1.48**	✗
F	Epoch	6	BMC3	15.41	✗	BMC3	13.56	✗	BMC3	**2.39**	✗
	...			...			...			...	
F	Rhnet	16	-	TO	?	-	TO	?	BMC3	**49.94**	✗
F	Fatman	17	BMC3	168.78	✗	BMC3	169.82	✗	BMC3	**6.72**	✗
F	Goodnet	17	-	TO	?	-	TO	?	BMC3	**378.15**	✗
	...			...			...			...	
F	Harnet	21	BMC3	1410.25	✗	BMC3	735.73	✗	BMC3	**58.40**	✗
F	Belnet2009	21	BMC2	1146.26	✗	BMC2	611.81	✗	BMC3	**24.26**	✗
F	Garr200404	22	BMC3	45.90	✗	BMC3	49.38	✗	BMC3	**4.70**	✗
	...			...			...			...	
F	KentmanJan2011	38	BMC3	167.92	✗	BMC3	86.44	✗	BMC2	**9.35**	✗
F	Cesnet200511	39	-	TO	?	-	TO	?	BMC3	**249.15**	✗
	...			...			...			...	
F	Forthnet	62	-	TO	?	-	TO	?	BMC3	**752.80**	✗
F	Latnet	69	-	TO	?	-	TO	?	BMC2	**209.20**	✗
F	Ulaknet	82	-	TO	?	-	TO	?	BMC2	**1043.74**	✗
Sum of runtimes (in hours):				82.99			79.15			27.62	
Nb of TOs (of 230 exper.):				146			138			9	

8 Related Work

Decomposing the overall problem into individual subproblems is a common strategy for computer scientists. Looking at the control part of the system separately from the data is a division that lends itself. For example, an essential part of software-defined networks [KRVR+15] is the clear separation of the data plane and the control plane to facilitate the development process.

A common formalism that provides separate features for the data and the control of the system in a non-distributed setting are *extended finite state machines* (EFSMs) [CK93; CK96]. An EFSM is a finite state machine extended with a set of variables. The finite state machine models the control part and the set of variables can be used as guards at the transitions of the finite state machine and can be updated by the control flow to model the data part. In the formal methods community, the use of EFSMs is widespread for blending control and data [WTD16; RSS21], e.g., for symbolic model checking [TOHT06], automata learning [CHJS18], and data-flow analysis [BBS11].

In [HHFM+94], the specification language SL [OR93] for reactive systems with real-time constraints is used to split the description of the desired behavior of the system components into a *trace* part, a *state* part, and a *timing* part. The trace part specifies the *control* flow of the system by means of regular expressions, whereas the state part specifies the *data* in the system, i.e., the communication values which possibly make use of local state variables. Such separation can allow for processing parts of the system automatically where including the concrete data would lead to a state space too large to process. For example, the separation of data and control for the synthesis of reactive monolithic systems with the *temporal stream logic* (TSL) [FKPS19] made it possible to synthesize an autonomous driving controller, a music player app [FKPS19], and an arcade game for field programmable gate arrays (FPGAs) [GHKF19].

Abstracting from the concrete data values of the system is a common approach to obtain finite-state systems for model checking with an infinite data domain [ESK14]. Oftentimes, the control part of the system is easy enough to be finite-state, whereas processing the concrete data leads to infinitely many states. Contrariwise, in many cases we are not interested in the concrete values of the data. For example, in the software-defined network example presented in Sec. 2.1, we are not concerned with the concrete content of the infinitely many packets flowing through the network, but only with their forwarding and which switches are involved in the process. Wolper introduces in [Wol86] the notion of *data-independence*. Intuitively, this means that changing the input of the program does not change the behavior of the program, but at most the corresponding output data. This significantly increased the applicability of model checking and synthesis algorithms based on propositional temporal logic specifications [Wol86]. The

idea of data-independence inspired a large body of further work (e.g., [RB99; NM10; SYGA+14; ESK14; KW19].

We refer to the survey [Sch02] for the general complexity results and the underlying algorithmic ideas of the model checking problem for CTL* and its fragments. We recall here the results for CTL*, CTL, and LTL. The model checking problem for CTL* is PSPACE-complete [CES83; CES86; EL87] and can be solved in $2^{\mathcal{O}(|\varphi|)}\mathcal{O}(|\mathscr{K}|)$ time [EL87; KVW00], where $|\varphi|$ is the length of the formula and $|\mathscr{K}|$ the size of the Kripke structure. For the fragment CTL, the model checking problem is P-complete [CES86; Sch02] and can be solved in $\mathcal{O}(|\varphi| \cdot |\mathscr{K}|)$ time [CES86; AC88]. Even though the time complexity is linear in both the size of the model and the length of the specification, usually the size of the model is the dominant factor due to the state explosion problem [CV03]. For the fragment of LTL, the model checking problem is again PSPACE-complete [SC85] and can be solved in $2^{\mathcal{O}(|\varphi|)}\mathcal{O}(|\mathscr{K}|)$ time [LP85; VW86; Var95a]. Note that this does not necessarily mean that CTL model checking is more efficient than LTL model checking. On the one hand, the size of the state space is again usually the limiting factor in practice [CHV18]. On the other hand, if the same properties are expressed in CTL as in LTL (if possible), the LTL formulas can be exponentially shorter than the CTL equivalents [BK08].

A considerable number of tools exist for the analysis and model checking of Petri nets, reflecting not least the many variants of Petri net models in which the tools can specialize (e.g., `Tina` [BRV04; BV06] for Timed Petri Nets, GreatSPN [CFGR95; BBDF+01; BBCP+09] for Generalized Stochastic Petri Nets, CPN/Tools [Jen; CJK97; BMJA+01] for Colored Petri Nets). An annual model checking contest [KBGH+21] provides tracks for the model checking of Petri nets against LTL specifications and CTL specifications, where for example the tools TAPAAL [BJS09; DJJJ+12], ITS-Tools [Thi15], LoLA [Sch00; Wol18], and GreatSPN [CFGR95; BBDF+01; BBCP+09] compete regularly. RGMEDD* [ADG20] is a recent tool that allows for model checking CTL* specifications on Petri nets.

The common approach for LTL and CTL model checking in Petri nets is to permit only places as atomic propositions and not transitions (*state-based* logics [Esp96]). This is also reflected by the model checking contest [KBGH+21], where the specifications are also limited to the cardinality of the places or to the firability of transitions (i.e., a conjunction of the cardinalities of places). To still impose fairness assumptions [Fra86], usually dedicated algorithm are used for model checking (see [EL85; EL87] for CTL, [LP85; KPR98; KPRS06] for LTL, and [LH00] for LTL on Petri nets). Additionally allowing transitions as atomic propositions enables us to directly encode fairness assumptions on the transitions of the Petri net and also maximality assumptions on the runs within the logic. However, this requires to decide on an in- or outgoing semantics, and this generality comes with the drawback of a larger state space induced by those transitions actually used in the formula. Specifically for labeled Petri nets, *action-based* versions of LTL and CTL are considered, which focus only on the transitions (respectively their labels) of the Petri net to state the desired properties [Esp96]. In these logics, the set of proposition only contains *true* and the next operator is equipped with a set of transitions K with the semantics that $\mathbf{X}_K$ is only satisfied in a state, if

a transition is taken from K to reach the next state. For the state-based as well as the action-based versions of LTL and CTL the model checking problem for 1-bounded Petri nets is PSPACE-complete [Esp96]. Lifting these results to their combination would yield a PSPACE-completeness result for 1-bounded Petri nets with transits and Flow-LTL. Another approach using variables for places *and* transitions, is the SAT encoding in [Hel99a; Hel99b] for the reachability problem in 1-bounded Petri nets. A good survey and introduction for decidability problems for 1-bounded and general Petri nets can be found in [Esp96; EN94]. The first successful applications of unfolding techniques to verification problems is due to McMillan [McM92; McM93; McM95].

The reductions in Chap. 5 and Chap. 6 show that we can express our model checking problem with standard Petri nets and LTL. The size and lack of clarity of the constructed Petri nets and LTL formulas already for the small examples presented in this thesis show that we do not want to model the problem directly at this level. The same applies when expressing the problem in other Petri net models like Colored Petri nets [Jen97]. Naively, one could use one uniquely colored token for each newly created data flow. This counteracts the idea of abstracting from the concrete data and leads to an infinite number of colors due to the possible infinitely many data flows. As a result, the model is as expressive as a Turing machine [Tur37], which makes most general questions undecidable [Pet80]. Another approach is to have one color for the control tokens and another color for the data flow. This abstracts from the concrete data and only needs two colors, but still results in possibly infinitely many colored tokens and does not cover the merging of the data flow. Going one step further and also merging the data flow appropriately results in introducing several transitions for each combination of a transition's transits. For each transit we have to check whether there is already a data flow at the postset place and whether there is a data flow at all at the preset place. This again takes clarity from the model and increases the chances for modeling errors.

There is a large body of work regarding the presented application domains for Petri nets with transits. For *software-defined networking*, we refer to [KRVR+15] for an introduction. Solutions for correctness of updates of software-defined networks include *consistent updates* [RFRS+12; CFJM16], *dynamic scheduling* [JLGK+14], and *incremental updates* [KRW13]. Both explicit and SMT-based model checking is used to verify software-defined networks [CVPK+12; MTW14; MKAC+11; WMLT+13; BBGI+14; PIKL+15]. Closest to the approach using Petri net with transits, are models of networks as Kripke structures to use model checking for synthesis of correct network updates [ETVV17; MHC17]. In each step of the construction of a sequence of updates in the synthesis routine, a model checker is called to verify the correctness. The model checking subroutine of the synthesizer assumes that each packet sees at most one switch that was updated after the packet entered the network. This restriction is implemented with explicit waits, which can afterwards often be removed by heuristics. Our model checking routine does not require this assumption.

For *access control* policies for physical spaces, *access nets* [FCS11] also present a model based on Petri nets. For access nets, the tokens of the net represent the persons in the building and are equipped with types representing their roles. This can be seen as colored tokens in Coloured Petri Nets [Jen81]. Furthermore, the transitions may depend

on some global clock, and special transitions, called *mandatory*, are enforced to be taken in a state, when any of them is enabled. These extensions (individual tokens and time) result in large state spaces, which they reduce by applying abstraction methods in the line of [CGL94] and by focusing on reachability properties. In [BCRR+09] challenges from the practice of managing access control policies are presented, which were identified through a series of interviews with concerned administrators. Policies for access control of physical spaces where the model is a Kripke structure can be synthesized if no policy updates are required [TDB16].

Part II

Synthesis of Distributed Systems with Local Conditions

Contents

9 Motivation

In this chapter we intuitively introduce the new model, called *Petri games with transits*, with *local existential*, *universal*, and *Flow-LTL* winning objectives for synthesizing local controllers for asynchronous distributed systems with causal memory. Formal definitions of the model and the winning objectives are given in Chap. 11.

Petri games with transits extend Petri games by refining the flow relation in the same way as Petri nets with transits extend standard Petri nets. A *Petri game* [FO14; FO17] models the distributed synthesis problem as a multi-player game between two teams: the *environment* players, representing external influences (the *uncontrollable* behavior), and the *system* players, representing the processes (the *controllable* behavior). Each player is modeled as a token (depicted as a black dot) of an underlying standard Petri net. The places of the net determine the team affiliation of the players. Tokens on *environment places* (depicted as white circles) belong to the team of environment players and tokens on *system places* (depicted as gray circles) belong to the team of system players. Players make moves by taking transitions (depicted as blue squares) of the underlying Petri net. Figure 9.1 without the colored arrows represents a Petri game with one environment and one system player.

The common goal of the system players is to collaborate in such a way that each player satisfies their local specification against all possible behavior of the environment players. Petri games represent games with a local universal safety objective on the token flow, i.e., the system players must at all times avoid reaching designated *bad places* (depicted as double circled places). Additionally, they must prevent the environment players from reaching such places. Due to the distributed setting, players only have a limited view of the global system. Each player remembers their own *causal past*, i.e., the places and transitions used to reach the player's current place. This information is exchanged with all players participating at a joint transition. Each system player can select their next transition solely based on the locally available information. However, transitions only depending on environment players must actually be taken if possible.

So, in Petri games the tokens carry the information about their causal past as they flow through the net. In *Petri games with transits* the refinement of this flow relation defines the second layer of the information flow. At this level, we define the correctness properties of the system. For Petri games with transits, we increase the expressiveness of the specification language. Rather than forbidding the players to ever reach a bad place by themselves, we base the new objectives on the data flow of the processes. For that we reuse the concepts introduced in Chap. 4. This means, we refine the token flow relation with transits (represented by the colored augmentations of the black arrows) and consider the data flow chains of the processes induced by the transits. For example

in Fig. 9.1, we see that when transition i fires, a new data flow is generated (depicted by the red dot-dashed arrow) and is passed on by the transitions (depicted by the solid blue and dashed orange arrows). The new objectives allow us to reason about *all* flow chains of the processes (the *universal* conditions) and to reason about the existence of *one* flow chain of the processes (the *existential* conditions). So, for the example, we can require that all data flow chains should avoid reaching any of the bad places or that just one safe flow chain suffices to win a play. The new specification allows for two kinds of winning conditions: the *place-based* winning conditions, i.e., existential and universal safety, reachability, Büchi, co-Büchi, and parity conditions based on the places of the data flow in the net, and the *local Flow-LTL* winning condition that allows us to define existential and universal temporal requirements on the data flow with respect to the places and transitions of the net. In the following, we consider two examples for introducing these conditions.

9.1 Manufacturing

For the place-based winning conditions, take the simplified example from the world of manufacturing depicted in Fig. 9.1 as a Petri game with transits. A robot, represented by the token initially residing in place I, has a milling and a drilling tool at its disposal (represented by the place M or by the place D, respectively). When firing transition i, a new flow chain is generated which, along with the robot, ends up in place R. Thus, it exists a robot with a fresh flow chain knowing nothing about the system other than it started in place I, used transition i, and is now in place R. With this information, the robot cannot make an informed decision and must randomly start either milling (with transition m to place M) or drilling (with transition d to place D). The system must win against all behavior of the environment. This means that the system must also win in case the environment decides to destroy exactly the tool the robot is currently using (transition d_m to place D_M or transition d_d to place D_D). Then, either only transition c_m or only transition c_d is available for the robot. Hence, the corresponding tool is destroyed and the robot and the data flow end up in a bad place $\bot_M$ or $\bot_D$, respectively. Thus, the system cannot win a *universal* safety objective because there exists already a violating chain. However, for the *existential* safety objective, the robot can recover from the failure (by taking transition r_m or transition r_d) and start in this or any later round working with the intact tool. Since the robot took transition c_m or c_d, i.e., communicated with the environment, it learned which tool is malfunctioning and thus, can make an informed decision in any later round.

This memory model with its intricate *causal* dependencies (and independencies) are naturally represented by the *unfolding* [Eng91; EH08] of the underlying Petri net $\mathcal{N}$. An unfolding represents the behavior of a Petri net by unrolling each loop of the execution and introducing copies of a place $p \in \mathcal{P}$ for each join of transitions in p. Hence, each transition of the unfolding represents a unique *instance* of a transition $t \in \mathcal{T}$ during an execution. The unfolding exhibits concurrency, causal dependency, and nondeterminism (forward branching of places) of the unique occurrences of the transitions in $\mathcal{N}$ during

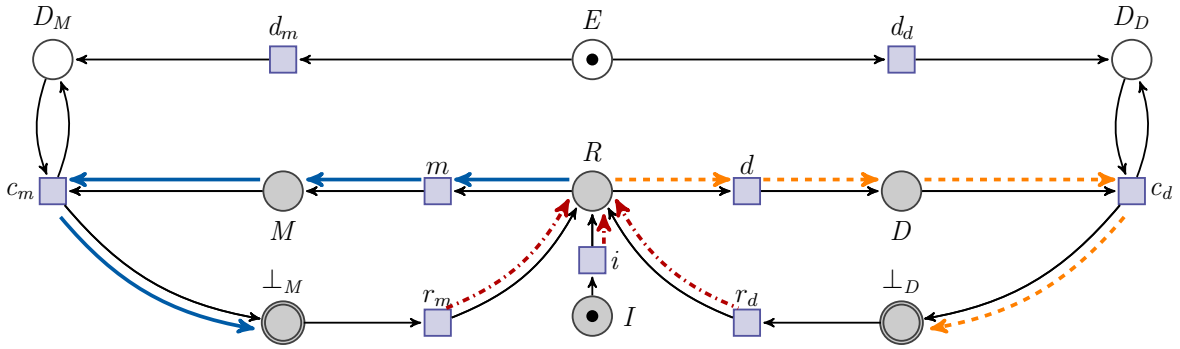


Fig. 9.1: *A simple Petri game with transits representing a manufacturing example with a robot R having a drilling tool D and a milling tool M at its disposal. A hostile environment E can destroy either one of these tools. Drilling or milling with a malfunctioning tool results in a bad situation $\bot_X$ for $X \in \{M, D\}$. For the* universal safety *condition* no *winning strategy exists. However, for the* existential safety *condition, the system can recover from a bad situation and can afterwards use the knowledge about the tools' states to select the intact tool.*

all possible executions. The notion of an unfolding is lifted to Petri games with transits by keeping the distinction between environment and system, and by transferring the decoration of the special places like bad places and the transits to the unfolding.

In Fig. 9.2 an excerpt of the unfolding of the Petri game with transits $\mathcal{G}$ depicted in Fig. 9.1 is shown. We can see that in the unfolding there are three copies R, R' and R'' of the place R of the Petri game with transits $\mathcal{G}$. In R we know nothing about the environment. In R' we have learned that the environment destroyed the milling tool, and in R'' the causal memory yields that the environment destroyed the drilling tool. Both pieces of information are available since transition c_m or c_d is in the past of R' or R'', respectively. The system can decide differently in each of these system places.

A *strategy* is a local controller for each system player which only decides on its current view and available information about the whole system. A strategy can be obtained by removing system-controlled branches of the unfolding. That is, transitions and their complete future are removed (obeying certain rules) which are considered as not being taken by the system. Transitions only dependent on environment players are not allowed to be removed. A strategy only allows nondeterminism for the environment players. The system must deterministically react to all possible actions of the environment. Resolving also the environment's nondeterminism results in a single concurrent execution of the system. In this setting, such concurrent runs of the Petri net are called *plays*.

In Fig. 9.2 a strategy σ is marked by the blue shaded area. This strategy contains only two maximal plays. A play is maximal in the sense that whenever it is possible to extend the play in the unfolding, it is indeed extended. Play π_1 is depicted by the red shaded area. For all safety conditions (universal and existential) this play is winning because the only flow chain $\xi_1 = i, R, m, M$ contains no bad place. However, for play π_2 (depicted by the green shaded area) there is the flow chain $\xi_2 = i, R, m, M, c_m, \bot_M$ reaching a bad

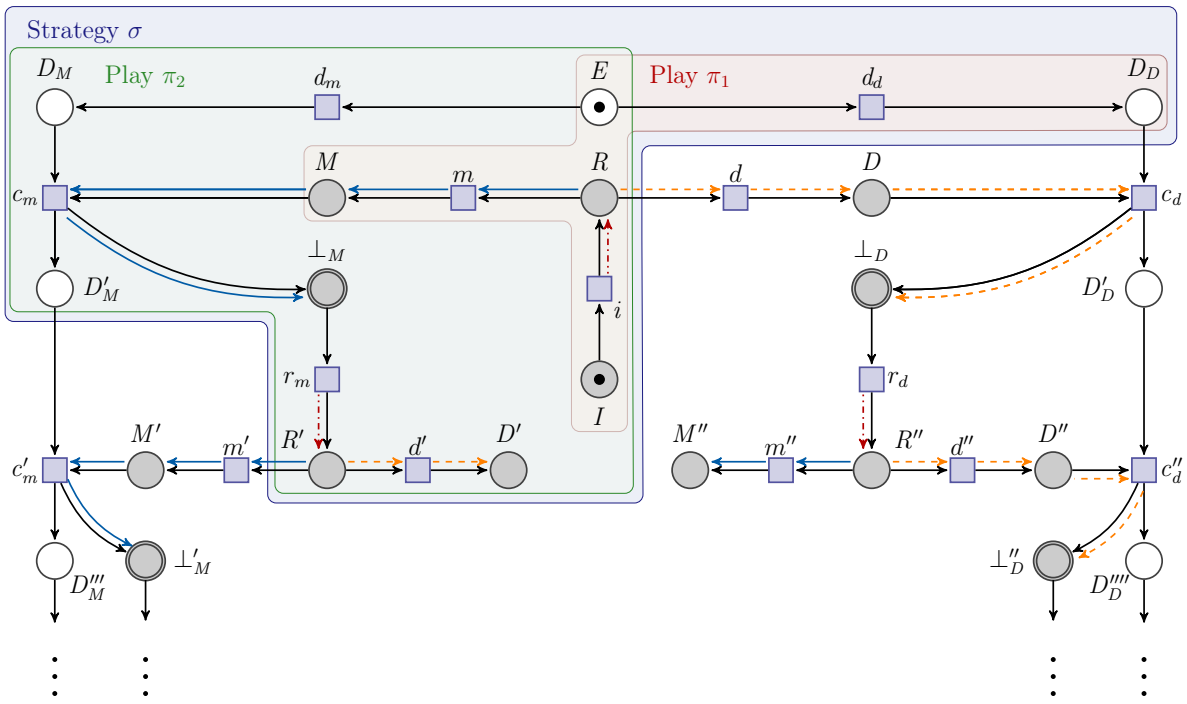


Fig. 9.2: *An excerpt of an unfolding of the Petri game with transits depicted in Fig. 9.1 with plays π_1 and π_2, and a winning strategy σ for the system players for the existential safety condition. Play π_1 is winning because no bad place occurs. Play π_2 is winning because even though the chain following the blue solid chain arrows reaches a bad place, there exists the chain starting with transition r_m staying only on safe places.*

place. Thus, play π_2 is not winning the universal safety objective. However, since play π_2 also contains the flow chain $\xi_3 = r_m, R', d', D'$ the play π_2 is winning the existential safety condition. Hence, σ is a winning strategy for the system players with respect to the existential safety condition since all maximal plays are winning. In Chap. 12 we provide a solving algorithm for generating such winning strategies of the system players automatically. The formal definitions of these concepts and such place-based winning conditions are given in Chap. 11. We consider the local Flow-LTL winning condition in a slightly more involved example of a parcel delivery system in the next section.

9.2 Parcel Delivery System

For the local Flow-LTL winning condition, take an example of a parcel delivery system with autonomous drones delivering an unbounded number of parcels to distributed locations. There is no central control unit that is constantly connected to the drones. During synchronizations, the drones communicate their causal past. However, the parcels may or may not be passed on. The goal of the drones is to deliver all parcels to the correct location, although some drones may malfunction unpredictably. Furthermore, no location should be favored by any drone. Figure 9.3 shows a formalization of this scenario

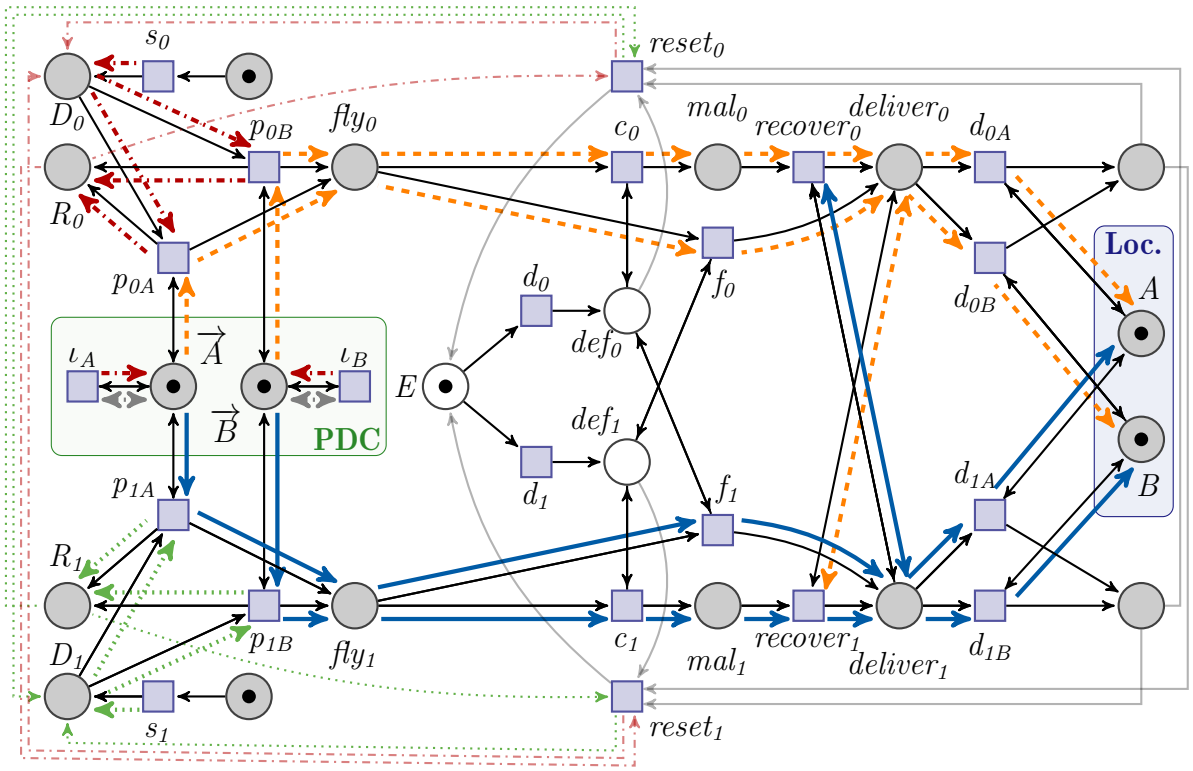


Fig. 9.3: *An example for the synthesis of asynchronous distributed systems modeled with a Petri game with transits. It represents a parcel delivery scenario with two locations A and B, two drones starting in D_0 and D_1, and a hostile environment (initially residing in E) recurrently selecting a malfunctioning drone, which eventually has to be recovered by the other. The unbounded number of parcels arriving in location $\overrightarrow{A}$ and $\overrightarrow{B}$ of the parcel distribution center should be delivered by the drones to their respective location A and B, and no location should be favored by any drone.*

for two drones and two locations with a Petri game with transits. The possible behavior of the first drone is depicted in the upper part of the figure following the red dot-dashed and the orange dashed arrows. Analogously, the possible behavior of the second drone is depicted in the lower part of the figure following the green dotted and the blue solid arrows. The parcel delivery center (PDC) is depicted in the middle at the left, the locations (Loc.) in the middle at the right, and the environment behavior deciding on the malfunctioning drone is depicted in the center of the figure. In the upper left respective lower left of the figure, the two drones are started with a fresh data flow via transitions s_0 and s_1 (indicated by the red dot-dashed and the green dotted arrow, respectively) as they move to the place D_0 respective D_1. As in the previous example, these transitions transit the data flow along *moving* control tokens. However, this changes for modeling the unbounded number of parcels. With transition ι_A and ι_B, new parcels may arrive in the parcel distribution center at any time (indicated by the red dot-dashed arrows). These transitions transit the data flow along *persistent* control tokens to allow for the

unbounded number of parcels to arrive at any given time. Remember that the coloring of the arrows only have a local significance to the transitions, i.e., the red dot-dashed arrows described so far are completely unrelated. The selection of a malfunctioning drone is modeled by a hostile environment process represented by the token initially residing in place E and the two environment transitions d_0 and d_1 deciding which drone is malfunctioning. So a drone $i \in \{0,1\}$ can pick up a package for location $L \in \{A,B\}$ via transition p_{iL} and deposit a receipt on place R_i. If the drone has no defect, it can fly towards the target locations via transition f_i and may deliver the package to some location $L' \in \{A,B\}$ via transition $d_{iL'}$. Previously, however, it could also first recover the defective drone via transition $recover_j$, for $j = i + 1 \; mod \; 2$.

As a correctness requirement for this scenario we request that a) all parcels should be delivered to the *correct* location and b) both drones behave *fairly* in the sense that they do not favor any location. The formulas

$$cor = \mathbb{A}\,(\bigwedge_{L\in\{A,B\}} (\overrightarrow{L} \rightarrow \Diamond L)) \quad \text{and} \quad fair = \bigwedge_{i\in\{0,1\}} (\mathbb{E}\,(\Box\Diamond D_i \wedge \Box\Diamond p_{iA} \wedge \Box\Diamond p_{iB}))$$

formalize these properties in local Flow-LTL. Formula *cor* requires that *all* ($\mathbb{A}$) data flows starting in place $\overrightarrow{L}$ must *eventually* ($\Diamond$) reach place $L \in \{A,B\}$ and the formula *fair* requires that for each drone $i \in \{0,1\}$ there *exists* ($\mathbb{E}$) a data flow that *infinitely often* ($\Box\Diamond$) visits place D_i, infinitely often uses transition p_{iA}, and infinitely often uses transition p_{iB}. In order to have anything to do at all, we additionally require that an unbounded number of parcels destined to each location indeed enter the parcel distribution center: $in = \bigwedge_{L\in\{A,B\}}(\mathbb{E}\,\Box\Diamond \iota_L)$.

A winning strategy for drone $i \in \{0,1\}$ satisfying $\varphi = cor \wedge fair \wedge in$ is to take the parcels destined for location A via transition p_{iA} and either send a distress signal via transition c_i or fly towards the target locations via transition f_i (depending on the environment's decision). Suppose the environment selected the other drone j, with $j = i + 1 \; mod \; 2$, to be malfunctioning. By synchronizing with the token in place def_j, the intact drone i learns of the defect of drone j and decides to first recover j and its parcel via transition $recover_j$. Due to the causal memory, drone i can remember taking transition p_{iA} and can decide for the correct location via transition d_{iA}. In the other case, drone i gets recovered and, for having a smaller scenario, delivers the package correctly by itself. However, due to transition $recover_i$, drone j also knows the destined location of the parcel of drone i and, in another scenario, could also deliver this parcel on its own. After a delivery, the drones fly back to the parcel distribution center via transitions $reset_D$, for $D \in \{0,1\}$. Note that the lightening of the arrows for transition $reset_D$ has no semantics and additional black arrows identical to the colored arrows are omitted for the sake of clarity. In the next round drone i takes the parcels destined for location B and repeats these choices alternately.

In Chap. 11 we provide the syntax and semantics for local Flow-LTL and make these concepts formal.

10 Models and Objectives

This chapter serves to recapitulate established concepts on which the synthesis approach for distributed asynchronous systems is based. In particular, in Sec. 10.2 we give an introduction to Petri games [FO14; FO17], the model that is extended in Chap. 11 to allow for enriched winning objectives, like universal and existential conditions. Furthermore, the solving algorithm of the synthesis approach for Petri games with transits and local objectives, presented in Chap. 12, consists predominantly of a reduction to a synthesis problem in a two-player infinite game over a finite graph. The introductory concepts for infinite games [BL69; GTW02; AG11; BCJ18] are presented in Sec. 10.1.

10.1 Infinite Games

In this section we consider two-player games played on a finite graph with an infinite duration and complete information [McN93; GTW02; AG11; BCJ18]. A state of the game is a vertex in the graph. The graph is called the game arena. Each vertex is uniquely associated to either player 0 or player 1. The game proceeds in rounds. In each round the player associated to the current vertex chooses the successor vertex with respect to the edges of the graph. The player can take the decision for the next vertex completely informed about the game arena and all turns taken so far. Taking turns for an infinite number of rounds results in a play of the players on the game arena. Thus, a play is an infinite path through the arena.

▶ **Definition 45 (Two-Player Game, Arena, and Play).** A *game arena* is a finite graph $A = (V, V_0, V_1, E)$ consisting of

- a finite set of *states* V partitioned into two disjoint sets of
- *player 0's states* V_0 and of
- *player 1's states* V_1, and
- the *edge relation* $E \subseteq V \times V$.

For a simplified notion we assume that all states have at least one outgoing edge.

A *two-player game* $G = (A, \texttt{WIN})$ consists of a game arena A and a set $\texttt{WIN} \subseteq V^\omega$ of infinite sequences of states of the arena, called the *winning condition*, or *winning objective*.

A *play* π on the arena A is an infinite sequence $\pi = v_0 v_1 v_2 \cdots \in V^\omega$ of states $v_i \in V$ that are related according to E, i.e., for all $i \in \mathbb{N}$ holds $(v_i, v_{i+1}) \in E$. A play π is

winning for player 0 w.r.t. to a winning condition $\texttt{WIN}$ iff $\pi \in \texttt{WIN}$ holds. Otherwise player 1 wins π. ◀

In the game-based synthesis approach for reactive systems [BL69], *player 0* represents the *system*, i.e., the controllable behavior, and *player 1* represents the *environment*, i.e., the uncontrollable behavior.

In Definition 16 we already recall some sets of infinite words used as acceptance conditions for the automata in Part I. We now extend and recall these sets for the winning conditions of the games and the acceptance conditions of the automata of this part.

▶ **Definition 46 (Winning Conditions).** Given an alphabet Σ, usually consisting of the states of the games or the automata. We define a function $\texttt{Occ} : \Sigma^\omega \to 2^\Sigma$ with $\texttt{Occ}(w) = \{a \in \Sigma \mid \exists m \in \mathbb{N} : w_m = a\}$, for collecting the letters *occurring* in w and the function $\texttt{Inf} : \Sigma^\omega \to 2^\Sigma$ with $\texttt{Inf}(w) = \{a \in \Sigma \mid \forall n \in \mathbb{N} : \exists m \geq n : w_m = a\}$ collecting all letters which *occur infinitely often* in an infinite word w over the alphabet Σ. For a finite set $W \subseteq \Sigma$ we define the

Safety condition: $\texttt{SAFE}(W) = \{w \in \Sigma^\omega \mid \texttt{Occ}(w) \cap W = \emptyset\}$ collecting all infinite words that do not visit any state of W,

Reachability condition: $\texttt{REACH}(W) = \{w \in \Sigma^\omega \mid \texttt{Occ}(w) \cap W \neq \emptyset\}$ collecting all infinite words that visit some state of W,

Büchi condition: $\texttt{BUCHI}(W) = \{w \in \Sigma^\omega \mid \texttt{Inf}(w) \cap W \neq \emptyset\}$ collecting all infinite words that visit some state of W infinitely often, and the

Co-Büchi condition: $\texttt{COBUCHI}(W) = \{w \in \Sigma^\omega \mid \texttt{Inf}(w) \cap W = \emptyset\}$ collecting all infinite words that visit none of the states of W infinitely often.

For a parity function $\Omega : \Sigma \to \mathbb{N}$ we define the

Parity condition: $\texttt{PARITY}(\Omega) = \{w \in \Sigma^\omega \mid \texttt{min}\{\Omega(\texttt{Inf}(w))\} \texttt{ mod } 2 = 0\}$ collecting all infinite words that have an even number as minimum for the parity of the states visited infinitely often.

We often call two-player games by their winning condition, e.g., we call the two-player game $G = (A, \texttt{PARITY}(\Omega))$ a *parity game*. For two conditions $\texttt{WIN}_1 \subset \Sigma^\omega$ and $\texttt{WIN}_2 \subset \Sigma^\omega$, we define the *conjunctive condition* by $\texttt{CONJ}(\texttt{WIN}_1, \texttt{WIN}_2) = \texttt{WIN}_1 \cap \texttt{WIN}_2$. ◀

A strategy is a player's function which determines their next move in a two-player game. In the game-based synthesis approach player 0's winning strategy can be seen as a correct-by-construction implementation of the controllers for the system of the encoded synthesis problem.

▶ **Definition 47 (Two-Player Game Strategy).** A *strategy* $\sigma : V^*V_i \to V$ of player i, for $i \in \{0,1\}$, of a two-player game $G = (A, \mathtt{WIN})$ with arena $A = (V, V_0, V_1, E, I)$, maps each sequence of states ending in a state of player i to some successor state according to E:

$$\sigma(wv) = v' \implies (v, v') \in E \text{ for all } w \in V^* \text{ and } v \in V_i.$$

A strategy σ is called *positional* (or *memoryless*) iff the decision of the player only depends on the last state of the input sequence, i.e., $\sigma(wv) = \sigma(v)$ for all $w \in V^*$ and $v \in V$. A play $\pi = v_0v_1v_2 \cdots \in V^\omega$ of G *conforms* to a strategy σ iff for all prefixes $wv_iv_{i+1} \in V^*V_0V$ of π holds $\sigma(wv_i) = v_{i+1}$ for $i \in \mathbb{N}$.

A strategy σ of player i, for $i \in \{0,1\}$, of G is a *winning* strategy of player i for a state v_0 iff each play π starting in v_0 and conforming to σ is winning for player i. ◀

A *winning region* for player i, denoted $\mathtt{W}_i$, is the set of all states $v \in V$ from which player i has a winning strategy, for $i \in \{0,1\}$. We call a two-player game *determined* iff from each state either player 0 or player 1 has a winning strategy, i.e., $V = \mathtt{W}_0 \uplus \mathtt{W}_1$. This means there are no draws in determined games.

Classical algorithms for solving such two-player games are based on fixpoint computations allowing a symbolic calculation of the winning set and memoryless strategies exist for both players (see for example [Jur11; BCJ18]). For safety and reachability games, memoryless winning strategies can be computed in $\mathcal{O}(|E|)$ time. For Büchi and co-Büchi games, this can be done in $\mathcal{O}(|E| \cdot |V|)$ time. For improvements to the classical algorithms, [BCJ18] points to [CJH03; CHP08; CH12; CH14].

According to [Küs01], the sufficiency of memoryless strategies for parity games was first proved independently by Emerson and Jutla [EJ91], and by Mostowski [Mos91].

▶ **Theorem 6 (Memoryless Determinacy of Parity Games [EJ91; Mos91]).** *For a parity function Ω, a two-player game $G = (A, \mathtt{PARITY}(\Omega))$ with an arena $A = (V, V_0, V_1, E)$ is determined with positional strategies.* ◀

It is shown by Emerson, Jutla, and Sistla that the problem of determining the winning player of a two-player parity game is in NP∩co-NP [EJS93; EJS01]. This upper-bound is improved to UP∩co-UP by Jurdzinski [Jur98]. A rich body of work exists for introducing new algorithms for solving parity games and for improving their time complexities. For example [Kla01; Jur11; BW18; BCJ18] present an overview of some different approaches and results. However, the problem of solving two-player parity games in polynomial time is still open [BCJ18], recent work [CJKL+17] presents a quasipolynomial time algorithm.

10.2 Petri Games

In this section we formally introduce the model Petri games [FO14; FO17] for the synthesis of distributed systems. An intuitive description as part of the example for Petri games with transits is already given in Chap. 9. We recall the example and represent the corresponding Petri game in Fig. 10.1. Syntactically, Petri games extend standard Petri nets (cp. Sec. 3.1) by identifying some places of the net as *bad* and by dividing the

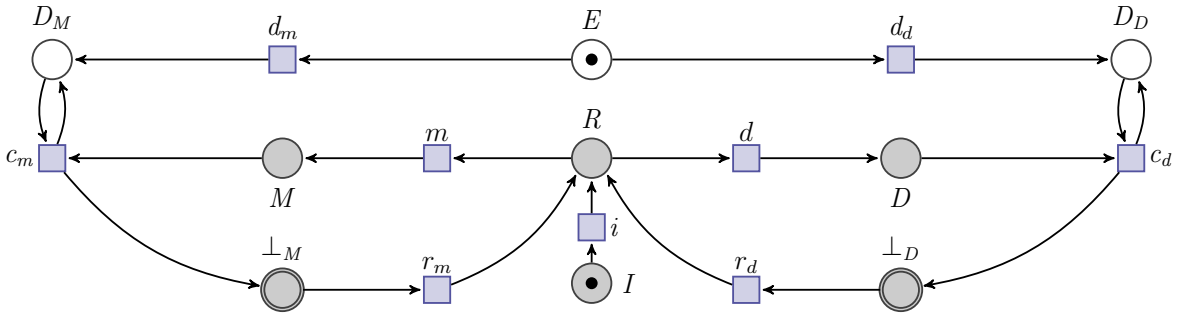


Fig. 10.1: *A Petri game $\mathcal{G} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, In, \mathcal{B})$ with one environment player initially residing in place E and one system player initially residing in place I, with the system places $\mathcal{P}_S = \{I, R, M, D, \bot_M, \bot_D\}$, the environment places $\mathcal{P}_E = \{E, D_D, D_M\}$, and the bad places $\mathcal{B} = \{\bot_M, \bot_D\}$. The Petri game $\mathcal{G}$ is the Petri game part of the Petri game with transits depicted in Fig. 9.1.*

places into *environment* places (depicted as white circles) and *system* places (depicted as gray circles).

▶ **Definition 48 (Petri Game [FO14]).** A *Petri game* is a six-tuple $\mathcal{G} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, In, \mathcal{B})$ that extends a finite Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ by partitioning the set of places $\mathcal{P} = \mathcal{P}_S \uplus \mathcal{P}_E$ into

- a set $\mathcal{P}_S$ of *system places* and
- a set $\mathcal{P}_E$ of *environment places*.

Additionally, $\mathcal{G}$ contains

- the set $\mathcal{B} \subseteq \mathcal{P}$ of *bad places*

to define a local *safety* objective, i.e., all processes have to avoid these places to win the game.

We call $\mathcal{N}$ the *underlying* Petri net, transitions $t \in \mathcal{T}$ with $pre^{\mathcal{N}}(t) \subseteq \mathcal{P}_S$ *pure system transitions* or just *system transitions*, and all other transitions *environment transitions*. Transitions with $pre^{\mathcal{N}}(t) \subseteq \mathcal{P}_E$ are called *pure environment transitions*. ◀

The *players* of a Petri game $\mathcal{G}$ are the tokens of the underlying Petri net $\mathcal{N}$. A token residing on a system place belong to the team of *system players*, whereas a token residing on an environment place belong to the team of *environment players*. The players play a Petri game by firing the transitions (depicted as blue squares) of the underlying Petri net.

In Petri games, the players remember their own *causal history*, i.e., the places and transitions they formerly used to reach the current place. A player does not know anything about another player as long as they reside in concurrent parts of the net. While taking a joint transition the players exchange all of their knowledge. When firing a joint transition we say that the participating players *synchronize*. This memory model

is naturally represented by the unfolding of the underlying net. The solid, together with the opaque parts of Fig. 10.2 represent an excerpt of an *unfolding* of the Petri game depicted in Fig. 10.1. The terms for unfolding, branching process, and run of a Petri game are defined by the terms of the underlying Petri net. See Sec. 3.1.2 for their formal definitions. We lift the distribution of the players to each of these Petri nets, e.g., $\mathscr{P}_S^U = \{p \in \mathscr{P}^U \mid \lambda^U(p) \in \mathscr{P}_S\}$ for an unfolding $\beta = (\mathscr{N}^U, \lambda^U)$.

The team of system players cooperate to achieve their common goal. This means, they try to ensure that no player (whether system or environment player) ever reaches a bad place. Based on their currently available information about the whole system, each player can decide which of their transitions are allowed for the next move. Since the environment players are uncontrollable, pure environment transitions cannot be restricted. This is resembled by the strategy of the system players. A strategy is a local controller for each system player that decides which transitions are allowed to be taken. The solid elements of Fig. 10.2 represent a strategy of the system player of the Petri game $\mathscr{G}$ presented in Fig. 10.1.

▶ **Definition 49 (Petri Game Strategy).** A *strategy* for the system players of a Petri game with an underlying Petri net $\mathscr{N}$ is a subprocess $\sigma = (\mathscr{N}^\sigma, \lambda^\sigma)$ of an unfolding $\beta = (\mathscr{N}^U, \lambda^U)$ of $\mathscr{N}$ satisfying the properties:

Justified refusal: $\forall t \in \mathscr{T}^U : (t \notin \mathscr{T}^\sigma \wedge pre^U(t) \subseteq \mathscr{P}^\sigma) \Rightarrow (\exists p \in pre^U(t) \cap \mathscr{P}_S^\sigma \; \forall t' \in post^U(p) : \lambda^U(t') = \lambda^U(t) \Rightarrow t' \notin \mathscr{T}^\sigma)$, i.e., if an instance t does not occur in the strategy σ, then σ uniformly forbids all instances t' of this transition from a place p in the precondition of t. This means that in every state of the system the strategy can only allow or disallow a transition of the original net and not a subset of the instances which are indistinguishable due to the system player's lack of knowledge. This condition also ensures that a strategy does not restrict any pure environment transition.

Determinism: $\forall p \in \mathscr{P}_S^\sigma, C \in \mathscr{R}(\mathscr{N}^\sigma) : p \in C \Rightarrow \exists^{\leq 1} t \in post^\sigma(p) : pre^\sigma(t) \subseteq C$, i.e., there is no situation in the strategy, where a system player allows two separate, enabled transitions.

For liveness properties the system itself is interested in proceeding. However, for a safety objective we have to ensure that the system does not directly win games by refusing to work. Hence, we consider strategies that are

Deadlock-avoiding: $\forall C \in \mathscr{R}(\mathscr{N}^\sigma) : (\exists t \in \mathscr{T}^U : pre^U(t) \subseteq C) \Rightarrow \exists t \in \mathscr{T}^\sigma : pre^\sigma(t) \subseteq C$, i.e., in every state of the strategy there *is* a continuation, whenever the system *can* proceed in the game $\mathscr{G}$. ◀

Note that we typically depict strategies together with the distribution of the places in terms of their team affiliation (system vs. environment). This distribution is directly induced by the Petri game.

A *play* π is an initial concurrent run $\pi = (\mathscr{N}^R, \rho)$ of the Petri game $\mathscr{G}$ and up to isomorphism a subprocess of the unfolding. Thus, where a strategy resolved the nondeterminism of the system players, a play further resolves the nondeterminism of the

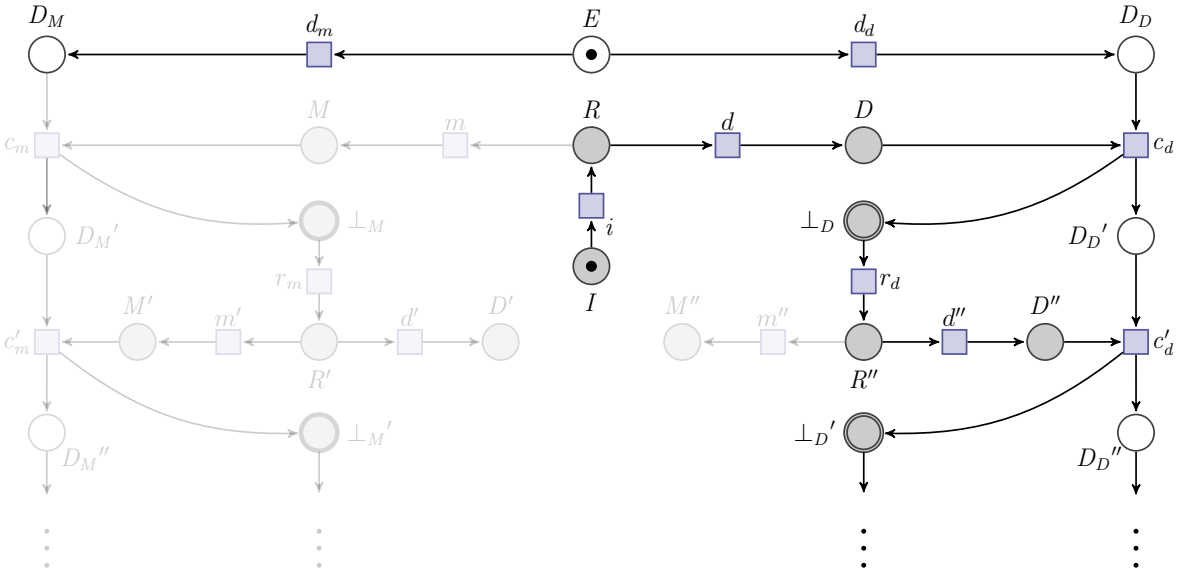


Fig. 10.2: *The solid elements represent an excerpt of a strategy of the system players for the Petri game $\mathcal{G}$ depicted in Fig. 10.1. Due to the solid bad places, this strategy is not winning for the system players. Additionally including the opaque elements constitutes an excerpt of an unfolding of $\mathcal{G}$.*

environment players. Two plays are depicted in Fig. 9.2. A play π *conforms* to a strategy σ iff π is a subprocess of σ. A play π is *winning* for the system players, iff $\mathscr{P}^R \cap \mathscr{B}^R = \emptyset$, with $\mathscr{B}^R = \{p \in \mathscr{P}^R \mid \rho(p) \in \mathscr{B}\}$. Otherwise the environment players win. A strategy σ is *winning*, iff all plays conforming to σ are winning.

In general, we are only interested in *maximal* plays of a strategy, this means we consider plays that continue as long as the strategy allows to continue. Formally, a play π conforming to a strategy σ is called *maximal* iff there is no play π' conforming to σ such that $\pi \sqsubseteq \pi'$ and $\pi \neq \pi'$.

▶ **Example 14.** The solid elements depicted in Fig. 10.2 constitute a deadlock-avoiding strategy $\sigma = (\mathcal{N}^\sigma, \lambda^\sigma)$ of the Petri game $\mathcal{G}$ shown in Fig. 10.1. Due to the deadlock-avoiding constraint of the strategy, the system player cannot stop in any of the depicted places to avoid reaching the bad place. The pure environment transitions d_m and d_d with $\lambda^\sigma(d_m) = d_m$ and $\lambda^\sigma(d_d) = d_d$ must appear in every strategy due to the justified refusal constraint. In place $R \in \mathscr{P}_S^\sigma$ with $\lambda^\sigma(R) = R$, a strategy has to decide whether to take transition m or transition d, because the determinism constraint of σ forbids to allow both. The strategy σ decides to take transition d, which means the bad place $\bot_D$ with $\lambda^\sigma(\bot_D) \in \mathscr{B}$ cannot be avoided if the environment decides to take transition d_d. Similarly, taking transition m results in the bad place $\bot_M \in \mathscr{B}^\sigma$. Thus, there is no deadlock-avoiding winning strategy for the system players of $\mathcal{G}$. ◀

In [FO14] Finkbeiner and Olderog show that the realizability problem for unbounded Petri games is in general undecidable due to the undecidability of VASS (Vector Addition

Systems with States) games [ABd03]. However, for Petri games with one environment player, a bounded number of system players, and a local safety objective, the problem is EXPTIME-complete [FO14]. The same complexity result applies for deciding the existence of a strategy for a bounded number of environment players and one system player with bad markings as global safety objective (i.e., avoiding bad markings) [FG17]. Furthermore, it is decidable whether there is a strategy for Petri games with a bounded number of system players and at most one environment player and a global safety objective [FGHO22].

11 Petri Games With Transits

In this section we introduce the new model *Petri games with transits* for the synthesis of distributed systems with local objectives. On the one hand, this model is based on Petri nets with transits [FGHO19a], which refine the flow relation of a Petri net to define the local objectives and, on the other hand, it is based on Petri games [FO14; FO17] to obtain the game semantics for the synthesis. An example is depicted in Fig. 9.1 on page 141. The graphical notation is the union of the graphical notations for Petri nets with transits (see Chap. 4) and for Petri games (see Sec. 10.2).

We enrich a standard Petri game with the notion of transits such that the underlying Petri net of the Petri game is a Petri net with transits and upgrade the safety winning condition to more general conditions.

▶ **Definition 50 (Petri Games with Transits).** The *arena* of a Petri game with transits is a six-tuple $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ with an underlying Petri net with transits $\mathscr{N}_\mathscr{A} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ that divides the places $\mathscr{P}$ into the set of *environment places* and *system places*, i.e., $\mathscr{P} = \mathscr{P}_E \uplus \mathscr{P}_S$, as for standard Petri games. We have the same notions for *players*, *plays*, *strategies*, etc. as before (cp. Sec. 10.2). The set of all plays of an arena $\mathscr{A}$ is denoted by $\Pi(\mathscr{A})$.

A *Petri game with transits* $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$ consists of an arena $\mathscr{A}$ combined with a set of plays $\texttt{WIN} \subseteq \Pi(\mathscr{A})$, called the *winning condition* or *winning objective*. A play is won by the system players if it is an element of `WIN`. Otherwise it is won by the environment players. A *play* of the Petri game with transits is the play of the arena. The same applies for *strategies*. ◀

We consider two kinds of specification languages for the winning conditions of a Petri game with transits. First, the *place-based* winning conditions, i.e., a set of dedicated places $\mathscr{W} \subseteq \mathscr{P}_S \cup \mathscr{P}_E$ or a coloring function $\Omega : \mathscr{P}_S \cup \mathscr{P}_E \to \mathbb{N}$ that define the set of winning plays `WIN`. Second, we generalize some of these conditions by defining a temporal logic, called *local Flow*-LTL, based on the temporal logic Flow-LTL introduced in Sec. 6.1. As in Chap. 5, we opt for the *ingoing stuttering* semantics when desiring an infinite semantics for finite chains. This means finite chains are mapped to an infinite trace that stutters on the last place.

▶ **Definition 51 (Traces).** A *trace* $\widetilde{\sigma}(\xi)$ of a (finite or infinite) flow chain $\xi = t_0, p_0, t_1, p_1, \ldots$ of a run $\beta = (\mathscr{N}_\mathscr{A}^R, \rho)$ of an arena $\mathscr{A}$ is a mapping $\widetilde{\sigma}(\xi) : \mathbb{N} \to \{\{t, p\}, \{p\} \mid p \in \mathscr{P} \wedge t \in \mathscr{T}\}$ with

$$\widetilde{\sigma}(\xi)(i) = \{\rho(t_i), \rho(p_i)\} \text{ for all } i \in \mathbb{N}$$

if ξ is infinite and

$$\widetilde{\sigma}(\xi)(i) = \begin{cases} \{\rho(t_i), \rho(p_i)\} & \text{for } i \leq n \\ \{\rho(p_n)\} & \text{otherwise} \end{cases}$$

if $\xi = t_0, p_0, t_1, p_1, \ldots, t_n, p_n$ is finite.

The ith *suffix* of a trace $\widetilde{\sigma}$ is the trace $\widetilde{\sigma}^i : \mathbb{N} \to \{\{t, p\}, \{p\} \mid p \in \mathscr{P} \wedge t \in \mathscr{T}\}$ with $\widetilde{\sigma}^i(j) = \widetilde{\sigma}(i + j)$ for all $j \in \mathbb{N}$. We define the function $\mathtt{Occ}(\widetilde{\sigma}(\xi)) = \{p \in \mathscr{P} \mid \exists m \in \mathbb{N} : p \in \widetilde{\sigma}(\xi)(m)\}$ to collect all places occurring in the trace $\widetilde{\sigma}(\xi)$ and the function $\mathtt{Inf}(\widetilde{\sigma}(\xi)) = \{p \in \mathscr{P} \mid \forall n \in \mathbb{N} : \exists m \geq n : p \in \widetilde{\sigma}(\xi)(m)\}$ to collect all places occurring infinitely often in $\widetilde{\sigma}(\xi)$. ◀

For the *place-based* conditions, we generalize the winning condition of standard Petri games from demanding that all players have to play safe, i.e., avoid reaching a bad place, to requesting that the flow chains of the system satisfy a safety, a reachability, a Büchi, a co-Büchi, or a parity condition. On each of these conditions, we allow for a *universal* and an *existential* view, i.e., checking the property for all flows chains or checking the existence of such a flow chain. We use the ingoing stuttering semantics to obtain an infinite semantics for finite chains.

▶ **Definition 52 (Winning Conditions).** Given a Petri game with transits $\mathscr{G} = (\mathscr{A}, \mathtt{WIN})$ with an arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and an underlying Petri net with transits $\mathscr{N}_\mathscr{A} = (\mathscr{P}, \mathscr{T}, \mathscr{F}, \Upsilon, In)$. For a set $\mathcal{S} \subseteq \mathscr{P}$ and a parity function $\Omega : \mathscr{P} \to \mathbb{N}$ we define:

Safety:

- the *existential safety condition* $\exists\text{-}\mathtt{SAFE}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \exists \xi \in \Xi(\pi) : \mathtt{Occ}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} = \emptyset\}$, collecting all plays for which a flow chain *exists* that does not reach any bad place,
- the *universal safety condition* $\forall\text{-}\mathtt{SAFE}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \forall \xi \in \Xi(\pi) : \mathtt{Occ}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} = \emptyset\}$, collecting all plays for which *all* flow chains do not reach any bad place,

Reachability:

- the *existential reachability condition* $\exists\text{-}\mathtt{REACH}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \exists \xi \in \Xi(\pi) : \mathtt{Occ}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} \neq \emptyset\}$, collecting all plays for which a flow chain *exists* that reaches some special place,
- the *universal reachability condition* $\forall\text{-}\mathtt{REACH}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \forall \xi \in \Xi(\pi) : \mathtt{Occ}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} \neq \emptyset\}$, collecting all plays for which *all* flow chains reach some special place,

Büchi:

- the *existential Büchi condition* $\exists\text{-}\mathtt{BUCHI}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \exists \xi \in \Xi(\pi) : \mathtt{Inf}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} \neq \emptyset\}$, collecting all plays for which a flow chain *exists* that reaches some special place infinitely often,
- the *universal Büchi condition* $\forall\text{-}\mathtt{BUCHI}(\mathcal{S}) = \{\pi \in \Pi(\mathscr{G}) \mid \forall \xi \in \Xi(\pi) : \mathtt{Inf}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} \neq \emptyset\}$, collecting all plays for which *all* flow chains reach some special place infinitely often,

Co-Büchi:

- the *existential co-Büchi condition* $\exists\text{-}\mathtt{COBUCHI}(\mathcal{S}) = \{\pi \in \Pi(\mathcal{G}) \mid \exists \xi \in \Xi(\pi) : \mathtt{Inf}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} = \emptyset\}$, collecting all plays for which a flow chain *exists* where every infinitely occurring place is a safe place,
- the *universal co-Büchi condition* $\forall\text{-}\mathtt{COBUCHI}(\mathcal{S}) = \{\pi \in \Pi(\mathcal{G}) \mid \forall \xi \in \Xi(\pi) : \mathtt{Inf}(\widetilde{\sigma}(\xi)) \cap \mathcal{S} = \emptyset\}$, collecting all plays for which *all* flow chains only contain the safe places infinitely often,

Parity:

- the *existential parity condition* $\exists\text{-}\mathtt{PARITY}(\Omega) = \{\pi \in \Pi(\mathcal{G}) \mid \exists \xi \in \Xi(\pi) : \mathtt{min}\{\Omega(\mathtt{Inf}(\widetilde{\sigma}(\xi))\} \bmod 2 = 0\}$, collecting all plays for which a flow chain *exists* where the minimum of the colors assigned by Ω to the places occurring infinitely often is even,
- the *universal parity condition* $\forall\text{-}\mathtt{PARITY}(\Omega) = \{\pi \in \Pi(\mathcal{G}) \mid \forall \xi \in \Xi(\pi) : \mathtt{min}\{\Omega(\mathtt{Inf}(\widetilde{\sigma}(\xi))\} \bmod 2 = 0\}$, collecting all plays for which *all* flow chains where the minimum of the colors assigned by Ω to the places occurring infinitely often is even. ◀

The existential and universal winning conditions complement themselves. This means that for example a play that has a safe flow chain with respect to a set $\mathcal{S}$, cannot be part of the plays where all flow chain reach a place in $\mathcal{S}$. For that, we define the *dual* (denoted by $\widetilde{\cdot}$) of such a winning condition.

▶ **Definition 53 (The Dual of a Place-Based Winning Condition).** Let $\mathtt{WIN}$ be one of the place-based winning conditions defined in Definition 52. We define the dual $\widetilde{\mathtt{WIN}}$ of the condition by

$$\widetilde{\mathtt{WIN}} = \begin{cases} \widetilde{Q}\text{-}\mathtt{REACH}(\mathcal{S}) & \text{if } \mathtt{WIN} = Q\text{-}\mathtt{SAFE}(\mathcal{S}) \\ \widetilde{Q}\text{-}\mathtt{SAFE}(\mathcal{S}) & \text{if } \mathtt{WIN} = Q\text{-}\mathtt{REACH}(\mathcal{S}) \\ \widetilde{Q}\text{-}\mathtt{COBUCHI}(\mathcal{S}) & \text{if } \mathtt{WIN} = Q\text{-}\mathtt{BUCHI}(\mathcal{S}) \\ \widetilde{Q}\text{-}\mathtt{BUCHI}(\mathcal{S}) & \text{if } \mathtt{WIN} = Q\text{-}\mathtt{COBUCHI}(\mathcal{S}) \\ \widetilde{Q}\text{-}\mathtt{PARITY}(\Omega') & \text{if } \mathtt{WIN} = Q\text{-}\mathtt{PARITY}(\Omega) \end{cases}$$

with $Q \in \{\forall, \exists\}$ and for $\widetilde{\exists} = \forall$, $\widetilde{\forall} = \exists$, and $\Omega'(p) = \Omega(p) + 1$ for all $p \in \mathscr{P}$. ◀

It is easy to see that for each winning condition $\mathtt{WIN}$ the dual is exactly the *complement*, i.e., $\widetilde{\mathtt{WIN}} = \Pi(\mathcal{G}) \setminus \mathtt{WIN}$, and the dual function is *self-inverse*, i.e., $\widetilde{\widetilde{\mathtt{WIN}}} = \mathtt{WIN}$.

For *local Flow*-LTL we, on the one hand, limit Flow-LTL (see [FGHO19a] or Sec. 6.1) to its *local* fragment and, on the other hand, extend it with the existential view.

▶ **Definition 54 (Local Flow-LTL).** For an arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, \mathit{In})$ of a Petri game with transits, we define the *syntax* of a *local Flow-LTL* formula φ with atomic propositions $a \in \mathscr{P}_S \cup \mathscr{P}_E \cup \mathscr{T}$ by

$$\varphi ::= \mathbb{A}\,\psi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \quad \text{with} \quad \psi ::= a \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \bigcirc\psi \mid \psi_1\,\mathcal{U}\,\psi_2.$$

The *flow operator* $\mathbb{A}$ in φ (or $\mathbb{E}$ as abbreviation for $\mathbb{E}\,\psi = \neg\mathbb{A}\,\neg\psi$) is used to universally (or existentially) select the flow chains and standard LTL is used with ψ to reason about the selected flow chains.

The *semantics* of a play $\pi = (\mathcal{N}^R_{\mathscr{A}}, \rho)$ of $\mathscr{A}$ satisfying a local Flow-LTL formula φ is defined as follows:

$$
\begin{array}{lll}
\pi \models \mathbb{A}\,\psi & \text{iff} & \text{for all flow chains } \xi \text{ of } \pi:\ \widetilde{\sigma}(\xi) \models \psi \\
\pi \models \neg\varphi & \text{iff} & \text{not } \pi \models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2
\end{array}
$$

$$
\begin{array}{lll}
\widetilde{\sigma}(\xi) \models a & \text{iff} & a \in \widetilde{\sigma}(\xi)(0) \\
\widetilde{\sigma}(\xi) \models \neg\psi & \text{iff} & \text{not } \widetilde{\sigma}(\xi) \models \psi \\
\widetilde{\sigma}(\xi) \models \psi_1 \wedge \psi_2 & \text{iff} & \widetilde{\sigma}(\xi) \models \psi_1 \text{ and } \widetilde{\sigma}(\xi) \models \psi_2 \\
\widetilde{\sigma}(\xi) \models \bigcirc\,\psi & \text{iff} & \widetilde{\sigma}(\xi)^1 \models \psi \\
\widetilde{\sigma}(\xi) \models \psi_1\ \mathcal{U}\ \psi_2 & \text{iff} & \text{there exists a } j \geq 0 \text{ with } \widetilde{\sigma}(\xi)^j \models \psi_2 \text{ and} \\
 & & \text{for all } 0 \leq i < j \text{ the following holds: } \widetilde{\sigma}(\xi)^i \models \psi_1
\end{array}
$$

with local Flow-LTL formulas φ, φ_1, and φ_2, atomic propositions $a \in \mathscr{P}_S \cup \mathscr{P}_E \cup \mathscr{T}$, and LTL formulas ψ, ψ_1, and ψ_2. We say the local data flow of a play π of $\mathscr{A}$ *satisfies* a local Flow-LTL formula φ iff $\pi \models \varphi$.

We define the *local Flow-LTL condition* $\texttt{Flow-LTL}(\varphi) = \{\pi \in \Pi(\mathscr{G}) \mid \pi \models \varphi\}$ collecting all plays satisfying the local Flow-LTL formula. ◀

For a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$, the system players *win* a play $\pi = (\mathcal{N}^R_{\mathscr{A}}, \rho)$ of $\mathscr{G}$ iff $\pi \in \texttt{WIN}$. A strategy σ is *winning* for the system players w.r.t. the *safety conditions* iff σ is deadlock-avoiding and all plays conforming to σ are won by the system. A strategy σ is *winning* for the system players w.r.t. all *other conditions* iff all maximal plays conforming to σ are won by the system. Otherwise, the *environment players win*.

Note that the winning condition of a Petri game $\mathscr{G} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, In, \mathscr{B})$ from [FO14], requesting that all players never reach a bad place $b \in \mathscr{B}$, is a special case of the universal safety condition $\forall\text{-}\texttt{SAFE}(\mathcal{S})$ for a corresponding Petri game with transits $\mathscr{G}'$. For example, we can create $\mathscr{G}'$ from $\mathscr{G}$ by adding a single fresh initial place p' and a single fresh transition t. The transition t moves the initial token to all original initial places $p \in In$. and thereby starts a new chain in p. All other transitions have a one-to-one mapping of the ingoing and outgoing arcs regarding the transits and, if there are more outgoing than ingoing arcs, the remaining outgoing arcs are mapped to an arbitrary ingoing arc.

For the decision procedure in the next chapter, we restrict the Petri games with transits under consideration in two ways. On the one hand, we consider (as in [FO14; FO17]) Petri games with transits with *one environment player* and a bounded number of system players, i.e., to any point in time there is at most one environment player available. On the other hand, we consider Petri games with transits which does not

have any *mixed communication*, i.e., in any state, each system player can provide either a communication to the environment or to other system players, but not to both at the same time.

In each state, each system actor can offer communication either with the environment or with other system actors, but not with both at the same time.

▶ **Definition 55 (Mixed Communication).** For a Petri game with transits $\mathcal{G}$ with an arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$, a place $p \in \mathcal{P}_S$ allows *mixed communication* iff

$$\exists t_1, t_2 \in post(p) : \mathcal{P}_E \cap pre(t_1) \neq \emptyset \wedge pre(t_2) \subseteq \mathcal{P}_S.$$

A Petri game $\mathcal{G}$ has *mixed communication* iff any place $p \in \mathcal{P}_S$ allows mixed communication. ◀

In practical application, we can often easily resolve the mixed communication by executing the pure system behavior and the communication of the system with the environment one after the other. We denote the class of *safe* Petri games with transits without *mixed communication* and with *one* environment player and a *bounded* number of system players by $\mathcal{G}^{1,b}_{s,\circ|\circ}$.

Note that the synthesis problem is undecidable for the complete (meaning non-local) Flow-LTL logic, because reasoning about markings with LTL is allowed. In [MT02], Madhusudan and Thiagarajan introduce a minimal set of constraints for the controller synthesis in an asynchronous distributed setting with linear temporal specifications that make the synthesis problem decidable. The restriction R1 of these constraints demands a *robustness* of the specification in the sense that in case one linearization of an execution satisfies the specification, also all other linearizations have to satisfy the specification. This cannot hold for markings in a Petri net in general. See [TH96] for more details on temporal logics that are robust in this sense. Furthermore, [FGHO22] shows that for Petri games with one environment and two system players already a specification consisting of a combination of good and bad markings is undecidable.

12 Synthesis of Distributed Systems with Local Data Flows

In this section we present a procedure for deciding whether a safe Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$ with arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and one environment player, a bounded number of system players, and without any mixed communication has a winning strategy for the system players and create one if it exists. This procedure consists of four steps:

1. The *information flow game* $G(\mathscr{A}) = (A(\mathscr{A}), \texttt{BUCHI}(V_1))$ with arena $A(\mathscr{A}) = (V, V_0, V_1, E, I)$ is created to answer the question whether there exists a strategy for the system players in $\mathscr{G}$ with the general requirements *justified refusal*, *determinism*, and *deadlock-avoidance*. This game is a two-player Büchi game over a finite graph. The intricate causal memory model of $\mathscr{G}$ is reduced to a model with complete information by focusing on a scheduling where the moves of the single environment player are delayed as long as possible. We show that player 0 has a winning strategy in $G(\mathscr{A})$ iff the system players have a *strategy* in $\mathscr{G}$.

2. The deterministic *transit automaton* $\Lambda(\mathscr{G})$ is created corresponding to the local winning condition $\texttt{WIN}$ of $\mathscr{G}$. This automaton serves for checking whether a play π is winning, i.e., $\pi \in \texttt{WIN}$. We introduce three kinds of transit automata. One for the existential, one for the universal, and one for the local Flow-LTL specifications. The acceptance conditions and sizes of the automata depend on $\texttt{WIN}$. The automata track either the possibly infinite number of flow chains in the system in a finite manner or guess and track only the desired one. We show that $\pi \in \texttt{WIN}$ iff the trace of each covering firing sequences of π is accepted by $\Lambda(\mathscr{G})$.

3. The composition of the two-player Büchi game $G(\mathscr{A})$ and the deterministic automaton $\Lambda(\mathscr{G})$ yields the product game $\mathbb{G}(\mathscr{G})$. The winning condition of the resulting two-player game $\mathbb{G}(\mathscr{G})$ depends on the acceptance condition of $\Lambda(\mathscr{G})$. We show that player 0 has a winning strategy in $\mathbb{G}(\mathscr{G})$ iff the system players have a *winning strategy* in $\mathscr{G}$. Solving $\mathbb{G}(\mathscr{G})$ for player 0 with standard game solving algorithms results in a winning strategy $\sigma_{||}$ as long as it exists.

4. Creating a Petri net while traversing the strategy tree of $\sigma_{||}$ in breadth-first order yields the winning strategy σ for the system players in $\mathscr{G}$.

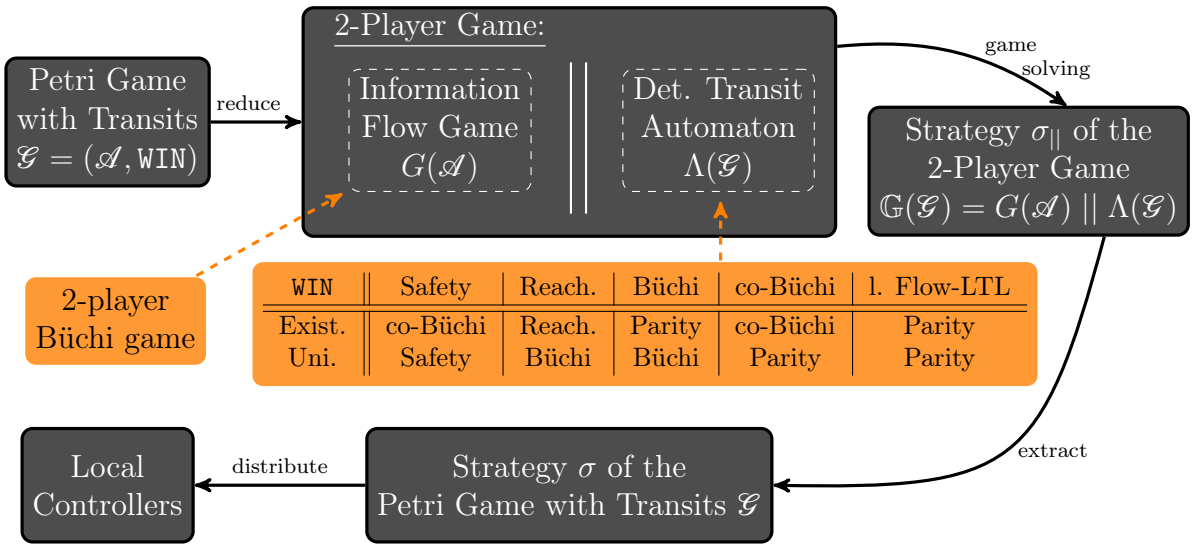


Fig. 12.1: *An overview of the decision procedure for Petri games with transits and local winning conditions. The Petri game with transits $\mathcal{G} = (\mathcal{A}, \mathtt{WIN})$ is initially reduced to the product of a two-player Büchi game $G(\mathcal{A})$, representing the justified refusal, determinism, and deadlock-avoidance properties of the strategy, and the deterministic transit automaton $\Lambda(\mathcal{G})$, representing the local winning property* WIN *of the data flow. The different winning conditions of $\mathcal{G}$ result in different acceptance conditions of $\Lambda(\mathcal{G})$. A strategy $\sigma_{||}$ for player 0 of the product game is obtained (if existent) by applying standard game solving techniques. This strategy is used to extract the winning strategy σ for the system players of $\mathcal{G}$. When the arena $\mathcal{A}$ is adequately annotated regarding which places belong to which process, σ can be easily distributed into the local controllers of the processes.*

Figure 12.1 gives an overview of this process. Section 12.1 corresponds to the first step, where we define the information flow game $G(\mathcal{A})$ and show its relevant properties. In Sec. 12.2 (corresponding to the second step), the existential, universal, and Flow-LTL transit automata are defined and their expediency is shown. Section 12.3 corresponds to the third step and introduces the product game $\mathbb{G}(\mathcal{G})$ and shows the complexity for solving Petri games with transits with local winning conditions belonging to the class $\mathcal{G}^{1,b}_{s,\circ|\circ}$. To make the reading experience more enjoyable, most proofs are swapped to Sec. 12.4 and only proof sketches are incorporated in the other sections.

12.1 Information Flow Game

For a Petri game with transits $\mathcal{G} = (\mathcal{A}, \mathtt{WIN})$ with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$, one environment player and a bounded number of system players, and without any mixed communication, we create a two-player Büchi game $G(\mathcal{A}) = (A(\mathcal{A}), \mathtt{BUCHI}(V_1))$, called the *information flow game*, with the game arena $A(\mathcal{A}) = (V, V_0, V_1, E, I)$. We show

that $\mathscr{G}$ has a *strategy* iff $G(\mathscr{A})$ has, and that the strategies can be transformed into one another. In Sec. 12.2 we define deterministic automata for the winning conditions and show in Sec. 12.3 that the parallel composition of such an automaton with the information flow game additionally maps the *winning* property of the strategies.

In the style of the two-player game presented in [FO17], the states of the information flow game are markings of the Petri game with transits which are enriched with additional information. Especially, each system player has a set of transitions (called *commitment set*) from which the next transition is selected. After every move, these sets are chosen by the system as early as possible to fix their next move. The key idea to mimick the causal memory of the Petri game with transits with the complete information in the two-player game is to delay the environment's moves until all system players cannot progress without the environment taking part in the next step (or will never interact with the environment again, called `type`-2 *behavior*). In [FO17], such situations are called *mcuts*. In an mcut, the environment's last position is communicated to each system player during their next move. By choosing the next commitment set directly after getting new information of the environment and these commitment sets stay fixed until the next communication with the environment (or a new one is directly chosen after moving and before the next environment step), in an mcut each system player already took their decision for their next move. So, despite directly revealing the environment move to *all* players, no decision of the system for a next move is taken overly informed. Note that here the restriction to one environment player is crucial to enable such a restricted environment delaying scheduling. The locality of our properties and the restriction to Petri games with transits without mixed communication ensure that no relevant behavior is missed by only investigating a subset of all schedulings.

To account for the more intricate winning conditions based on the local data flow, we employ two new techniques. First, we outsource the check of these conditions to separate deterministic automata to prove the general properties of the strategies of the Petri game with transits separately in the information flow game. This feature enables us to focus on the winning property for the automata and to not prove the strategy properties for each condition individually. In particular, this makes the framework very adaptable for developing further winning conditions. Second, we incorporate the pure system behavior (`type`-2 behavior) into the information flow game to provide winning conditions where the satisfaction is based on a dependency on features inside and outside the `type`-2 behavior. For this purpose, we introduce two concepts: a *generation* identifier for each system place in a marking of a state and a *round robin* identifier for each state. The *generation* identifier indicates the level of informedness this player (if of `type`-2) will ever have about the environment's movements. These identifiers are used to ensure that the causal memory model of the Petri game with transits is correctly mimicked in the two-player game with complete information. Only the correct choice of the generations allow the system players to win. The `type`-2 players are scheduled after every mcut with a *round robin procedure*. For that we additionally save in each state $(D, \curlyvee, t)$ with the round robin identifier $\curlyvee$ the position in this procedure and with t the transition (or the special symbol $\top$ or τ) which led into this state. The so-called *decision set* D is such a previously described enriched marking and is formally introduced in the next section.

12.1.1 States

The states in the information flow arena $A(\mathscr{A}) = (V, V_0, V_1, E, I)$ are three-tuples $(D, \curlyvee, \mathfrak{t}) \in V$ where the first component D is called a *decision set*.

▶ **Definition 56 (Decision Sets and Commitment Sets).** For a given a Petri game with transits with the arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and the underlying Petri net $\mathscr{N}$, the set of all possible *decision sets* is defined as $\mathscr{D} = 2^{\mathcal{D}}$ with

$$\begin{aligned}\mathcal{D} =& \{(p, 0, post(p)) \in \mathscr{P}_E \times \{0\} \times 2^{\mathscr{T}}\} \\ &\cup \{(p, i, C) \in \mathscr{P}_S \times \{0, \ldots, \mathtt{max}_S\} \times (2^{\mathscr{T}} \cup \{\top\}) \mid C \subseteq post(p) \vee C = \top\}\end{aligned} \tag{12.1}$$

where $\mathtt{max}_S = max\{|M \cap \mathscr{P}_S| \in \mathbb{N} \mid M \in \mathscr{R}(\mathscr{N})\}$ is the maximal number of system players simultaneously occurring in the game at any point in time.

For a a decision set $D \in \mathscr{D}$ an element $(p, i, C) \in D$ is called a *decision* and $C \in 2^{\mathscr{T}} \cup \{\top\}$ is called a *commitment set*. ◀

For a decision set $D \in \mathscr{D}$ a decision $(p, i, C) \in D$ denotes that a token resides on place $p \in \mathscr{P}$ and the corresponding player *commits* to use one of the transitions occurring in the commitment set for the next move if $C \subseteq \mathscr{T}$ or, if $C = \top$, has to decide for a commitment set before doing any further step. The generation of a `type`-2 typed place is saved with $i > 0$, and $i = 0$ denotes that p is not a `type`-2 typed place. We call those places `type`-1 typed. Transitions in a decision set that move only `type`-2 typed places are called `type`-2 *transitions* (or *of type* `type`-2) and those moving only `type`-1 typed places `type`-1 *transitions* (or *of type* `type`-1).

To retrieve the *marking of a decision set*, we define $\mathscr{M} : \mathscr{D} \to 2^{\mathscr{P}}$, with $D \mapsto \{p \in \mathscr{P} \mid (p, \cdot, \cdot) \in D\}$ to collect all places of a decision set. A transition $t \in \mathscr{T}$ is *enabled* in a decision set D, written $D\{t\rangle$, iff $pre(t) \subseteq \mathscr{M}(D)$. A transition $t \in \mathscr{T}$ is `type`-1 *enabled* in a decision set D, written $D\{t\rangle_1$, iff t is enabled in the to only `type`-1 typed places restricted marking of D, i.e, $pre(t) \subseteq \{p \in \mathscr{M}(D) \mid \exists C \in 2^{\mathscr{T}} : (p, 0, C) \in D\}$. A transition $t \in \mathscr{T}$ is `type`-2 *enabled* in a decision set D, written $D\{t\rangle_2$, iff t is enabled in the to a specific generation restricted marking of D, i.e., $\exists i \in \mathbb{N} \setminus \{0\} : pre(t) \subseteq \{p \in \mathscr{M}(D) \mid \exists C \in 2^{\mathscr{T}} : (p, i, C) \in D\}$. This means for t being `type`-2 enabled we need that all places in the preset belong to the same generation. Note that it suffices for a transition t to be `type`-1 or `type`-2 enabled that each place of the preset of t occurs in the decision set with the matching generation, as long as it has no $\top$ as commitment set. A transition $t \in \mathscr{T}$ is *chosen* in a decision set D, written $D(t\rangle$, iff there is a place of t's preset in $\mathscr{M}(D)$ and none of them disallows t, i.e., $pre(t) \cap \mathscr{M}(D) \neq \emptyset \wedge \forall (p, i, C) \in D : p \in pre(t) \implies t \in C$. A transition $t \in \mathscr{T}$ is *firable* (generally, `type`-1, or `type`-2) in a decision set D, written $D[t\rangle$ ($D[t\rangle_1$ or $D[t\rangle_2$, respectively), iff t is enabled (generally, `type`-1, or `type`-2) and chosen. This means D contains all places of the preset of t (with the same generation) and all of them have t in their commitment set. A decision set D is *nondeterministic* iff $\exists t_1, t_2 \in \mathscr{T} : t_1 \neq t_2 \wedge pre(t_1) \cap pre(t_2) \cap \mathscr{P}_S \neq \emptyset \wedge D[t_1\rangle \wedge D[t_2\rangle$. Otherwise a decision set is *deterministic*. We retrieve a *free generation* of a decision set D by $\mathtt{free}_G(D) = min\{i \in \mathbb{N} \mid (p, i, C) \notin D \wedge i \neq 0\}$.

The set of all decision sets which correspond to situations where the Petri game with transits *terminates* is defined by $\texttt{TERM} = \{D \in \mathscr{D} \mid \forall t \in \mathscr{T} : \neg \mathscr{M}(D)[t\rangle \wedge \forall (\cdot,\cdot,C) \in D : C \neq \top\}$. We define sets of decision sets which correspond to bad situations in the Petri game with transits, i.e., a *deadlock* occurred $\texttt{DL} = \{D \in \mathscr{D} \mid \exists t \in \mathscr{T} : \mathscr{M}(D)[t\rangle \wedge \forall t \in \mathscr{T} : \neg D[t\rangle\}$ or *nondeterminism* has been encountered $\texttt{NDET} = \{D \in \mathscr{D} \mid D \text{ is nondeterministic}\}$.

The states of the information flow arena are decision sets equipped with a round identifier $\text{ɾ} \in \{0, \ldots, |\mathscr{T}|\}$ and with the responsible action $\text{ŧ} \in \mathscr{T} \cup \{\top, \tau\}$ that led into the current state. The round identifier ɾ saves the id of the next transition for implementing a round robin scheduling for `type`-2 typed places. The identifier $\text{ɾ} = 0$ schedules environment transitions, the other values identify transitions in a unique order. The responsible action ŧ is either the fired transition $t \in \mathscr{T}$, $\top$ when a commitment set was resolved, or τ for looping behavior in terminating or bad situations.

▶ **Definition 57 (Information Flow Arena – States).** Let $\mathscr{O} = \{v \in \mathscr{D} \mid \forall p \in \mathscr{P} : |\{(p,\cdot,\cdot) \in v\}| \leq 1 \wedge |\{(p,\cdot,\cdot) \in v \mid p \in \mathscr{P}_E\}| = 1\} \times \{0, \ldots, |\mathscr{T}|\} \times (\mathscr{T} \cup \{\tau, \top\})$ be the set of *ordinary* states, i.e., the states corresponding to markings of *1-bounded* Petri games with transits with *one* environment player. We define

- the set of *all states*: $V = \mathscr{O} \cup \{v_↯\}$ with an *error* state $v_↯$, which is used for the non-winning loops of the game,
- the set of *player 1's states*:

$$V_1 = \{(D, \text{ɾ}, \text{ŧ}) \in V \mid D \in \texttt{TERM} \vee (\forall (p,i,C) \in D : C \neq \top \wedge \forall t \in \mathscr{T} : (\neg D[t\rangle_1 \vee \emptyset \neq \mathscr{M}(D) \cap \mathscr{P}_E \subseteq pre(t)))\},$$

  i.e., all terminating situations, or situations where the `type`-1 behavior of the system can only proceed by involving the environment token, and
- the set of *player 0's states*: $V_0 = V \setminus V_1$

of the information flow arena. ◀

Note that the V_1 states correspond to situations in which the system's decisions result in the system being informed of the environment's current position for each next `type`-1 typed transition involving a system player.

We define the function $\texttt{M} : V \to 2^{\mathscr{P}}$ for retrieving the corresponding marking of a state of an information flow arena by $v \mapsto \begin{cases} \mathscr{M}(D) & \text{if } v = (D,\cdot,\cdot) \in \mathscr{O} \\ \emptyset & \text{otherwise} \end{cases}$.

▶ **Example 15.** Consider the Petri game with transits depicted in Fig. 9.1. A state v_0 of the information flow arena $A(\mathscr{A})$ corresponding to the initial marking (i.e., with $\texttt{M}(v_0) = \{E, I\}$) is for example $v_0 = (\{(E, 0, \{d_m, d_d\}), (I, 0, \top)\}, 1, \top)$. We have $v_0 \in V_0$ because it cannot be in `TERM` and also cannot satisfy the second disjunct due to the $\top$. However, the state $v_1 = (\{(D_M, 0, \{c_m\}), (D, 0, C)\}, 1, d_m)$ with $C \subseteq \{c_d\}$ is a V_1 state

because $v_1 \in \texttt{TERM}$ and $v_2 = (\{(D_M, 0, \{c_m\}), (M, 0, C)\}, 1, d_m)$ with $C \subseteq \{c_m\}$ is a V_1 state because for $C = \emptyset$ the system refuses to play any further ($v_2 \in \texttt{DL}$), so especially there is no `type`-1 typed transition fireable and, for $C = \{c_m\}$, the only fireable transition is c_m which involves the environment. ◀

12.1.2 Edges and Game

For states that correspond to situations where the Petri game with transits has terminated, we define a set of τ-loops $\texttt{LOOPS} = \{((D, \lightning, \lightning), \tau, (D, \lightning, \tau)) \in V_1 \times \{\tau\} \times V_1 \mid D \in \texttt{TERM}\}$ for accepting such situations. For states that correspond to undesired situations in the Petri game with transits, we define a set of bad states $\mathbf{B} = \{(D, \lightning, \lightning) \in V \mid D \in \texttt{DL} \cup \texttt{NDET}\}$ and add a τ-step to the error state and non-accepting τ-loops $\texttt{LOOPS}_{\lightning} = \{(v, \tau, v_{\lightning}) \mid v \in \mathbf{B}\} \cup \{(v_{\lightning}, \tau, v_{\lightning})\}$ to reject these situations.

The edge relation E of the information flow arena $A(\mathscr{A}) = (V, V_0, V_1, E, I)$ consists of these loops and four different kinds of edges:

- Firstly, we define the set of edges $\texttt{SYS}_\top$ that deals with system states V_0 containing a $\top$-symbol. Here we have to decide on new commitment sets and have the option to decide on the next generation of `type`-2 typed places in case we have not already decided on a non-zero generation. Thus, the marking stays the same and all combinations of possible transitions of the postset are allowed as a commitment set for the places moved in the previous step. All places can decide whether they are not `type`-2 typed at all or opt for the next free generation, if they have not been assigned to one before. The round robin indicator is preserved and the lastly used transition is set to $\top$ for the successor state:

$$\begin{aligned}\texttt{SYS}_\top = \{((D, \lightning, \lightning), \top, (D', \lightning, \top)) \in V_0 \times \{\top\} \times V \mid \exists (p, i, C) \in D : C = \top \wedge \\ \mathscr{M}(D) = \mathscr{M}(D') \wedge \forall (p, i, C) \in D : \exists (p, i', C') \in D' : (\\ C = \top \implies C' \subseteq post(p) \wedge \\ C \neq \top \implies C' = C \wedge \\ i = 0 \implies (i' = 0 \vee i' = \texttt{free}_G(D)) \wedge \\ i \neq 0 \implies i' = i)\}.\end{aligned}$$

- Secondly, we define the set of edges `SYS` which handles the case of `type`-1 fireable transitions t not synchronizing with the environment in a system state V_0. The markings of the decision sets reflect the firing of t. Since states that contain a $\top$-symbol have to resolve this symbol first, no decision set may contain a $\top$-symbol. For the successor's commitment sets, we directly choose new transitions for places of the postset of t, all others commitment sets are preserved as well as the generations of the places. We only move `type`-1 typed places and have to preserve their typing. Newly created tokens can only be `type`-1 typed, since otherwise the places in the preset must have been not `type`-1 typed. These edges have nothing to do with the round robin scheduling and thus, preserve the $\lightning$ indicator. The

successor's last transition indicator t is set to the fired transition:

$$\begin{aligned}\texttt{SYS} = \{((D, r, t), t, (D', r, t)) \in V_0 \times \mathcal{T} \times V \mid \forall (p, i, C) \in D : C \neq \top \wedge \\ pre(t) \subseteq \mathcal{P}_S \wedge D[t\rangle_1 \wedge \mathcal{M}(D)[t\rangle \mathcal{M}(D') \wedge \\ \forall (p', i', C') \in D' : ((p' \in post(t) \implies C' \subseteq post(p') \wedge i' = 0) \wedge \\ (p' \notin post(t) \implies (p', i', C') \in D))\}\end{aligned}$$

- Thirdly, we define the set of edges $\texttt{SYS}_2$ that advances the `type`-2 typed places. To achieve a fair scheduling of the tokens which can proceed without any future interaction with the environment, we implement a round robin procedure. In every V_1 state we first check every transition $t \in \mathcal{T}$ with an index greater than the current round robin identifier r once and allow the firing of the first transition moving one generation not equal to zero. Since the $\texttt{SYS}_2$ edges kind of remain in V_1 states (only with a $\top$-resolving step in between), we can thereby fire one round of `type`-2 typed transitions first and only afterwards, a transition moving the environment token is allowed to fire (see the `ENV` edges). We define a function $nxt : \{0, \ldots, |\mathcal{T}|\} \times \mathcal{D} \to \mathcal{T} \cup \{-\}$ for choosing the next `type`-2 enabled transition with

$$nxt(r, D) = \begin{cases} t & \text{if } \exists i \in \{r, \ldots, |\mathcal{T}|\} : t = \langle \mathcal{T} \rangle_{i-1} \wedge D[t\rangle_2 \\ & \quad \wedge \forall j \in \{r, \ldots, i\} : \neg D[\langle \mathcal{T} \rangle_{j-1}\rangle_2 \\ - & \text{otherwise} \end{cases}.$$

  The symbol $-$ indicates that there is no next `type`-2 enabled transition. We are only allowed to advance when we found such a next transition and the priority is not given to the environment ($r \neq 0$) or no environment transition is fireable at all. Is this the case, the markings reflect the firing of the transition, the generations are preserved, as well as commitment sets of places not involved in the firing of the transition. The other commitment sets are chosen in the next step and thus, are set to $\top$. The last transition indicator of the successor is again set to the fired transition. Note that the check whether t is `type`-2 fireable is done within the *nxt* function:

$$\begin{aligned}\texttt{SYS}_2 = \{((D, r, t), t, (D', r', t)) \in V_1 \times \mathcal{T} \times V \mid (r \neq 0 \vee \neg \exists t' \in \mathcal{T} : D[t'\rangle_1) \wedge \\ t = nxt(r, D) \in \mathcal{T} \wedge \mathcal{M}(D)[t\rangle \mathcal{M}(D') \wedge \forall (p', i', C') \in D' : \\ (p' \in post(t) \implies (C' = \top \wedge \exists p \in pre(t) : (p, i', C) \in D) \wedge \\ p' \notin post(t) \implies (p', i', C') \in D) \wedge \\ r' = \begin{cases} r + 1 & \text{if } nxt(r + 1, D') \in \mathcal{T} \\ 0 & \text{otherwise} \end{cases}\}.\end{aligned}$$

- Fourthly, we define the set of edges `ENV` starting in a player 1 state and treating the progress of the environment token. In a V_1 state, all system tokens which are `type`-1 typed have moved as far as possible and need the environment token to further progress. The markings of the decision sets reflect the firing of the

transition. For all not in the firing involved places everything is preserved. The indicator for the last transition of the successor is set to the fired transition. This case is selected iff one round of `type`-2 transitions have fired (i.e., $\curlyvee = 0$), or there is no `type`-2 fireable transition:

$$\begin{aligned}\texttt{ENV} = \{(D, \curlyvee, \lightning), t, (D', \curlyvee', t)) \in V_1 \times \mathscr{T} \times V \mid\ & (\curlyvee = 0 \vee \neg\exists t' \in \mathscr{T} : D[t'\rangle_2) \wedge \\ & D[t\rangle_1 \wedge \mathscr{M}(D)[t\rangle\mathscr{M}(D') \wedge \\ & \forall (p', i', C') \in D' : p' \in post(t) \cap \mathscr{P}_S \implies C' = \top \wedge i' = 0 \wedge \\ & \qquad p' \notin post(t) \implies (p', i', C') \in D \\ & \wedge \curlyvee' = 1\}.\end{aligned}$$

Note that introducing the $\top$ and assigning those states to player 0 allows us to guess the commitment sets and generations.

Note that the single sets of edges are disjunctive because $\texttt{SYS}_\top$ and `SYS` start in V_0 states, whereas $\texttt{SYS}_2$ and `ENV` start in V_1 states. Furthermore, $\texttt{SYS}_\top$ and `SYS` are disjunctive due to the constraint of an existing $\top$-symbol and the sets $\texttt{SYS}_2$ and `ENV` are disjunctive because $t = nxt(\curlyvee, D) \in \mathscr{T}$ implies $D[t\rangle_2$.

With these edges and the previously defined states, we can now define the information flow game. Observe that different to Sec. 10.1, we define the game arena with an initial state because we are only interested in plays starting in a state corresponding to the initial marking of the Petri game with transits. Furthermore, we allow for labeled edges to properly define the product automaton. All notions, such as plays or strategies, can be directly adopted.

▶ **Definition 58 (Information Flow Game).** For a given Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$, the *information flow arena* $A(\mathscr{A}) = (V, V_0, V_1, E, I)$ consists of a finite set of *states* $V = V_0 \dot\cup V_1$, which are partitioned into the states of player 0 and player 1 as defined in Definition 57. The *edge relation* $E \subseteq V \times (\mathscr{T} \cup \{\tau, \top\}) \times V$ is defined by $E = \{(v, t, v') \in \texttt{LOOPS} \cup \texttt{ENV} \cup \texttt{SYS} \cup \texttt{SYS}_\top \cup \texttt{SYS}_2 \mid v \notin \mathbf{B}\} \cup \texttt{LOOPS}_\lightning$ with the definitions stated above.

The *initial state* starts by letting the system players choose their commitment sets and generations, by giving the priority to the $\texttt{SYS}_2$ transitions, and by setting the indicator for the last transition arbitrarily to $\top$: $I = (D, 1, \top)$ with $D = \{(p, 0, C) \in \mathscr{P} \times \{0\} \times (2^{\mathscr{T}} \cup \{\top\}) \mid p \in In \wedge (p \in \mathscr{P}_E \implies C = post(p)) \wedge (p \in \mathscr{P}_S \implies C = \top)\}$.

The *information flow game* of $\mathscr{G}$ is defined as $G(\mathscr{A}) = (A(\mathscr{A}), \texttt{BUCHI}(V_1))$ as the information flow arena with a Büchi condition over the states of player 1. ◀

Each play of the information flow game $G(\mathscr{A})$ induces a unique initial firing sequence of $\mathscr{A}$ by focusing on the labels $t \in \mathscr{T}$ of the edge relation $E \in V \times (\mathscr{T} \cup \{\tau, \top\}) \times V$ (cp. Definition 60), because all edges either keep the marking or the successor marking corresponds to the firing of a transition.

Note that each state in V has a successor because we add looping transitions with `LOOPS` for terminating states and add a path to a non-winning loop with $\texttt{LOOPS}_\lightning$ for

deadlocks. For all other states $(D, \curlyvee, \mathcal{t})$ there must be a fireable transition $t \in \mathscr{T}$ in the corresponding situation in the Petri game with transits, i.e., $\mathscr{M}(D)[t\rangle$. Each state with a $\top$ has a $\texttt{SYS}_\top$ successor. In all other cases, either $D[t\rangle_1$ or $D[t\rangle_2$ hold due to not having a deadlock. For $pre(t) \subseteq \mathscr{P}_S$ and $D[t\rangle_1$, the edges SYS and otherwise either $\texttt{SYS}_2$ or ENV yield the successor. The *nxt* function does not make any harm in the latter case. Initially and after the usage of each ENV edge, the priority is given to the type-2 edges ($\curlyvee = 1$). If there is no transition with $D[t'\rangle_2$, there is an ENV successor no matter the state of $\curlyvee$. Otherwise, there is a $\texttt{SYS}_2$ successor because $nxt(\curlyvee, D)$ yields the first of these transitions. For the successor, $\curlyvee$ is either increased when there is another type-2 transition with higher index in $\langle\mathscr{T}\rangle$, or the priority is given to the ENV edges for the successor state. If there is no fireable environment transition the $\texttt{SYS}_2$ edges yield a successor no matter the actual state of $\curlyvee$.

The Büchi condition over the V_1 states ensures that each system player has truthfully chosen their generation. Incorrectly deciding for a generation not equal to zero results in blocking the corresponding transition because a type-2 transition t can only fire, when all tokens in the preset of t have the same generation. This has the same effect as removing t from the system player's commitment set. When there is no other fireable transition, this results in a deadlock and for deadlocks there is only one edge into the sink state $v_\lightning \in V_0$. Untruly deciding to be type-1 typed results in plays infinitely often visiting only V_0 states due to the scheduling. We only reach a V_1 state when there is a type-1 transition that involves the environment (or in terminating situations). Thus, a player that can infinitely long play without synchronizing indirectly or directly with the environment and is marked as type-1 would infinitely long be prioritized by the scheduling by using the SYS edges. Since we loop in terminating states and those are V_1 states, winning plays still cover terminating plays of the Petri game with transits. Finally, for correct type-2 behavior, infinitely many V_1 states are visited even when the environment player terminated due to the round robin procedure.

We can unroll a strategy of an information flow game to obtain a strategy tree. We inductively create the *strategy tree*, a finitely branching infinite tree $T_\sigma = (T, E_T, l, r)$ with nodes T, labeled edges $E_T \subseteq T \times \mathscr{T} \cup \{\top, \tau\} \times T$, a labeling function $l : T \to V$, and a root $r \in T$, from a strategy $\sigma_{\texttt{inf}} : V^*V_0 \to V$ as follows: The root is labeled with I, i.e., $l(r) = I$. For each node $v_T \in T$ reached by the path labeled with $Iwv \in V^*$ (or for the special case $v_T = r$, then $v = I$) with $l(v_T) = v$, we add for each $(v, t, v') \in E$ a child labeled with v' if $v \in V_1$ holds and add a unique child labeled with $v' = \sigma_{\texttt{inf}}(Iwv)$ if $v \in V_0$ holds. The labels of E are used as labels for the tree's edges.

▶ **Example 16.** Figure 12.2 shows an extract of the information flow game $G(\mathscr{A})$ for the Petri game with transits depicted in Fig. 9.1 with two different winning strategies $\sigma^1_{\texttt{inf}}$ and $\sigma^2_{\texttt{inf}}$ for player 0. The gray rectangles represent player 0's states, whereas player 1's states are depicted white. Initially, a $\top$-resolution step is made by using the $\texttt{SYS}_\top$ edges. Both branches, to the left as well as to right, can only lead to deadlocks, regardless of the chosen generation, as the system chooses to block the only progressive transition i. For reasons of clarity, we omit the step into the non-accepting sink $v_\lightning$ for all bad states. A good $\top$-resolution results in $v_1 = ((E, 0, \{d_d, d_m\}), (I, 0, \{i\}), 1, \top) \in V_0$. Note that

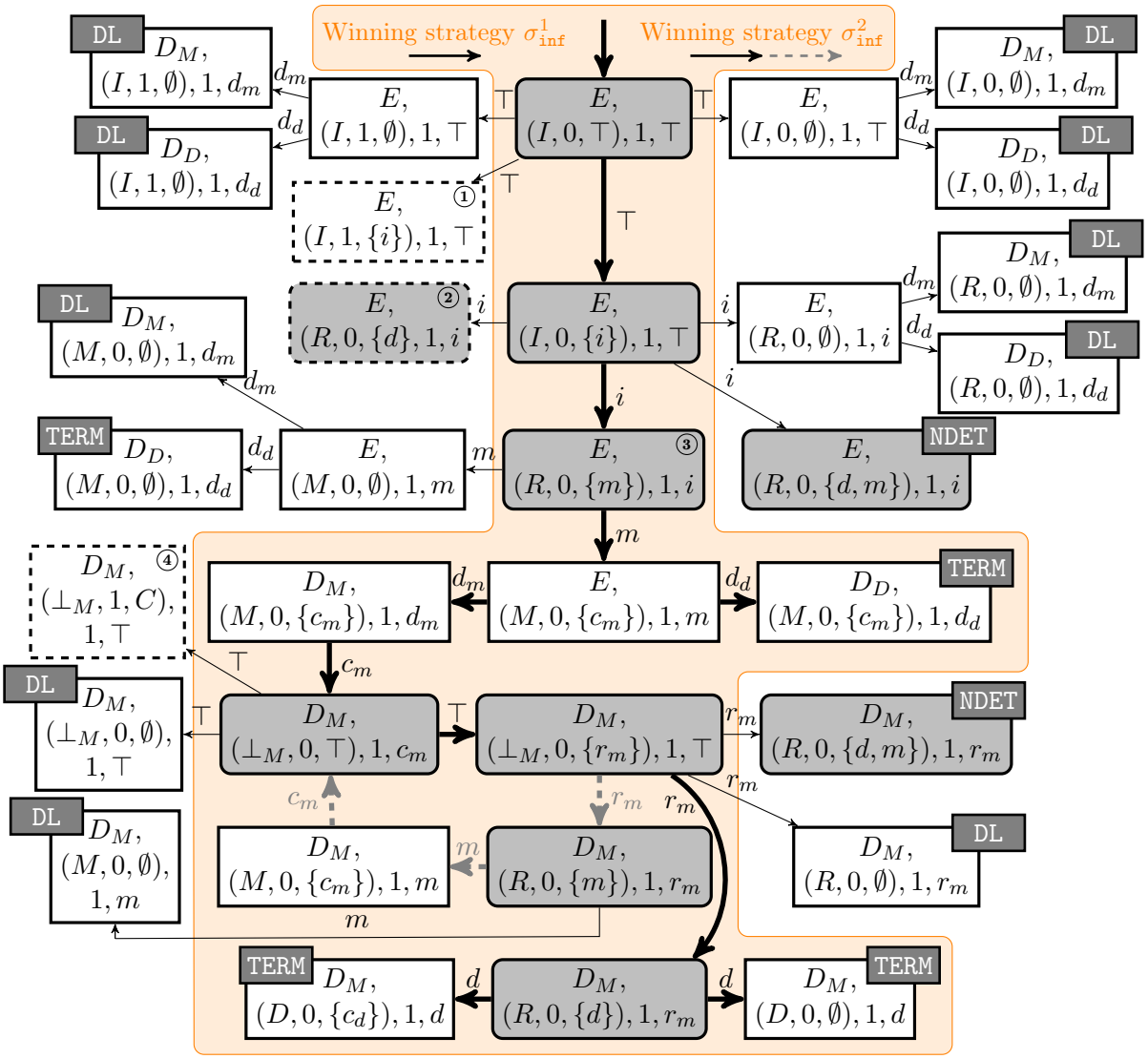


Fig. 12.2: *An extract of the information flow game $G(\mathscr{A})$ for the Petri game with transits depicted in Fig. 9.1. The states of player 0 are depicted by gray rectangles and the states of player 1 by white rectangles. The successors for the states labeled with ①, ②, ④, as well as the state $v_\lightning$ and the non- and accepting loops are omitted for clarity. Similarly, we omit the generations and commitment sets for the environment player. The successor for state labeled by ① lead into non-accepting behavior described in Example 16 and the state labeled by ② is symmetrical to the state labeled by ③. The element labeled by ④ represents two states (one for $C = \emptyset$ and one for $C = \{r_m\}$). The first state is a deadlock and the successors of the other are described in Example 16. The orange shaded area depicts two winning strategies for player 0. Both strategies start by following the black thick arrows. The strategy σ^2_{inf} ends by using the loop depicted by the dashed gray arrows.*

we omit the generations and commitment sets in Fig. 12.2 for the environment player because the generation is always zero and the commitment set is always the complete postset. In $v_1 \in V_0$ only edges from SYS are possible. The successor states labeled with ② and ③ are symmetrical in the sense that the system chooses either to use the drilling or the milling tool, but the environment has not yet decided which tool is defective. Thus, we omit all successors of the state labeled with ②. In the good successor of the state labeled with ③, $v_2 = ((E, 0, \{d_d, d_m\}), (M, 0, \{c_m\}), 1, m) \in V_1$, the system is as advanced as it can be on its own. Even though $\curlyvee = 1$, only edges from ENV are possible because there are no type-2 typed transitions fireable. The orange shaded area together with the solid thick black arrows and the dashed thick gray arrows depict infinitely many winning strategies for player 0. Following the solid thick black arrows until state $v_3 = ((D_M, 0, \{c_m\}), (\bot_M, 0, \{r_m\}), 1, \top) \in V_0$, then finitely looping with the dashed thick gray arrows, until leaving v_3 with the solid thick black arrow yield infinitely many winning strategies due to the not depicted τ-loops in all states in TERM. The strategy $\sigma^1_{\texttt{inf}}$ directly leaves v_3 without any looping, and $\sigma^2_{\texttt{inf}}$ never leaves v_3 with solid thick black arrows again. Both constitute positional winning strategies for player 0.

We also omit all successors for the state labeled by ① in Fig. 12.2. Here the system wrongly decides that the process in I will never synchronize with the environment again. Apart from branches where the system chooses bad commitment sets and end in NDET or DL, there still cannot be any winning successor. Regardless of the system's decision to mill or drill, one branch of a V_1-state (the environment chose to destroy that very tool) must eventually end in DL due to the token in D_M or D_D must be of generation zero and the token in M or D must be of generation one, because the system can never change the generation back. This is different for the states represented by the element labeled by ④. For $C = \emptyset$ we obtain a deadlock, but for $C = \{r_m\}$ we get the type-2 behavior corresponding to state v_3. Here, the path corresponding to the gray dashed arrows leads to a deadlock because the generations in D_M and M also do not fit for this path. However, the path corresponding to the thick black arrows leads to accepting behavior and constitutes another winning strategy. ◀

12.1.3 Properties

In this section we provide core properties of the information flow game $G(\mathscr{A}) = (A(\mathscr{A}), \texttt{BUCHI}(V_1))$ with arena $A(\mathscr{A}) = (V, V_0, V_1, E, I)$ for a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$. Especially, we show the single exponential size of the arena and give a construction to create a strategy $\sigma = (\mathcal{N}^\sigma, \lambda^\sigma)$ for the system players of $\mathscr{G}$ (not necessarily winning) from a winning strategy $\sigma_{\texttt{inf}}$ for player 0 of $G(\mathscr{A})$ and vice versa and show their correspondence.

The size of the arena can be derived directly from the definition of its states (Definition 57) and the definition of the decision sets (Eq. 12.1). For the proof see page 187.

▶ **Lemma 21 (Size of an Information Flow Arena).** *For a given arena $\mathscr{A}$ of a Petri game with transits, the corresponding information flow arena $A(\mathscr{A})$ is single-exponential in the size of the underlying Petri net.* ◀

We construct a strategy of the system players in a Petri game with transits from a winning strategy of player 0 in the information flow game by defining a function $\sigma_{\texttt{inf2pg}}$. This function inductively creates for a given winning strategy $\sigma_{\texttt{inf}} : V^*V_0 \to V$ a Petri net $\mathcal{N}^\sigma = (\mathscr{P}^\sigma, \mathscr{T}^\sigma, \mathscr{F}^\sigma, In^\sigma)$ and an initial labeling homomorphism $\lambda^\sigma : \mathscr{P}^\sigma \cup \mathscr{T}^\sigma \to \mathscr{P} \cup \mathscr{T}$ by traversing the induced strategy tree T_σ of $\sigma_{\texttt{inf}}$ in breadth-first order. In case an edge of the tree is labeled with a transition $t \in \mathscr{T}$, a fresh transition and places corresponding to $post^{\mathscr{A}}(t)$ are added to $\mathcal{N}^\sigma$. The labeling function λ^σ maps the added nodes to its originals. For creating the appropriate flow relation, we define a function $c : T \to 2^{\mathscr{P}^\sigma}$ that assigns to each node of the tree T_σ a corresponding cut of $\mathcal{N}^\sigma$. Finally, we use the function $\texttt{gen} : T \to 2^{\{0,\dots,\texttt{max}_S\}}$ to select a unique strategy for the `type`-2 case. Since $\sigma_{\texttt{inf}}$ can branch in V_1 states, the `type`-2 typed processes must choose their successors equally in all branches. Therefore, the function `gen` stores the generations for which a strategy still has to be added. In V_1 states, the construction selects exactly one successor that is responsible for adding this strategy. We can show that the pair constructed by this definition is a strategy (*not* necessarily winning) for the corresponding Petri game with transits.

▶ **Definition 59 (Constructing a Strategy for a Petri Game with Transits).** Given a winning strategy $\sigma_{\texttt{inf}} : V^*V_0 \to V$ for player 0 in $G(\mathscr{A})$ and the corresponding strategy tree $T_\sigma = (T, E_T, l, r)$. By traversing T_σ in breadth-first order, we step-wise create a Petri net $\mathcal{N}^\sigma = (\mathscr{P}^\sigma, \mathscr{T}^\sigma, \mathscr{F}^\sigma, In^\sigma)$, an initial labeling homomorphism $\lambda^\sigma : \mathscr{P}^\sigma \cup \mathscr{T}^\sigma \to \mathscr{P} \cup \mathscr{T}$, a cut mapping function $c : T \to 2^{\mathscr{P}^\sigma}$, and a generation selecting function $\texttt{gen} : T \to 2^{\{1,\dots,\texttt{max}_S\}}$:

(IA): For the root $r \in T$ we create a new place $p^\sigma \in \mathscr{P}^\sigma$ in $\mathcal{N}^\sigma$ for each $p \in \texttt{M}(l(r)) \subseteq \mathscr{P}$. We map those new places with λ^σ to their originals and remember the created cut In^σ with c. There is no generation, thus, we set $\texttt{gen}(r) = \emptyset$.

(IS): Consider an edge $(v_T, t, v_T') \in E_T$ of the tree T_σ with $l(v_T) = (D, \curlywedge, \lightning)$ and $l(v_T') = (D', \curlywedge', \lightning')$ for which we already considered the node v_T.

Case $t \in \{\top, \tau\}$: Nothing is added to $\mathcal{N}^\sigma$ and the cut is copied, i.e., $c(v_T') = c(v_T)$.
Case $t \in \mathscr{T}$: If $D[t\rangle_2 \Rightarrow i \in \texttt{gen}(v_T)$ holds for the corresponding $i \in \{1, \dots, \texttt{max}_S\}$ with $(p, i, \cdot) \in D$ for all $p \in pre^{\mathscr{A}}(t)$, a fresh transition t' with $\lambda^\sigma(t') = t$ is added and connected via $\mathscr{F}^\sigma$ to the previously created places $P = c(v_T) \cap \{p \in \mathscr{P}^\sigma \mid \lambda^\sigma(p) \in pre^{\mathscr{A}}(t)\}$ as postset transition. Furthermore, we add a set of fresh places P' with $\lambda^\sigma(P') = post^{\mathscr{A}}(t)$ and $|P'| = |post^{\mathscr{A}}(t)|$ and connect them via $\mathscr{F}^\sigma$ as the postset of t'. The new cut is defined via the firing equation of t', i.e., $c(v_T') = (c(v_T) \setminus P) \cup P'$. Otherwise, we add nothing and copy the cut.

For the generation selecting function, we consider two cases:
Case $l(v_T) \in V_1$: For all selected generations, we choose exactly one successor, i.e., $\texttt{gen}(v_T') = \{i \in \texttt{gen}(v_T) \mid \forall (v_T, t'', v_T'') \in E_T : i \notin \texttt{gen}(v_T'')\}$.
Case $l(v_T) \in V_0$: The newly chosen generations are added to the selection, i.e., $\texttt{gen}(v_T') = \{i \in \{1, \dots, \texttt{max}_S\} \mid \exists p \in \mathscr{P} : (p, 0, \top) \in D \wedge (p, i, \cdot) \in D') \vee i \in \texttt{gen}(v_T)\}$.

We define the function $\sigma_{\texttt{inf2pg}}$ to map a given strategy $\sigma_{\texttt{inf}}$ to the tuple $(\mathcal{N}^\sigma, \lambda^\sigma)$ created by the construction above. ◀

We now show that such a Petri net $\mathcal{N}^\sigma$ and initial labeling homomorphism λ^σ is already a deadlock-avoiding strategy $\sigma = (\mathcal{N}^\sigma, \lambda^\sigma)$ of the system players in $\mathcal{G}$.

▶ **Theorem 7 (Transferable Strategies: From Information Flow Game to Petri Game).** *Given a Petri game with transits $\mathcal{G} \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathcal{A}$ and a corresponding information flow game $G(\mathcal{A})$. If player 0 has a winning strategy $\sigma_{\texttt{inf}}$ in $G(\mathcal{A})$ then $\sigma = \sigma_{\texttt{inf2pg}}(\sigma_{\texttt{inf}})$ is a deadlock-avoiding strategy of the system players in $\mathcal{G}$.* ◀

Proof Sketch. That σ is a *subprocess* of an unfolding $\beta_U = (\mathcal{N}^U, \lambda^U)$ of $\mathcal{G}$ directly follows from Definition 59, where σ is created inductively by adding only fresh nodes according to firing sequences in $\mathcal{G}$ and labeling them accordingly. The strategy does not restrict any *pure environment transition* $t \in \mathcal{T}^U$ with $pre^U(t) \subseteq \mathcal{P}^U_E$, because a winning play visits infinitely many V_1 states (especially left by `ENV` edges in case $pre^U(t) \subseteq \mathcal{P}^\sigma_E$) and in a V_1 state all successors have to be considered by T_σ. So a transition $t \in \mathcal{T}^U$ with $pre^U(t) \subseteq \mathcal{P}^\sigma_E$ will eventually be added, i.e., $t \in \mathcal{T}^\sigma$. The other condition of the *justified refusal* property holds mainly because the two-player game does not consider any instances of transitions and due to the restricted scheduling with choosing the commitment sets as early as possible. The nondeterminism condition ensures that we do not reach another cut containing the place $p \in pre^U(t) \cap \mathcal{P}^\sigma_S$ while allowing any instance of $\lambda^U(t)$. Here the restriction to Petri games with transits without mixed communication is crucial to not miss any nondeterminism. By that the game forbids or allows all instances equally in corresponding cuts. The strategy σ is *deterministic* because the nondeterminism condition of the decision set ensures that there is no cut in the construction where two transitions sharing a system place are enabled. Adding the transitions in different cuts is impossible as a conflict would arise between the transitions. This cannot happen in a branching process for transitions enabled in the same cut. For transitions moving only `type`-2-typed places, the selection of exactly one successor in each branching state ensures the determinism property. The strategy σ is *deadlock-avoiding* because there cannot be any infinite $\top$-loops due to E and τ-loops can only stem from situations where also the unfolding terminates. For all other edges the construction also adds the transitions. Only in case of `type`-2 transitions. There, the transitions are only added for one branch. Since such transitions are independent of the branching of the environment, we still do not deadlock. For more details see the proof on page 187. □

For the other direction we define a function $\sigma_{\texttt{pg2inf}}$ which maps a given deadlock-avoiding strategy $\sigma = (\mathcal{N}^\sigma, \lambda^\sigma)$ of the system players in a Petri game with transits $\mathcal{G} \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathcal{A}$ to a winning strategy $\sigma_{\texttt{inf}}$ of player 0 in the two-player game $G(\mathcal{A})$. For that, we first define a function creating a sequence of transitions from a prefix of a play or a complete play of $G(\mathcal{A})$.

▶ **Definition 60 (Transitions of a Play).** We define the function $fs : V^* \cup V^\omega \to \mathcal{T}^* \cup \mathcal{T}^\omega$ inductively with $fs(\epsilon) = \epsilon$ and $fs(wv) = \begin{cases} fs(w) \cdot t & \text{if } v = (D, \curlywedge, t) \text{ and } t \in \mathcal{T} \\ fs(w) & \text{otherwise} \end{cases}$. ◀

By that, we obtain an initial firing sequence of the Petri game with transits for all plays of the information flow game. Note that for infinite plays, the corresponding sequence of transitions may be finite due to the τ-loops. For the construction of the strategy $\sigma_{\mathtt{inf}}$ for the information flow game from a deadlock-avoiding Petri game with transits strategy σ, we use the notion of whether a system place $p \in \mathscr{P}_S^\sigma$ will ever receive new information about the environment again, i.e., $\mathtt{type}(p) = 2$ iff $\forall p_E \in \mathscr{P}_E^\sigma : p_E \not\leq p \implies \forall x \in \mathit{fut}^\sigma(p) : p_E \not\leq x \wedge |\mathit{fut}^\sigma(p)| = \infty$.

The intuition of the definition of $\sigma_{\mathtt{inf}} : V^*V_0 \to V$ is that we focus on mapping words $\pi_{\mathtt{inf}}$ that are plays of $G(\mathscr{A})$ and where $\mathit{fs}(\pi_{\mathtt{inf}})$ corresponds to a covering firing sequence of a play conforming to σ. In all other situations we choose an arbitrary successor. Since all input words end in V_0 states, we choose the successors according to $\mathtt{SYS}_\top$ and $\mathtt{SYS}$ edges and choose the only ambiguous elements, i.e., the generations and the commitment sets according to σ.

▶ **Definition 61 (Constructing an Information Flow Strategy).** For a given deadlock-avoiding strategy $\sigma = (\mathscr{N}^\sigma, \lambda^\sigma)$ and the corresponding information flow game $G(\mathscr{A}) = (A(\mathscr{A}), \mathtt{BUCHI}(V_1))$ with arena $A(\mathscr{A}) = (V, V_0, V_1, E, I)$, we define a function $\sigma_{\mathtt{inf}} : V^*V_0 \to V$.

Given a word $\pi_{\mathtt{inf}} = v_0 \cdots v_n v \in V^*V_0$. If $\pi_{\mathtt{inf}}$ is not a prefix of a play in $G(\mathscr{A})$, we choose an arbitrary successor according to E. Otherwise:
If $v \in \mathbf{B} \cup \{v_\lightning\}$, there is only the error state $v_\lightning$ as successor, i.e., $\sigma_{\mathtt{inf}}(\pi_{\mathtt{inf}}) = v_\lightning$.
Otherwise, $v = (D, \lightning, t)$.
If there is *no* cut $\mathscr{C} \subseteq \mathscr{P}^\sigma$ which is reached from the initial cut In^σ by step-wise firing the transitions $t \in \mathscr{T}^\sigma$ corresponding to the sequence $\mathit{fs}(\pi_{\mathtt{inf}})$ of the prefix of the play $\pi_{\mathtt{inf}}$, we can again choose an arbitrary successor. Otherwise:
If there exists a $(p, i, \top) \in D$, we define $\sigma_{\mathtt{inf}}(\pi_{\mathtt{inf}}) = (D', \lightning, \top) = v'$ such that $(v, \top, v') \in \mathtt{SYS}_\top$ and for all $(p, i, C) \in D$ there is a $(p, i', C') \in D'$ with

$$C' = \begin{cases} \lambda^\sigma(\mathit{post}^\sigma(p^\sigma)) & \text{if } C = \top \\ C & \text{otherwise} \end{cases} \quad \text{and } i' = \begin{cases} \mathtt{free}_G(D) & \mathtt{type}(p^\sigma) = 2 \wedge i = 0 \\ i & i \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

for $p^\sigma \in \mathscr{C}$ with $\lambda^\sigma(p^\sigma) = p$.
Otherwise, we choose one of the system transitions $t^\sigma \in \mathscr{T}^\sigma$ with $\mathit{pre}^\sigma(t^\sigma) \subseteq \mathscr{P}_S^\sigma$ and $\mathscr{C}[t^\sigma\rangle\mathscr{C}'$ that must exists due to $\mathscr{C}$ corresponding to $v \in V_0 \setminus \mathbf{B}$ and thus, cannot be terminating or deadlocking. We define $\sigma_{\mathtt{inf}}(\pi_{\mathtt{inf}}) = (D', \lightning, t) = v'$ with $t = \lambda^\sigma(t^\sigma)$ such that $(v, t, v') \in \mathtt{SYS}$ and for all $p' \in \mathit{post}^{\mathscr{A}}(t)$ there is a $(p', 0, C') \in D'$ with

$$C' = \lambda^\sigma(\mathit{post}^\sigma(p^\sigma))$$

for $p^\sigma \in \mathscr{C}'$ with $\lambda^\sigma(p^\sigma) = p'$.

We define the function $\sigma_{\mathtt{pg2inf}}$ to map a given deadlock-avoiding strategy $\sigma = (\mathscr{N}^\sigma, \lambda^\sigma)$ to the function $\sigma_{\mathtt{inf}}$ created by the construction above. ◀

We now show the function $\sigma_{\mathtt{inf}}$ is already a winning strategy of player 0 in the information flow game.

▶ **Theorem 8 (Transferable Strategies: From Petri Game to Information Flow Game).** *Given a Petri game with transits $\mathcal{G} \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathcal{A}$ and a corresponding information flow game $G(\mathcal{A})$. If the system players have a deadlock-avoiding strategy σ in $\mathcal{G}$ then $\sigma_{\texttt{inf}} = \sigma_{\texttt{pg2inf}}(\sigma)$ is a winning strategy of player 0 in $G(\mathcal{A})$.* ◀

Proof Sketch. The function $\sigma_{\texttt{inf}}$ is a *strategy* because it defines successors for V_0 states and in all cases the definition either fits to the successor definition of the edges $\texttt{SYS}_\top$, $\texttt{SYS}$, or $\texttt{LOOPS}_{\lightning}$, or an arbitrary successor is chosen. The strategy $\sigma_{\texttt{inf}}$ is *winning* because for all plays conforming to $\sigma_{\texttt{inf}}$ the sequence $fs(\pi_{\texttt{inf}})$ corresponds to an initial firing sequence in $\mathcal{A}$ and thus, each state corresponds to a cut $\mathscr{C}$ of $\mathcal{N}^\sigma$. With that we can show that the non-accepting sink $v_{\lightning} \in V_0$ is avoided by every play, because σ is deterministic and deadlock-avoiding. To avoid the case where we deadlock due to a non-uniform assignment of generations to places in the preset of the necessary transition, it is crucial that whenever a $\top$-symbol is resolved, *all* places in the successor decision set can choose a new generation in case they had been of generation zero before. This is exploited by the constructed strategy $\sigma_{\texttt{inf}}$ by uniformly choosing the generations according to $\mathcal{N}^\sigma$. Finally, we can show that it is not possible that $\pi_{\texttt{inf}}$ still does not visit infinitely many V_1 states, even though it avoids $v_{\lightning} \in V_0$ and loops in V_1 states for terminating situations, because then we would haven chosen wrong generations, which however happens correctly according to σ. For more details see the proof on page 193.□

12.2 Transit Automata

With the information flow game we have a procedure to check the existence of a strategy for the system players in a Petri game with transits and create one if it exists. In this game the general properties of a strategy, i.e., being a subprocess of the unfolding, the causality, the determinism, the justified refusal, and the deadlock-avoidance is covered. In this section we introduce three kinds of *transit automata* which, triggered by firing sequences of the Petri game with transits $\mathcal{G}$, check whether the winning condition of $\mathcal{G}$ is satisfied. In the end, the parallel composition of the information flow game and a deterministic version of the transit automaton creates a two-player game over a finite graph with complete information that serves to decide whether there is a *winning* strategy for the system players in a Petri game with transits and create one if it exists.

The general idea is that the automaton tracks for every situation of the game the current flow chains. We create an *existential* transit automaton which nondeterministically guesses the relevant chain to check the existential winning conditions. Similarly, we use a *universal* transit automaton which universally branches into all current chains to check the universal winning conditions. Finally, we use the ideas of the existential transit automaton to build the transit automaton used to check local Flow-LTL formulas with only places as atomic propositions. The challenge is to properly handle the infinitely many flow chains which may exist due to the possibility of creating new chains at any point during the game. Depending on the winning condition, the determinization of the automaton or the special treatment of runs without chains leads in some cases to

a more expressive acceptance condition of the automaton in comparison to the winning condition of the game. Table 12.1 gives on overview of this relation.

Tab. 12.1: *The first row lists the types of winning conditions of a Petri game with transits. The second and the third row shows for all but the last column the corresponding acceptance conditions of the deterministic existential or universal transit automata. The last column shows the acceptance conditions of the local Flow-LTL transit automata for local Flow-LTL formulas with existential or universal flow subformulas.*

	Safety	Reachability	Büchi	co-Büchi	Parity	Local Flow-LTL
Existential	co-Büchi	Reachability	Parity	co-Büchi	Parity	Parity
Universal	Safety	Büchi	Büchi	Parity	Parity	Parity

12.2.1 Existential Transit Automata

In this section, we define the transit automata used to check whether a Petri game with transits is winning with respect to the *existential* safety, reachability, Büchi, co-Büchi, and parity conditions. The automata for the corresponding universal conditions are presented in Sec. 12.2.2 and the local Flow-LTL condition is considered in Sec. 12.2.3.

In its initial state, the existential transit automata guess the desired chain for the winning condition and afterwards track this chain when triggered by transitions of the Petri game with transits that extend the chain. To not accept a run of the automaton for the existential safety condition that never starts tracking a chain, we have to switch to a co-Büchi condition to avoid runs visiting the initial state infinitely often and adapt the transition relation to loop in bad states. For the existential co-Büchi condition we can just add the initial state to the co-Büchi states of the automaton to avoid accepting these runs. For parity we can exclude this run by mapping the initial state to an odd color.

▶ **Definition 62 (Existential Transit Automaton (ETA)).** For a given Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN})$ with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ and an existential local winning condition $\texttt{WIN} = \exists\text{-}W(\mathscr{W})$ with $W \in \{\texttt{SAFE}, \texttt{REACH}, \texttt{BUCHI}, \texttt{COBUCHI}, \texttt{PARITY}\}$ and $\mathscr{W} \subseteq \mathcal{P}_S \cup \mathcal{P}_E$ respective $\mathscr{W} : \mathcal{P}_S \cup \mathcal{P}_E \to \mathbb{N}$, we define the nondeterministic *existential transit automaton (ETA)* $\Lambda_\exists^{\texttt{WIN}}(\mathcal{G}) = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \to, \texttt{ACC})$ with the *states* $S = \mathcal{P} \uplus \{s_0\}$ and the *initial state* s_0. We use the relation $\to' = \mathit{guess} \uplus \mathit{transit} \uplus \mathit{total} \uplus \mathit{stutter} \subseteq S \times (\mathcal{T} \cup \{\top, \tau\}) \times S$ with

$$
\begin{aligned}
\mathit{guess} &= \{(s_0, t, s) \in S \times \mathcal{T} \times S \mid (\triangleright, s) \in \Upsilon(t)\} \\
\mathit{transit} &= \{(s, t, s') \in S \times \mathcal{T} \times S \mid (s, s') \in \Upsilon(t)\} \\
\mathit{total} &= \{(s, t, s) \in S \times \mathcal{T} \times S \mid \neg\exists s' \in S : (s, t, s') \in \mathit{transit}\} \\
\mathit{stutter} &= \{(s, l, s) \in S \times \{\top, \tau\} \times S\}
\end{aligned}
$$

to define the following *transition relation* $\rightarrow \subseteq S \times (\mathscr{T} \cup \{\top, \tau\}) \times S$. The *acceptance condition* $\texttt{ACC} \subseteq S^\omega$ and $\rightarrow$ are dependent on the given winning condition $\texttt{WIN}$ of the Petri game with transits:

$$\rightarrow = \begin{cases} (\rightarrow' \setminus (\mathscr{W} \times \mathscr{T} \times S)) \cup (\mathscr{W} \times \mathscr{T} \times \mathscr{W}) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{SAFE}(\mathscr{W}) \\ \rightarrow' & \text{otherwise} \end{cases}$$

and

$$\texttt{ACC} = \begin{cases} \texttt{COBUCHI}(\{s_0\} \cup \mathscr{W}) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{SAFE}(\mathscr{W}) \\ \texttt{REACH}(\mathscr{W}) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{REACH}(\mathscr{W}) \\ \texttt{BUCHI}(\mathscr{W}) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{BUCHI}(\mathscr{W}) \\ \texttt{COBUCHI}(\{s_0\} \cup \mathscr{W}) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{COBUCHI}(\mathscr{W}) \\ \texttt{PARITY}(\Omega) & \text{if } \texttt{WIN} = \exists\text{-}\texttt{PARITY}(\mathscr{W}) \end{cases}$$

with $\Omega = \mathscr{W} \cup \{(s_0, 1)\}$. ◀

The existential transit automaton is a nondeterministic automaton with at most $|\mathscr{P}|+1$ reachable states. Note that the edges in the set *total* additionally include loops in the initial state for all transitions $t \in \mathscr{T}$. These are used to allow for the choosing of a later created flow chain by skipping the tracking of the previous creations of the corresponding instances of the starting transition. Thus, we can loop in the initial state s_0 as long as we do not want to track a chain. When tracking a chain has started, we can only loop with transitions not extending the chain (or with $\top$ and τ). All other transitions have to extend the chain, and it can nondeterministically be decided which of the possible extensions of the chain should be tracked.

Tab. 12.2: *References for determinizing nondeterministic word automata dependent on the different acceptance conditions. The best algorithm of each construction causes an exponential blow-up.*

nondet.	Safety	Reach.	Büchi	co-Büchi	Parity
det.	Safety [RS59]	Reach. [RS59]	Parity [Saf88; Pit06; Sch09]	co-Büchi [MH84][1]	Parity [Saf88; Pit06; SV14]

[1] "When applied to universal Büchi automata, the translation in [MH84], of alternating Büchi automata into NBW, results in DBW. By dualizing it, one gets a translation of NCW to DCW." [BK09]

The determinizations of the transit automata yield the automata used for the product automata in Sec. 12.3 to provide a decision procedure for Petri games with transits. Table 12.2 gives an overview of results from the literature for the determinization of nondeterministic word automata with different acceptance conditions. These results together with the correspondence of the acceptance and the winning conditions in Definition 62 yield the corresponding entries of the second row of Tab. 12.1.

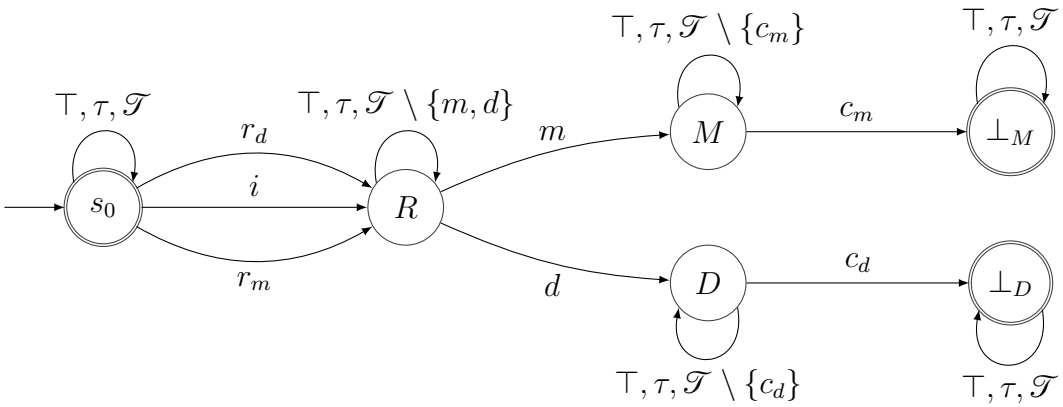


Fig. 12.3: *The ETA corresponding to the Petri game with transits depicted in Fig. 9.1 on page 141 for the existential safety condition* $\exists$-SAFE$(\{\bot_M, \bot_D\})$.

▶ **Example 17.** In Fig. 12.3 the ETA $\Lambda^{\texttt{WIN}}_{\exists}(\mathscr{G}) = (\mathscr{T} \cup \{\top, \tau\}, \{s_0, R, M, D, \bot_M, \bot_D\}, s_0, \to, \texttt{COBUCHI}(\{s_0\} \cup \{\bot_M, \bot_D\}))$ for the Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$, depicted in Fig. 9.1 on page 141, with $\texttt{WIN} = \exists\text{-}\texttt{SAFE}(\{\bot_M, \bot_D\})$ is shown. Since there are no transitions branching the data flow, the only nondeterminism is present in the initial state s_0. ◀

We show that the acceptance of the constructed automata correspond to whether a play of the corresponding Petri game with transits is winning. The first statement of Lemma 22 shows that the acceptance of the automaton is independent of the ordering of concurrent transitions of a covering firing sequence of a play. The second statement shows that a play is winning iff all of its covering firing sequences are accepted by the automaton. For that, we first define the sequence of transitions belonging to a firing sequence $\zeta = M_0[t_0\rangle M_1[t_1\rangle M_2 \cdots$ of a play $\pi = (\mathscr{N}^R_{\mathscr{A}}, \rho)$ that stutters with τ-steps for finite firing sequences: $\sigma_{\mathscr{T}}(\zeta) = \rho(t_0)\rho(t_1)\cdots$ if ζ is infinite and $\sigma_{\mathscr{T}}(\zeta) = \rho(t_0)\cdots\rho(t_{n-1})\tau^\omega$ if $\zeta = M_0[t_0\rangle \cdots [t_{n-1}\rangle M_n$ is finite.

▶ **Lemma 22 (Expediency of the ETA).** *Given a play π of a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$ with an* existential *local winning condition $\texttt{WIN} = \exists\text{-}W(\mathscr{W})$ with $W \in \{\texttt{SAFE}, \texttt{REACH}, \texttt{BUCHI}, \texttt{COBUCHI}, \texttt{PARITY}\}$, then*

(i) $\exists \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\texttt{WIN}}_{\exists}(\mathscr{G})) \implies \forall \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\texttt{WIN}}_{\exists}(\mathscr{G}))$, *and*

(ii) $\pi \in \texttt{WIN} \iff \forall \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\texttt{WIN}}_{\exists}(\mathscr{G}))$

hold. ◀

Proof Sketch. Property *(i)* easily follows from the construction of the automaton. Two covering firing sequences of a run π only differ in the order of concurrent transitions. All transitions have to occur in both covering firing sequences and the order of causally dependent transitions must be preserved. Transitions extending a flow chain are causally dependent and the ETA only allows state changes for transitions extending the chain.

Hence, the runs of the automaton on words belonging to two covering firing sequences of the same play only differ in finite looping and the concrete time points for the state changes. Since the acceptance conditions are independent of these differences, all traces of covering firing sequences are accepted when one trace is accepted.

For the *completeness* proof of Property *(ii)*, i.e., direction "⇒", we can focus on the transitions extending the winning flow chain. These are again causally related and must occur in every covering firing sequence in the same order. All other transitions of the firing sequence may only enforce finite loops in the states corresponding to the places of the chain. Since the adaption of the transition relation compensates the type change of the acceptance condition for the $\exists$-$\texttt{SAFE}(\mathscr{W})$ condition, it is easy to see that the winning property on the places of the chain and the acceptance condition of the run of the automaton coincide.

For the *soundness* proof, i.e., direction "⇐", the acceptance condition for each ETA enforces to leave the initial state s_0 at some point, and thus, demands the existence of a chain. State changes can only happen with *transit* in case the corresponding chain can be extended, and, since *total* is disjoint to *transit*, no extending of the chain can be missed. Since the states of the run and the places of the chain coincide, the constructed chain satisfies the winning property, and therewith, the play is winning. For more details see the proof on page 194. □

▶ **Example 18.** Consider the two plays π_1 and π_2 depicted in Fig. 9.2 on page 142. For play π_1, there are the covering firing sequences $\zeta_1 = \{E, I\}[d_d\rangle\{D_D, I\}[i\rangle\{D_D, R\}[m\rangle\{D_D, M\}$, $\zeta_2 = \{E, I\}[i\rangle\{E, R\}[d_d\rangle\{D_D, R\}[m\rangle\{D_D, M\}$, and $\zeta_3 = \{E, I\}[i\rangle\{E, R\}[m\rangle\{E, M\}[d_d\rangle\{D_D, M\}$. All traces $\sigma_{\mathscr{T}}(\zeta_1) = d_d \cdot i \cdot m \cdot \tau^\omega$, $\sigma_{\mathscr{T}}(\zeta_2) = i \cdot d_d \cdot m \cdot \tau^\omega$, and $\sigma_{\mathscr{T}}(\zeta_3) = i \cdot m \cdot d_d \cdot \tau^\omega$ are accepted by the ETA presented in Example 17 because we can loop in state s_0 until letter i is read and change to state R. Then, we possibly loop in state R again, until m is read, switch to state M, and loop there forever. The trace $\sigma_{\mathscr{T}}(\zeta_3)$ corresponds to the respective play for the strategy $\sigma_{\texttt{inf}}$ depicted in Fig. 12.2 on page 168. Similarly, all traces of covering firing sequences of play π_2 are accepted. We only consider the trace of the covering firing sequence $\zeta_4 = \{E, I\}[i\rangle\{E, R\}[m\rangle\{E, M\}[d_m\rangle\{D_M, M\}[c_m\rangle\{D'_M, \bot_M\}[r_m\rangle\{D'_M, R'\}[d'\rangle\{D'_M, D'\}$ corresponding to the respective play of $\sigma_{\texttt{inf}}$. For this trace $\sigma_{\mathscr{T}}(\zeta_4) = i{\cdot}m{\cdot}d_m{\cdot}c_m{\cdot}r_m{\cdot}d{\cdot}\tau^\omega$, we have to loop in s_0 additionally for the first creation of the chain (reading letter i). Only after reading letter r_m, we switch into state R and afterwards to state D, in which we loop forever. ◀

12.2.2 Universal Transit Automata

Similarly to the ETA, the *universal transit automaton (UTA)* serves for checking whether a Petri game with transits is winning with respect to the *universal* safety, reachability, Büchi, co-Büchi, and parity condition. Instead of guessing and then tracking the desired chain, the UTA tracks *all* chains when triggered by transitions by universally branching for all new or branching data flow chains. To also accept branches that do not possess any chain, we switch to a Büchi acceptance condition for the universal reachability condition,

mark the initial state as a Büchi state, and loop in the desired reachable states. For the universal Büchi condition we can just add the initial state to the Büchi states to also accept branches without any data flow chain. Mapping the initial state to an even color solves this case for the parity condition.

▶ **Definition 63 (Universal Transit Automaton (UTA)).** For a given Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN})$ with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ and a universal local winning condition $\texttt{WIN} = \forall\text{-}W(\mathscr{W})$ with $W \in \{\texttt{SAFE}, \texttt{REACH}, \texttt{BUCHI}, \texttt{COBUCHI}, \texttt{PARITY}\}$ and $\mathscr{W} \subseteq \mathcal{P}_S \cup \mathcal{P}_E$ respective $\mathscr{W} : \mathcal{P}_S \cup \mathcal{P}_E \to \mathbb{N}$, we define the *universal transit automaton (UTA)* $\Lambda_\forall^{\texttt{WIN}}(\mathcal{G}) = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \to, \texttt{ACC})$ with the *states* $S = \mathcal{P} \uplus \{s_0\}$ and the *initial state* s_0. We use the relation $\to' = \mathit{guess} \uplus \mathit{transit} \uplus \mathit{total} \uplus \mathit{stutter} \subseteq S \times (\mathcal{T} \cup \{\top, \tau\}) \to \mathbb{B}^+(S)$ with

$$
\begin{aligned}
\mathit{start} =& \{(s_0, t, B) \in S \times \mathcal{T} \times \mathbb{B}^+(S) \mid B = s_0 \wedge \bigwedge_{(\triangleright, s) \in \Upsilon(t)} s\} \\
\mathit{transit} =& \{(s, t, B) \in S \times \mathcal{T} \times \mathbb{B}^+(S) \mid \exists (s, s') \in \Upsilon(t) : B = \bigwedge_{(s, s') \in \Upsilon(t)} s'\} \\
\mathit{total} =& \{(s, t, s) \in (S \setminus \{s_0\}) \times \mathcal{T} \times \mathbb{B}^+(S) \mid \neg\exists (s, t, \cdot) \in \mathit{transit}\} \\
\mathit{stutter} =& \{(s, l, s) \in S \times \{\top, \tau\} \times \mathbb{B}^+(S)\}
\end{aligned}
$$

to define the following *transition relation* $\to \subseteq S \times (\mathcal{T} \cup \{\top, \tau\}) \to \mathbb{B}^+(S)$. The *acceptance condition* $\texttt{ACC} \subseteq S^\omega$ and $\to$ are dependent on the given winning condition WIN of the Petri game with transits:

$$
\to = \begin{cases} (\to' \setminus (\mathscr{W} \times \mathcal{T} \times \mathbb{B}^+(S))) \cup (\mathscr{W} \times \mathcal{T} \times \mathscr{W}) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{REACH}(\mathscr{W}) \\ \to' & \text{otherwise} \end{cases}
$$

and

$$
\texttt{ACC} = \begin{cases} \texttt{SAFE}(\mathscr{W}) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{SAFE}(\mathscr{W}) \\ \texttt{BUCHI}(\{s_0\} \cup \mathscr{W}) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{REACH}(\mathscr{W}) \\ \texttt{BUCHI}(\{s_0\} \cup \mathscr{W}) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{BUCHI}(\mathscr{W}) \\ \texttt{COBUCHI}(\mathscr{W}) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{COBUCHI}(\mathscr{W}) \\ \texttt{PARITY}(\Omega) & \text{if } \texttt{WIN} = \forall\text{-}\texttt{PARITY}(\mathscr{W}) \end{cases}
$$

with $\Omega = \mathscr{W} \cup \{(s_0, 0)\}$. ◀

In the initial state, the automata loop for all transitions that do not start any chain because the conjunction over an empty set resolves to *true*. Whenever a transition starts a chain, a new branch for every starting chain is created tracking this chain. Additionally, another branch is started that remains in the initial state to wait for further chains to be started later in the game. Again we loop with $\top$ and τ and with all transitions not forwarding the local data flow.

To obtain a deterministic automaton, we can first dualize, determinize, and then again dualize the automaton. *Dualizing* an alternating automaton means interchanging the

conjunctions and disjunctions of the transition relation and using the complement of the acceptance condition. Hence, for a universal automaton the dualization yields a nondeterministic automaton. So, for a UTA $\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}) = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \to, \mathtt{ACC})$ the *dual* automaton is the nondeterministic automaton $\widetilde{\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})} = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \widetilde{\to}, S^\omega \setminus \mathtt{ACC})$. Muller and Schupp show that the dual automaton accepts exactly the complement language [MS87]. Thus, $\mathcal{L}(\widetilde{\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})}) = \overline{\mathcal{L}(\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}))}$. Using determinizing techniques (see Tab. 12.2) yields a deterministic automaton Λ with $\mathcal{L}(\widetilde{\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})}) = \mathcal{L}(\Lambda)$. Finally, dualizing a deterministic automaton does not change anything for the transition relation. Thus, dualizing Λ yields a deterministic automaton $\widetilde{\Lambda}$ with the complement acceptance condition and $\mathcal{L}(\widetilde{\Lambda}) = \overline{\mathcal{L}(\Lambda)}$. Hence, $\mathcal{L}(\widetilde{\Lambda}) = \mathcal{L}(\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}))$. Table 12.3 gives an overview of the transformations of the acceptance conditions. Note that the last row fits the last row of Tab. 12.1.

Tab. 12.3: *Correspondence of the acceptance conditions of the universal transit automaton $\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})$, its dual automaton Λ' (nondeterministic), the deterministic version Λ of Λ', and its dual automaton $\widetilde{\Lambda}$ (deterministic) with respect to the winning condition $\forall$-$W(\mathscr{W})$ of the Petri game with transits $\mathcal{G}$.*

$\forall$-$W(\mathscr{W})$	Safety	Reachability	Büchi	co-Büchi	Parity
$\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})$	Safety	Büchi	Büchi	co-Büchi	Parity
$\widetilde{\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})}$	Reachability	co-Büchi	co-Büchi	Büchi	Parity
Λ	Reachability	co-Büchi	co-Büchi	Parity	Parity
$\widetilde{\Lambda}$	Safety	Büchi	Büchi	Parity	Parity

By applying the dualization techniques to the edges and the acceptance sets of the existential and universal automata, we can easily show the following correspondence of these automata. For more details see the proof on page 196.

▶ **Lemma 23 (Dualization).** *Existential and universal transit automata are* dual *to one another:* $\widetilde{\Lambda_\exists^{\mathtt{WIN}}(\mathcal{G})} = \Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})$ *and* $\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}) = \widetilde{\Lambda_\exists^{\mathtt{WIN}}(\mathcal{G})}$. ◀

Similarly as for the existential transit automata, we can show that the acceptance of the universal transit automata fit to the winning conditions of the corresponding Petri game with transits. This result is mainly due to the corresponding lemma for the ETA (Lemma 22) by exploiting the dualization relation (Lemma 23).

▶ **Lemma 24 (Expediency of the UTA).** *Given a play π of a Petri game with transits $\mathcal{G} = (\mathcal{A}, \mathtt{WIN})$ with an* universal *local winning condition $\mathtt{WIN} = \forall$-$W(\mathscr{W})$ with $W \in \{\mathtt{SAFE}, \mathtt{REACH}, \mathtt{BUCHI}, \mathtt{COBUCHI}, \mathtt{PARITY}\}$, then*

(i) $\exists \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \in \mathcal{L}(\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G})) \implies \forall \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \in \mathcal{L}(\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}))$, *and*

(ii) $\pi \in \mathtt{WIN} \iff \forall \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \in \mathcal{L}(\Lambda_\forall^{\mathtt{WIN}}(\mathcal{G}))$ *holds.* ◀

Proof. Since the ETA and the UTA are dual, this lemma directly follows from Lemma 22. For Property *(i)* we show the contraposition. Let $\zeta \in Z(\pi)$ be the existing covering firing sequence of π with $\sigma_{\mathcal{T}}(\zeta) \notin \mathscr{L}(\Lambda_{\forall}^{\texttt{WIN}}(\mathcal{G}))$. Hence, $\sigma_{\mathcal{T}}(\zeta) \in \overline{\mathscr{L}(\Lambda_{\forall}^{\texttt{WIN}}(\mathcal{G}))} = \mathscr{L}(\widetilde{\Lambda_{\forall}^{\texttt{WIN}}}(\mathcal{G}))$ due to the definition of the dualization. Lemma 23 yields $\sigma_{\mathcal{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}(\mathcal{G}))$ and Property *(i)* of Lemma 22 yields $\forall \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}(\mathcal{G}))$. Thus, $\forall \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \notin \overline{\mathscr{L}(\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}(\mathcal{G}))} = \mathscr{L}(\widetilde{\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}}(\mathcal{G})) = \mathscr{L}(\Lambda_{\forall}^{\texttt{WIN}}(\mathcal{G}))$. For Property *(ii)* we can use the same arguments: Due to the duality of the winning conditions, we know $\pi \in \texttt{WIN} \iff \pi \notin \widetilde{\texttt{WIN}}$. From Lemma 22 we know that this holds iff $\exists \zeta \in Z(\pi) : \sigma_{\mathcal{T}}(\zeta) \notin \mathscr{L}(\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}(\mathcal{G}))$. Thus, $\sigma_{\mathcal{T}}(\zeta) \in \overline{\mathscr{L}(\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}(\mathcal{G}))} = \mathscr{L}(\widetilde{\Lambda_{\exists}^{\widetilde{\texttt{WIN}}}}(\mathcal{G})) = \mathscr{L}(\Lambda_{\forall}^{\texttt{WIN}}(\mathcal{G}))$. Hence, Property *(i)* yields the result for the first direction and since for the other direction the premise holds for all firing sequences, it also holds for the special one. □

▶ **Example 19.** The UTA corresponding to the Petri game with transits depicted in Fig. 9.1 on page 141 with the *universal* safety condition $\forall$-$\texttt{SAFE}(\{\bot_M, \bot_D\})$ is very similar to the automaton presented in Fig. 12.3 on page 176. The only differences are that the acceptance condition changes to $\texttt{SAFE}(\{\bot_M, \bot_D\})$ and that in the initial state s_0 there are only loops for all $t \in \mathcal{T} \setminus \{i, r_d, r_m\}$ and for each letter i, r_d, and r_m there is a universal edge branching from state s_0 into the states s_0 *and* R.

We consider the traces $\sigma_{\mathcal{T}}(\zeta_3) = i \cdot m \cdot d_d \cdot \tau^\omega$ and $\sigma_{\mathcal{T}}(\zeta_4) = i \cdot m \cdot d_m \cdot c_m \cdot r_m \cdot d \cdot \tau^\omega$ of the covering firing sequences presented in Example 18 on page 177 of the plays π_1 and π_2 depicted in Fig. 9.2 on page 142. Both are accepted by the ETA corresponding to the existential safety condition $\exists$-$\texttt{SAFE}(\{\bot_M, \bot_D\})$. For the universal safety condition $\forall$-$\texttt{SAFE}(\{\bot_M, \bot_D\})$, trace $\sigma_{\mathcal{T}}(\zeta_3)$ is still accepted because for the corresponding run there is one branch looping in s_0 and the other goes from R to M and loops there. So no branch ever reaches $\bot_M$ or $\bot_D$. For the trace $\sigma_{\mathcal{T}}(\zeta_4)$, however, we have the branch that switches from s_0 to R, then to M, then loops once for the letter d_m, and with c_m finally reaches $\bot_M$. Hence, this branch violates the acceptance condition $\texttt{SAFE}(\{\bot_M, \bot_D\})$ of the UTA. ◀

12.2.3 Local Flow-LTL

In this section the construction of a deterministic parity automaton $\Lambda^{\text{F-}LTL}(\mathcal{G})$ from a Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{Flow-LTL}(\varphi))$ with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$ and a *local Flow-LTL* formula φ having only places as atomic propositions is presented. The construction proceeds in six steps:

1. We transform the local Flow-LTL formula φ over atomic propositions $AP = \mathcal{P}_E \cup \mathcal{P}_S$ into a form φ', where all occurrences of subformulas $\mathbb{A}\,\psi_i'$ are substituted by their equivalences $\neg\mathbb{E}\,\neg\psi_i'$ and all negations either occur within the scope of the $\mathbb{E}$-operator or are directly in front of it.
2. For each $\mathbb{E}\,\psi_i$ in φ' that is not in the scope of a negation, we construct an NBA A_{ψ_i} with size $2^{\mathcal{O}(|\psi_i|)}$ that accepts all words satisfying ψ_i [VW94].

3. We use the ideas of the ETA and the automaton A_{ψ_i} to build an extended product automaton $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ that accepts a run whenever the guessed chain satisfies ψ_i.

4. For all subformulas $\neg\mathbb{E}\,\psi_i$ in φ', we build the complement automaton $\overline{\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})}$ of the NBA $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ at the cost of an additional exponent [Mic88; Saf88].

5. We build the intersections or unions of all these NBAs $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ or $\overline{\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})}$ with respect to the operators $\wedge$ or $\vee$ used in φ' to combine the corresponding flow subformulas. This can be done with a linear respective quadratic blow-up for each operation [Cho74].

6. We determinize the NBA resulting from the previous step to obtain a DPA $\Lambda^{\text{F-}LTL}(\mathscr{G})$ at the cost of another exponent [Sch09].

The first step can easily be done using De Morgan's laws. For the second step, great efforts have been made by many researchers to improve this translation of an LTL formula ψ_i to a nondeterministic Büchi automaton A_{ψ_i} (e.g., [SB00; GO01; GL02; ST03; EWS05; BKRS12]). The third step employs the following definition of an adapted product of a general version of the ETA of Definition 62 and the NBA A_{ψ_i} of Step 2. Note that we can straight-forwardly define the product, when ψ_i does not contain any next operator or the system does not have any finite chains. In these cases, we can just loop in the first component for transitions not extending the chain and either move the second component corresponding to the stuttering in the current place (in case there is no $\bigcirc$-operator) or also loop in the second component (in case there are no finite chains). The additional states and edges are only necessary to cover the general case.

▶ **Definition 64 (Single Flow-LTL Transit Automata).** Given a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{Flow-LTL}(\varphi))$ with an arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and a Flow-LTL formula φ over atomic propositions $AP = \mathscr{P}_E \cup \mathscr{P}_S$. Furthermore, let $A_{\psi_i} = (AP, Q, q_0, \to, \texttt{BUCHI}(B))$ be the NBA constructed from an LTL formula ψ_i of a flow subformula $\mathbb{E}\,\psi_i$ of φ in Step 2.

We define the sets $S = \mathscr{P} \uplus \{s_0\}$ for tracking the chain, $S_{\mathfrak{s}} = \{p_{\mathfrak{s}} \mid p \in \mathscr{P}\}$ for the firing of *infinitely* many transitions not extending the chain, and $S_c = \{p_c \mid p \in \mathscr{P}\}$ for the firing of *finitely* many transitions not extending the chain. Then, the *single Flow-LTL transit automaton (SFTA)* $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}) = (\mathscr{T} \cup \{\top, \tau\}, \boldsymbol{\mathcal{S}}, (s_0, q_0), \delta, \texttt{BUCHI(B)})$ is defined with the set of *states* $\boldsymbol{\mathcal{S}} = ((S \uplus S_{\mathfrak{s}} \uplus S_c) \times Q) \uplus \{s_{\lightning}\}$, the *transition relation* $\delta = \mathit{init} \uplus \mathit{transit} \uplus \mathit{guessFin} \uplus \mathit{potInfy} \uplus \mathit{fin} \uplus \mathit{error} \uplus \mathit{stutter}$ with

$$
\begin{aligned}
\mathit{init} &= \{((s_0, q_0), l, (s_0, q_0)) \in \boldsymbol{\mathcal{S}} \times \mathscr{T} \times \boldsymbol{\mathcal{S}}\} \\
&\quad \cup \{((s_0, q_0), t, (s', q')) \in \boldsymbol{\mathcal{S}} \times \mathscr{T} \times \boldsymbol{\mathcal{S}} \mid (\triangleright, s') \in \Upsilon(t) \wedge (q_0, s', q') \in \to\} \\
\mathit{transit} &= \{((s, q), t, (s', q')) \in (S \times Q) \times \mathscr{T} \times \boldsymbol{\mathcal{S}} \mid (s, s') \in \Upsilon(t) \wedge (q, s', q') \in \to\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{guessFin} &= \{((s,q),t,(s',q)) \in (S \times Q) \times \mathcal{T} \times \mathcal{S} \mid \neg\exists(s,\cdot) \in \Upsilon(t) \wedge s' \in \{s_c, s_{\mathfrak{s}}\}\} \\
\mathit{potInfy} &= \{((s,q),t,(s,q)) \in (S_c \times Q) \times \mathcal{T} \times (S_c \times Q) \mid s = p_c \wedge \neg\exists(p,\cdot) \in \Upsilon(t)\} \\
&\quad \cup \{((s,q),t,(s',q')) \in (S_c \times Q) \times \mathcal{T} \times (S \times Q) \mid s = p_c \wedge (p,s') \in \Upsilon(t) \\
&\qquad\qquad \wedge (q,s',q') \in \rightarrow\} \\
\mathit{fin} &= \{((s,q),t,(s,q')) \in (S_{\mathfrak{s}} \times Q) \times \mathcal{T} \times (S_{\mathfrak{s}} \times Q) \mid s = p_{\mathfrak{s}} \wedge \neg\exists(p,\cdot) \in \Upsilon(t) \\
&\qquad\qquad \wedge (q,p,q') \in \rightarrow\} \\
\mathit{error} &= \{((s,q),t,s_{\lightning}) \in (S_{\mathfrak{s}} \times Q) \times \mathcal{T} \times \mathcal{S} \mid s = p_{\mathfrak{s}} \wedge \exists(p,\cdot) \in \Upsilon(t)\} \\
&\quad \cup \{(s_{\lightning},l,s_{\lightning}) \mid l \in \mathcal{T} \cup \{\top,\tau\}\} \\
\mathit{stutter} &= \{((s,q),l,(s,q')) \in \mathcal{S} \times \{\tau\} \times \mathcal{S} \mid (q,s,q') \in \rightarrow\} \\
&\quad \cup \{((s,q),l,(s,q)) \in \mathcal{S} \times \{\top\} \times \mathcal{S}\},
\end{aligned}
$$

and the *acceptance set* $\mathtt{B} = (S \setminus (\{s_0\} \cup S_c)) \times B$. ◀

Again, the SFTA can nondeterministically guess the desired chain in its initial state with the *init* edges. It can only leave the initial state when a chain is started. For transitions extending the chain, the *transit* edges track the chain and trigger the automaton A_{ψ_i} checking the LTL formula ψ_i accordingly. For transitions not extending the chain, the automaton guesses nondeterministically with the *guessFin* edges whether no transition is ever extending the chain again (states $S_{\mathfrak{s}} \times Q$), or only finitely many concurrent transitions are fired, before another transition extending the chain fires (states $S_c \times Q$). For the second case the *potInfy* edges are used. In this case, the automaton loops with transitions not extending the chain. Thus, it also does not trigger the automaton A_{ψ_i}. Furthermore, it returns into the tracking mode as soon as another transition extending the chain is fired. Here, also the automaton A_{ψ_i} is triggered to process the next state of the chain. In the other case the automaton stutters with the *fin* edges on the last place of the chain and the automaton A_{ψ_i} is triggered accordingly for transitions not extending the chain. In case the automaton guessed wrongly and there is another transition extending the chain, the edges *error* are used to change into a non-accepting sink state. Stuttering with the *stutter* edges covers the infinite stuttering for finite runs, where the the automaton A_{ψ_i} is triggered and the finite stuttering with $\top$-edges, where the automaton is not triggered.

In Step 4 of the construction of the transit automaton checking a local Flow-LTL formula, we build the complement automaton $\overline{\Lambda^{\mathbb{E}LTL}_{\psi_i}(\mathscr{G})}$ of the NBA $\Lambda^{\mathbb{E}LTL}_{\psi_i}(\mathscr{G})$ for all subformulas $\neg\mathbb{E}\,\psi_i$ in φ'. The optimal construction is exponential in the size of the input automaton [Mic88; Saf88; Kla91; KV01b].

In Step 5 of the construction, unions and intersections of these automata are build according to the disjunctions and conjunctions in the transformed local Flow-LTL formula φ'. Thus, we define the automaton $\Lambda^{\text{F-}LTL}_{\varphi'}(\mathscr{G})$ inductively:

$$
\Lambda^{\text{F-}LTL}_{\varphi'}(\mathscr{G}) = \begin{cases}
\Lambda^{\mathbb{E}LTL}_{\psi}(\mathscr{G}) & \text{if } \varphi' = \mathbb{E}\,\psi \\
\overline{\Lambda^{\mathbb{E}LTL}_{\psi}(\mathscr{G})} & \text{if } \varphi' = \neg\mathbb{E}\,\psi \\
\Lambda^{\text{F-}LTL}_{\varphi_1}(\mathscr{G}) \cap \Lambda^{\text{F-}LTL}_{\varphi_2}(\mathscr{G}) & \text{if } \varphi' = \varphi_1 \wedge \varphi_2 \\
\Lambda^{\text{F-}LTL}_{\varphi_1}(\mathscr{G}) \cup \Lambda^{\text{F-}LTL}_{\varphi_2}(\mathscr{G}) & \text{if } \varphi' = \varphi_1 \vee \varphi_2
\end{cases}
$$

where we consider the subformulas $\mathbb{E}\,\psi_i$ and $\neg\mathbb{E}\,\psi_i$ as atoms. Since NBA are closed under union and intersection, $\Lambda_{\varphi'}^{\text{F-}LTL}(\mathscr{G})$ is still a nondeterministic Büchi automaton. Furthermore, the union of two NBA with n_1 and n_2 states result in an NBA having $n_1 + n_2$ states and the intersection yields an NBA with $2n_1n_2$ states [Cho74]. Thus, the number of states of $\Lambda_{\varphi'}^{\text{F-}LTL}(\mathscr{G})$ is polynomial in the number of states of the automata $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ and $\overline{\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})}$. Hence, it is single-exponential in $|\psi_i|$ and polynomial in $|\mathscr{G}|$ for specifications only having $\mathbb{E}\,\psi_i$ atoms and double-exponential in $|\psi_i|$ and single-exponential in $|\mathscr{G}|$ for specifications that also contains $\neg\mathbb{E}\,\psi_i$ subformulas.

Finally, determinizing the nondeterministic Büchi automaton $\Lambda_{\varphi'}^{\text{F-}LTL}(\mathscr{G})$ yields the *Flow-LTL transit automaton (FTA)* $\Lambda^{\text{F-}LTL}(\mathscr{G})$. This step results in a deterministic parity automaton at the cost of another exponent [Sch09]. Note that we can spare also an exponent when the specification only consists of $\neg\mathbb{E}\,\psi_i$ formulas. In this case, we can first apply De Morgan to put the negation at the beginning of the formula while interchanging the conjunctions and disjunctions. We then build the corresponding intersections and unions of the NBA $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$, determinize the resulting NBA to a DPA (costing one exponent), and only afterwards complement the DPA. The complementation of a DPA can be done without any blow-up. However, the intersection and union of DPA is exponential [Bok18], so applying this approach to the general case would also add another exponent.

As for the other two transit automata, we can show that the acceptance of the FTA is independent of the specific covering firing sequence of a play triggering the automaton and that the winning property of a play corresponds to the acceptance of the automaton.

▶ **Lemma 25 (Expediency of the FTA).** *Given a play π of a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{Flow-LTL}(\varphi))$ with a local Flow-LTL φ with only places as atomic propositions, then*

(i) $\exists \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{F\text{-}LTL}(\mathscr{G})) \implies \forall \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{F\text{-}LTL}(\mathscr{G}))$, *and*

(ii) $\pi \in \texttt{Flow-LTL}(\varphi) \iff \forall \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{F\text{-}LTL}(\mathscr{G}))$

holds. ◀

Proof Sketch. Since the acceptance of a word of $\Lambda^{\text{F-}LTL}(\mathscr{G})$ boils down to the acceptance of the single Flow-LTL transit automata $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$, it suffices to show that these automata satisfy the properties. Property *(i)* can be shown analogously to the proof of Lemma 22 for the ETA. The automaton A_{ψ_i} is only triggered for transitions extending the chain or in case of a finite chain. No relevant state changes occur for transitions not extending the chain and the ordering of these transitions are the only difference of two covering firing sequences of a play. For Property *(ii)*, we can also show both directions similarly as in the proof of Lemma 22 for the ETA. We can create the desired flow chain or the accepting run quite directly from the other. The triggering of the automaton A_{ψ_i} is done exactly corresponding to the trace of the flow chain. Since the composition of $\Lambda^{\text{F-}LTL}(\mathscr{G})$ from the SFTA $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ fits the composition of the formula both directions directly follow from the properties of the SFTA. For more details see the proof on page 196. □

12.3 Decision Procedure

In this section we combine the two concepts presented in Sec. 12.1 and Sec. 12.2. Given a Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$, we create the information flow game $G(\mathcal{A}) = (A(\mathcal{A}), \texttt{BUCHI}(V_1))$ with arena $A(\mathcal{A}) = (V, V_0, V_1, E, I)$ and the deterministic transit automaton $\Lambda(\mathcal{G})$ corresponding to the transits of $\mathcal{A}$ and the winning condition $\texttt{WIN}$. The parallel composition of those two elements yields the product game $\mathbb{G}(\mathcal{G}) = G(\mathcal{A}) \mid\mid \Lambda(\mathcal{G})$ serving to solve the initial question, whether the system players of the Petri game with transits $\mathcal{G}$ have a deadlock-avoiding winning strategy and calculating a strategy if it exists.

To have a uniform presentation, we consider all deterministic transit automata of Sec. 12.2.1, Sec. 12.2.2, and Sec. 12.2.3 as special cases of deterministic parity automata, regardless of their actual acceptance condition. The less expressive conditions can easily be expressed as parity conditions. We denote these automata uniformly by $\Lambda(\mathcal{G})$. For the complexity results in Theorem 9, we come back to the actual conditions.

▶ **Definition 65 (Product Game).** Let $\mathcal{G} = (\mathcal{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\circ|\circ}$ be Petri game with transits with arena $\mathcal{A} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, \Upsilon, In)$, $G(\mathcal{A}) = (A(\mathcal{A}), \texttt{BUCHI}(V_1))$ be the corresponding information flow game with arena $A(\mathcal{A}) = (V, V_0, V_1, E, I)$, and $\Lambda(\mathcal{G}) = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \to, \texttt{PARITY}(\Omega))$ be the corresponding deterministic parity automaton. We define the product game $\mathbb{G}(\mathcal{G}) = (\mathbb{A}, \texttt{CONJ}(\texttt{PARITY}(\Omega_1), \texttt{PARITY}(\Omega_2)))$ with arena $\mathbb{A} = (V \times S, V_0 \times S, V_1 \times S, \to', (I, s_0))$ with $((v, s), l, (v', s')) \in \to'$ iff $(v, l, v') \in E$ and $(s, l, s') \in \to$, the parity functions $\Omega_1 : V \times S \to \mathbb{N}$ with

$$\Omega_1((v, s)) = \begin{cases} 0 & \text{if } v \in V_1 \\ 1 & \text{otherwise} \end{cases},$$

and $\Omega_2((v, s)) = \Omega(s)$. We abbreviate this construction by $\mathbb{G}(\mathcal{G}) = G(\mathcal{A}) \mid\mid \Lambda(\mathcal{G})$. ◀

Note that we can end up with simpler winning conditions of the product game for simpler winning conditions of the Petri game with transits. For example, for a universal Büchi condition, we can use a generalized Büchi condition rather than a generalized parity condition, which can be solved in polynomial time [BCGH+10; CH14; CDHL16] rather than in NP ∩ co-NP [CHP07; SSR08]. In fact, there is no need for the generalized parity condition at all because a conjunction of a Büchi condition and a parity condition can also be expressed with a single parity condition at the cost of an exponential blow-up of the state space with respect to the number of colors (cp. the complexity proof on page 200).

By projecting on the single components of a play and exploiting the results for the information flow game and the transit automata, we can show that Petri games with transits and local winning conditions are decidable.

▶ **Lemma 26 (Reduction).** *For an arena $\mathcal{A}$ and a local winning condition $\texttt{WIN}$ of a Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\circ|\circ}$, the product game $\mathbb{G}(\mathcal{G}) = G(\mathcal{A}) \mid\mid \Lambda(\mathcal{G})$ has a winning strategy for player 0 iff $\mathcal{G}$ has a deadlock-avoiding winning strategy for the system players.* ◀

Proof Sketch. For the *completeness* proof, i.e., direction "if", we use Theorem 8 to obtain a winning strategy $\sigma_{\mathtt{inf}} = \sigma_{\mathtt{pg2inf}}(\sigma)$ of player 0 in the information flow game $G(\mathscr{A})$ from the deadlock-avoiding winning strategy σ of the system players in $\mathscr{G}$. We can construct a strategy $\sigma_{||}$ for the product game by using $\sigma_{\mathtt{inf}}$ to define the successor $v \in V$ of the first component and by adding the successor of the second component according to the deterministic decision of $\Lambda(\mathscr{G})$ with respect to the label used in $G(\mathscr{A})$ to reach the successor v. We can show that $\sigma_{||}$ is winning for player 0 by using that $\sigma_{\mathtt{inf}}$ is winning, that Definition 61 ensures the correspondence of the edge labels of a play conforming to $\sigma_{\mathtt{inf}}$ and a covering firing sequence of a maximal play π conforming to the winning strategy σ, and using either Lemma 22, Lemma 24, or Lemma 25, to show the acceptance of the trace of the firing sequence. Neither the finitely many $\top$-labels distinguishing the sequence of edge labels from the covering firing sequence do affect the acceptance of the automaton, nor the possible τ-loops that can only occur infinitely often at the end of the word prevent $\sigma_{||}$ from being winning.

For the *soundness* proof, i.e., direction "only if", we can create a winning strategy $\sigma_{\mathtt{inf}}$ of $G(\mathscr{A})$ from $\sigma_{||}$ by focusing on the first component due to $\Lambda(\mathscr{G})$ being deterministic. Theorem 7 yields $\sigma = \sigma_{\mathtt{inf2pg}}(\sigma_{\mathtt{inf}})$ is a deadlock-avoiding strategy for the system players of $\mathscr{G}$. Due to Definition 59, all plays π conforming to σ have a covering firing sequence ζ that corresponds to the selection taken by $\sigma_{\mathtt{inf}}$. Thus, the word of the edge labels of the corresponding play conforming to $\sigma_{||}$ is accepted by the automaton. Since again the $\top$- and τ-labels does not make any harm, $\sigma_{\mathscr{T}}(\zeta)$ is accepted. Hence, either Lemma 22, Lemma 24, or Lemma 25 yield that π is winning and so, σ is winning. For more details see the proof on page 199. □

▶ **Example 20.** We proceed with the running example of Fig. 9.1 on page 141. Determinizing the ETA of Example 17 results in a deterministic co-Büchi automaton with 97 states. However, we can still see that the winning strategy $\sigma^1_{\mathtt{inf}}$ presented in Fig. 12.2 on page 168 of the information flow game corresponds to a winning strategy in the product game, while the winning strategy $\sigma^2_{\mathtt{inf}}$ does not. For $\sigma^1_{\mathtt{inf}}$, we have the words $w_1 = \top \cdot i \cdot m \cdot d_d \cdot \tau^\omega$ and $w_2 = \top \cdot i \cdot m \cdot d_m \cdot c_m \cdot \top \cdot r_m \cdot d \cdot \tau^\omega$ constructed from the edge labels of the plays conforming to $\sigma^1_{\mathtt{inf}}$. For both sequences we can see, as in Example 18 for the corresponding firing sequences ζ_3 and ζ_4, that the ETA accepts because the additional finitely many $\top$-steps do not have any impact on the acceptance. Observe that in this case the determinization of the automaton is crucial. Building the product with the nondeterministic version enforces to choose whether to track the chain started while firing transition i or not already at the moment i fires. This means that the decision has to be taken before the environment destroys one of the tools. Hence, the decision is taken for both plays uniquely. However, to have the word accepted, we need to track the chain started with transition i for w_1 and the chain started with transition r_m for w_2. The deterministic version delays this decision from the point of the guessing to the relevant point for the acceptance.

For $\sigma^2_{\mathtt{inf}}$, there is additionally to w_1 the word $w_3 = \top \cdot i \cdot m \cdot d_m \cdot (c_m \cdot \top \cdot r_m \cdot m)^\omega$ for the second play conforming to $\sigma^2_{\mathtt{inf}}$. However, for this input word there is no accepting run of the ETA because no matter which chain is tracked (either started with i or any instance

of r_m), all will eventually reach $\bot_M$ and the ETA loops in this state. Deciding to track no chain, i.e., staying in s_0 is also no possibility because s_0 is part of the co-Büchi states.

Note that in this example the positional strategy $\sigma^1_{\texttt{inf}}$ already suffices for having a winning strategy of the product game. In general, it might be necessary to use a loop once, for example. Then, this strategy of the information flow game with memory may result in a memoryless strategy in the product game because the necessary memory is stored in the states of the automaton. ◀

This reduction provides the solving mechanism for Petri games with transits and local objectives on their data flow. For the complexity, we simplify the generalized parity winning condition $\texttt{CONJ}(\Omega_1, \Omega_2)$ of the product game to a generalized Büchi, a parity, or even a Büchi condition depending on the winning conditions of the Petri game with transits, to provide faster solving algorithms and memoryless strategies. For local Flow-LTL, the complexity depends on the type of specifications used for the local data flow in the formula. Consider a local Flow-LTL formula φ in a form, where all negations are within the scope of an $\mathbb{E}$ or an $\mathbb{A}$ operator. Then, we have a lower complexity in case φ only contains $\mathbb{E}$ operators or only $\mathbb{A}$ operators. This means, it is easier to solve games that have only existential requirements on the local data flow or only universal requirements but no mixtures.

▶ **Theorem 9.** *For a Petri game with transits* $\mathcal{G} = (\mathscr{A}, \texttt{WIN}) \in \mathcal{G}^{1,b}_{s,\text{o}|\text{o}}$, *answering the question whether the system players have a deadlock-avoiding winning strategy is* EXPTIME-complete *for all* place-based *winning conditions*

$$\texttt{WIN} \in \{\exists\text{-}\texttt{SAFE}(\mathcal{S}), \forall\text{-}\texttt{SAFE}(\mathcal{S}), \exists\text{-}\texttt{REACH}(\mathcal{S}), \forall\text{-}\texttt{REACH}(\mathcal{S}), \exists\text{-}\texttt{BUCHI}(\mathcal{S}), \forall\text{-}\texttt{BUCHI}(\mathcal{S}),$$
$$\exists\text{-}\texttt{COBUCHI}(\mathcal{S}), \forall\text{-}\texttt{COBUCHI}(\mathcal{S}), \exists\text{-}\texttt{PARITY}(\Omega), \forall\text{-}\texttt{PARITY}(\Omega)\}$$

with $\mathcal{S} \subseteq \mathscr{P}_S \cup \mathscr{P}_E$ *and* $\Omega : \mathscr{P}_S \cup \mathscr{P}_E \to \mathbb{N}$.

For $\texttt{WIN} = \texttt{Flow-LTL}(\varphi)$, *for a local Flow-LTL formula* φ *with only places as atomic propositions, answering the question is* single-exponential *in* $|\mathscr{A}|$ *and* double-exponential *in* $|\varphi|$, *when* φ *only has existential local data flow specifications or only universal local data flow specification. Otherwise it is* double-exponential *in* $|\mathscr{A}|$ *and* triple-exponential *in* $|\varphi|$.

In case a winning strategy for the system players exists, it can by constructed in single-exponential *time.* ◀

Proof Sketch. Both, the ETA and the UTA are single-exponential in $|\mathscr{A}|$ due to the determinization step. The information flow arena is also single-exponential in $|\mathscr{A}|$. The conjunction of the Büchi condition of the information flow game and the corresponding acceptance condition of the transit automaton may result in a Büchi, a generalized Büchi, or a parity condition. These games can be solve in polynomial time in the size of the arena and possibly in single-exponential time in the number of colors. The numbers of colors stemming from the reduction may also only be linear in $|\mathscr{A}|$. For local Flow-LTL the complexity directly follows from the size of the transit automaton. Due to the property of such games having memoryless strategies, a finite representation of the corresponding Petri game with transit strategy can be constructed as in Definition 59 in single-exponential time. For more details see the proof on page 200. □

12.4 Proofs

In this section we provide the proofs of the properties of the information flow game (Sec. 12.1.3), of the transit automata (Sec. 12.2), and of the decision procedure for Petri games with transits (Sec. 12.3).

For the properties of the information flow game, we start by showing the single-exponential size of the arena.

Proof (Lemma 21: Size of an Information Flow Arena): Definition 57 and Eq. 12.1 yield that each state $v \in V$ is a set of decisions from $\{(p, 0, post(p)) \in \mathscr{P}_E \times \{0\} \times 2^{\mathscr{T}}\} \cup \{(p, i, C) \in \mathscr{P}_S \times \{0, \dots, \mathtt{max}_S\} \times (2^{\mathscr{T}} \cup \{\top\}) \mid C \subseteq post(p) \vee C = \top\}$ equipped with a round identifier in $\{0, \dots, |\mathscr{T}|\}$ and last transition in $\mathscr{T} \cup \{\tau, \top\}$. Since $\mathscr{A}$ is 1-bounded, there can only be $k = \mathtt{max}_S + 1$ decisions in one set. We have at most $|\mathscr{P}| \cdot (\mathtt{max}_S + 1) \cdot \left(2^{|\mathscr{T}|} + 1\right) \leq |\mathscr{P}| \cdot (\mathtt{max}_S + 1) \cdot 2 \cdot 2^{|\mathscr{T}|} = n$ possible different decisions. Since we consider sets of these decisions and $\mathscr{A}$ is 1-bounded, a reachable state must be one of the combinations for drawing k decisions from the set of n possible decisions without repetition and without considering the order. There are $\binom{n}{k} \leq n^k$ ways to do so. Hence, there are at most $\left(|\mathscr{P}| \cdot (\mathtt{max}_S + 1) \cdot 2 \cdot 2^{|\mathscr{T}|}\right)^{\mathtt{max}_S+1} \cdot (|\mathscr{T}| + 1) \cdot (|\mathscr{T}| + 2) = |\mathscr{P}|^{\mathtt{max}_S+1} \cdot (\mathtt{max}_S + 1)^{\mathtt{max}_S+1} \cdot 2^{\mathtt{max}_S+1} \cdot 2^{|\mathscr{T}| \cdot (\mathtt{max}_S+1)} \cdot (|\mathscr{T}| + 1) \cdot (|\mathscr{T}| + 2)$ possible reachable states in $A(\mathscr{A})$. □

We now show that the Petri net and the labeling function constructed in Definition 59 indeed represent a strategy of the given Petri game with transits.

Proof (Theorem 7: From Information Flow Game to Petri Game): Let $\mathscr{G}$ be a Petri game with transits with arena $\mathscr{A} = (\mathscr{P}_S, \mathscr{P}_E, \mathscr{T}, \mathscr{F}, \Upsilon, In)$ and $\sigma_{\mathtt{inf}}$ a winning strategy of player 0 in the information flow game $G(\mathscr{A}) = (A(\mathscr{A}), \mathtt{BUCHI}(V_1))$ with its arena $A(\mathscr{G}) = (V, V_0, V_1, E, I)$. We show that $\sigma = \sigma_{\mathtt{inf2pg}}(\sigma_{\mathtt{inf}}) = (\mathscr{N}^\sigma, \lambda^\sigma)$ is a deadlock-avoiding strategy of the system players in $\mathscr{G}$.

Subprocess: Firstly, we show that σ is a *subprocess* of an unfolding $\beta_U = (\mathscr{N}^U, \lambda^U)$ of $\mathscr{A}$. The *subnet property* can be easily seen by choosing the unfolding according to the names of the newly created nodes in the construction. The flow relation also satisfies the subnet property due to the construction only adding flows according to the original flows. The correspondence of the labeling functions, i.e., $\lambda^\sigma = \lambda^U_{|\mathscr{P}^\sigma \cup \mathscr{T}^\sigma}$, follows from having the same names and mapping the nodes to its origins in the construction.

To show that σ is an *initial branching process*, we first show that λ^σ is *initial*, i.e., $\lambda^\sigma(In^\sigma) = In$ and λ^σ is a *homomorphism*. The induction anchor **(IA)** maps the, in this step newly created, places $p^\sigma \in \mathscr{P}^\sigma$ (stored as In^σ) for each $p \in \mathtt{M}(l(r)) = \mathtt{M}(I) = In$ to p. Hence, $\lambda^\sigma(In^\sigma) = In$ holds. The construction satisfies the homomorphism property for the *type preserving of the nodes*, because only places are mapped to places and transitions to transitions, i.e., $\lambda^\sigma(\mathscr{P}^\sigma) \subseteq \mathscr{P}$ and $\lambda^\sigma(\mathscr{T}^\sigma) \subseteq \mathscr{T}$ holds. Also the *pre- and postset preserving* feature is given because **(IS)** only connects the freshly added transition $t' \in \mathscr{T}^\sigma$ with $\lambda^\sigma(t') = t$ to places in its preset $p \in \mathscr{P}^\sigma$ with $\lambda^\sigma(p) \in pre^{\mathscr{A}}(t) = pre^{\mathscr{A}}(\lambda^\sigma(t'))$. Hence, $\lambda^\sigma(pre^\sigma(t')) = pre^{\mathscr{A}}(\lambda^\sigma(t'))$ holds. The same holds for the postset

because **(IS)** only connects the newly created postset places $P' = post^{\sigma}(t')$ such that $\lambda^{\sigma}(P') = post^{\mathscr{A}}(t) = post^{\mathscr{A}}(\lambda^{\sigma}(t))$ and this is the only situation in the construction where postset edges are added to transitions. Hence, $\lambda^{\sigma}(post^{\sigma}(t')) = post^{\mathscr{A}}(\lambda^{\sigma}(t'))$ holds. Furthermore, λ^{σ} is also *injective on transitions with the same preset*, i.e., $\forall t_1, t_2 \in \mathscr{T}^{\sigma} : (pre^{\sigma}(t_1) = pre^{\sigma}(t_2) \wedge \lambda^{\sigma}(t_1) = \lambda^{\sigma}(t_2)) \Rightarrow t_1 = t_2$ holds. Assume there are two different transitions t_1 and t_2 satisfying these properties. Then, these transitions can only occur in different branches of the to $\sigma_{\texttt{inf}}$ corresponding strategy tree T_{σ} because in each step **(IS)** the new transition t' is connected to the places in the corresponding cut $c(v_T)$ and the successor cut is generated by creating *fresh* places in $post^{\sigma}(t')$ and by firing t'. Hence, $pre^{\sigma}(t_1) = pre^{\sigma}(t_2)$ can only be satisfied for transitions in different branches. But we cannot have two *different* transitions mapping to the same original transition ($\lambda^{\sigma}(t_1) = \lambda^{\sigma}(t_2)$) in different branches while preserving the preset due to the scheduling in the `type`-1 case, and due to adding the successor for one branch with `gen` in the `type`-2 case. Thus, λ^{σ} satisfies the injective property.

To show that $\mathscr{N}^{\sigma}$ is an *occurrence net* we check that each place only has *at most one ingoing arc*, i.e., $\forall p \in \mathscr{P}^{\sigma} : |pre^{\sigma}(p)| \leq 1$. In the construction, transitions are only added to the preset of any place $p \in \mathscr{P}^{\sigma}$ in step **(IS)**. Here, this is only done for the newly created places. Thus, no two arcs could merge into the same place. There is also no transition in *self-conflict*, i.e., $\neg\exists p \in \mathscr{P}^{\sigma} : \exists t_1, t_2 \in post^{\sigma}(p) : t_1 \neq t_2 \wedge t_1 \leq t \wedge t_2 \leq t$, because we only add transitions occurring in the strategy tree T_{σ} and these transitions belong to firing sequences in $\mathscr{A}$. The construction only adds places and transitions according to these firing sequences, and since each step only adds fresh nodes, there is no possibility to have a place where a flow branches and these flows later merge again. Moreover, there can be no *self-loops*, i.e., $\forall x \in \mathscr{P}^{\sigma} \cup \mathscr{T}^{\sigma} : \neg(x < x)$ because the construction only adds fresh nodes and adds only postset edges in **(IS)** to these newly created nodes. Thus, we cannot get back with the flow relation to any previously added node. This is similar for the *infinitely decreasing sequence* property. The construction starts with fresh places in **(IA)**, in each step, **(IS)** adds at most one transition to the postset of the previously freshly created places, and again creates new fresh places in the postset of this transition. Hence, starting in any node $n \in \mathscr{P}^{\sigma} \cup \mathscr{T}^{\sigma}$, there is no infinite path following the flow relation $\mathscr{F}^{\sigma}$ backwards. Lastly, also the *initial places* are the only ones which do not have any predecessor, i.e., $In^{\sigma} = \{p \in \mathscr{P}^{\sigma} \mid pre^{\sigma}(p) = \emptyset\}$: In **(IS)** each newly created place is connected via $\mathscr{F}^{\sigma}$ to the newly added transition t'. Thus, all new places have this transition in their preset and therewith only the places created in **(IA)** have no predecessor. Hence, in total σ is a *subprocess* of an unfolding β_U of $\mathscr{A}$.

We now show that σ fulfills the three conditions *justified refusal*, *determinism*, and *deadlock-avoidance*.

Justified Refusal: To show that the system players only *refuse* to play transitions of the unfolding which are not violating the game play, i.e., they do not disallow pure environment transitions and do not allow an instance of a transition for which they already refused another instance in the same situation (place), we have to show that $\forall t \in \mathscr{T}^{U} : t \notin \mathscr{T}^{\sigma} \wedge pre^{U}(t) \subseteq \mathscr{P}^{\sigma} \implies \exists p \in pre^{U}(t) \cap \mathscr{P}^{\sigma}_{S} : \forall t' \in post^{U}(p) : \lambda^{U}(t) = \lambda^{U}(t') \implies t' \notin \mathscr{T}^{\sigma}$ holds. Let $t \in \mathscr{T}^{U}$. This means there must be a sequence of

transitions with $In[t_0, t_1, \ldots, t_n\rangle M$ such that $M[\lambda^U(t)\rangle$. Due to the construction in Definition 59, we know that $pre^U(t) \subseteq \mathscr{P}^\sigma$ yields that all $p \in pre^U(t)$ have been added via the construction. We consider two cases: (i) there is no node $v_T \in T$ with $pre^U(t) \subseteq c(v_T)$ and (ii) there is such a node.

Case (i): Since all places has been created, but there is no node still having all places in the corresponding cut, there must be a transition $t^\star \in \mathscr{T}^\sigma$ which takes tokens of $pre^U(t)$ before all places $p \in pre^U(t)$ are created in the construction. Thus, $pre^\sigma(t^\star) \cap pre^U(t) \neq \emptyset$. Let $v_T^\star \in T$ be the node where $t^\star$ is added to the strategy and $C^\star = c(v_T^\star)$ the corresponding cut. We denote the set of places in the preset of t created before the firing of $t^\star$ with $X^{pre} = C^\star \cap pre^U(t)$ and the ones which are later created with $X^{post} = pre^U(t) \setminus C^\star$. We know that $X^{post} \neq \emptyset$ because otherwise $pre^U(t) \subseteq c(v_T^\star)$. All places in $X^{pre} \cup pre^\sigma(t^\star)$ are pairwise concurrent due to $pre^\sigma(t^\star) \subseteq C^\star$. Also all places in $X^{post} \cup pre^\sigma(t^\star)$ must be pairwise concurrent, because otherwise either the places in X^{post} are causally dependent on the tokens used in the firing of $t^\star$, thus t could never be fireable and therewith $t \notin \mathscr{T}^U$, or the places are created in different branches and are depend on the branching, thus again $t \notin \mathscr{T}^U$.
Assume $\exists p_E \in pre^\sigma(t^\star) \cap \mathscr{P}_E^\sigma$. If t is of `type-2`, then $t^\star$ is of `type-2` with the same generation because $pre^U(t) \cap pre^\sigma(t^\star) \neq \emptyset$ and the firing relation $D[t^\star\rangle$ ensures no intergenerational firing. Contradiction. Otherwise, $l(v_T^\star) \in V_1$ and $t^\star$ is an environment transition. Thus, all not `type-2` typed system tokens must have maximally progressed and are directly or indirectly dependent on $t^\star$. Hence, all places $p \in X^{post}$ depend on $t^\star$, but $pre^\sigma(t^\star) \cap pre^U(t) \neq \emptyset$, thus t cannot be fireable ergo $t \notin \mathscr{T}^U$. *Contradiction.*
So, $pre^\sigma(t^\star) \subseteq \mathscr{P}_S^\sigma$. Thus, also $pre^U(t) \subseteq \mathscr{P}_S^\sigma$ must hold because otherwise there is a system place $p \in pre^\sigma(t^\star) \cap pre^U(t)$ which allows mixed communication.
Assume $\forall p \in pre^U(t) \cap \mathscr{P}_S^\sigma : \exists t' \in post^U(p) : \lambda^U(t') = \lambda^U(t) \wedge t' \in \mathscr{T}^\sigma$, which is the negation of the conclusion of the justified refusal property. Since t only has system places in its preset, each place in $pre^U(t)$ and especially each place X^{post}, has such a transition t' in its postset. Thus, consider a place $p \in X^{post}$ with transition $t' \in post^U(p)$ satisfying the conjunction. *Assume* $t' \neq t$. Due to the injectivity property of λ^U, we know $pre^U(t) \neq pre^U(t')$. Since $|pre^U(t)| = |pre^\sigma(t')|$, there must be a place $p' \in pre^U(t) \setminus pre^\sigma(t')$ because $t' \in \mathscr{T}^\sigma$. The place p' cannot be in X^{post} because then p and p' would be concurrent and thus, $p' \in pre^\sigma(t')$. Thus, $p' \in X^{pre}$ and especially $p' \in C^\star$. The place p' cannot be in $pre^\sigma(t^\star)$ because then p and p' would be concurrent and thus $p' \in pre^\sigma(t')$. So $p' \in (C^\star \cap pre^U(t)) \setminus pre^\sigma(t^\star)$ holds. This means p' is already existing in cut $C^\star$ and is not stolen by $t^\star$. So either it is still in the cut where the last place of X^{post} is created and thus p and p' are concurrent and again $p' \in pre^\sigma(t')$. Otherwise, there must be another transition $t_2^\star$ taking p' before all token of $pre^U(t)$ are created and we can start case (i) from the beginning with $t_2^\star$. This can only be done finitely often, due to the eventually creation of all place of $pre^U(t)$. *Contradiction.* Thus, $t = t'$ yields the final *contradiction*, because $t \notin \mathscr{T}^\sigma$.

Case (ii): There is a node $v_T \in T$ with $pre^U(t) \subseteq c(v_T)$. Let $l(v_T) = (D, \curlywedge, \mathfrak{t})$ and thus, $pre^{\mathscr{A}}(\lambda^U(t)) \subseteq \mathscr{M}(D)$.

Case $pre^U(t) \subseteq \mathscr{P}_E^\sigma$*:* Since $\exists p \in pre^U(t) \cap \mathscr{P}_S^\sigma$ is not satisfied, we have to show that $pre^U(t) \subseteq \mathscr{P}^\sigma \implies t \in \mathscr{T}^\sigma$ holds. There must be at least one successor reached by an edge of $\texttt{SYS}_\top, \texttt{SYS}, \texttt{SYS}_2$ or $\texttt{ENV}$ for $l(v_T)$ because otherwise D would be a deadlock and thus, $\sigma_{\texttt{inf}}$ would not be winning.
Case $l(v_T) \in V_1$*:* Since $l(v_T)$ is a player 1 state, all successors have to be in $\sigma_{\texttt{inf}}$ and must be obtained via $\texttt{SYS}_2$ or $\texttt{ENV}$. If $\rho = 0 \vee \neg\exists t' \in \mathscr{T} : D[t'\rangle_2$, then there must be an edge $((D, \rho, \mathbf{t}), \lambda^U(t), (D', \rho', \lambda^U(t))) \in \texttt{ENV}$ because $D[\lambda^U(t)\rangle_1$ holds. Hence, Definition 59 yields $t \in \mathscr{T}^\sigma$. Case $\rho \neq 0 \wedge \exists t' \in \mathscr{T} : D[t'\rangle_2$. If $\rho = 1$, then $nxt(\rho, D)$ will find this t' (or first another $t'' \in \mathscr{T}$ with $D[t''\rangle_2$) and there must be a $\texttt{SYS}_2$ edge. For this edge there is either another $t'' \in \mathscr{T}$ with $D[t''\rangle_2$ and ρ' is incremented or $\rho' = 0$. This edge does not move any token in $pre^{\mathscr{A}}(\lambda^U(t)) \subseteq \mathscr{P}_E$ because $\texttt{SYS}_2$ edges only move system tokens. Hence, for $\rho' = 0$ we can move to the previous case and otherwise we are again in this case with $\rho > 1$ (both after the $\top$-resolution, see case below). If $\rho \neq 1$, then we previously had to be in a state with $\rho = 1$ because I starts with $\rho = 1$, $\texttt{ENV}$ only sets $\rho = 1$, $\texttt{SYS}_2$ only increments ρ (when it does not reset it), and there is no other situation where ρ is set. Hence, there must be a transition $t'' \in \mathscr{T}$ with $t'' = nxt(\rho, D)$. So, there is again a $\texttt{SYS}_2$ edge, and with the same arguments as above we are again either in the previous or in this case. There cannot be an infinite sequence of only $\texttt{SYS}_2$ and $\texttt{SYS}_\top$ edges always in the case $\rho \neq 0 \wedge \exists t' \in \mathscr{T} : D[t'\rangle_2$ because the function nxt traverses the finite and ordered list $\langle\mathscr{T}\rangle$ in ascending order. So, eventually ρ is set to zero and hence, t is eventually added to the strategy.
Case $l(v_T) \in V_0$*:* The one existing edge must be from $\texttt{SYS}_\top$ or $\texttt{SYS}$. In both cases no token in $pre^{\mathscr{A}}(\lambda^U(t)) \subseteq \mathscr{P}_E$ is moved because $\texttt{SYS}_\top$ edges do not move any tokens and $\texttt{SYS}$ edges move only system tokens. There cannot be infinitely many $\texttt{SYS}_\top$ or $\texttt{SYS}$ edges in the future of v_T without eventually reaching a node $v_T' \in T$ with $l(v_T') \in V_1$ because otherwise the Büchi condition would be violated and thus, $\sigma_{\texttt{inf}}$ would not be winning. Hence, we eventually end up in a node $v_T' \in T$ with $l(v_T') \in V_1$ and $pre^U(t) \subseteq c(v_T')$.
Case $\exists p \in pre^U(t) \cap \mathscr{P}_S^\sigma$*:* Let $t \notin \mathscr{T}^\sigma$, $pre^U(t) \subseteq \mathscr{P}^\sigma$, and $t' \in post^U(p)$ with $\lambda^U(t) = \lambda^U(t')$ hold. We show that $t' \notin \mathscr{T}^\sigma$ holds. Definition 59 yields that $t \notin \mathscr{T}^\sigma$ implies that a) $D[\lambda^U(t)\rangle_2$ but $i \notin \texttt{gen}(v_T)$ for the generation i corresponding to the firing of $\lambda^U(t)$ or b) for none of the nodes $v_T' \in T$ with $pre^U(t) \subseteq c(v_T')$ there is a successor edge $(v_T', \lambda^U(t), v_T'') \in E_T$. We first show that case a) cannot happen, i.e., it is not possible that there is a node in the tree with a decision allowing the firing of a type-2 transition, but the transition does not occur in the strategy.
When $\lambda^U(t)$ is type-2 fireable in D, there must have been previously a $\texttt{SYS}_\top$ edge choosing the generation $i \in \{1, \ldots, \texttt{max}_S\}$ and thus, the construction adds i to the generations of the successor node. The generations are passed on in the construction to the successors and only in nodes corresponding to V_1 states exactly one successor is chosen. All type-2 fireable transitions in states of a winning strategy are independent from the branching in V_1 states because they have proven to never depend on the environment again (untruthfully deciding for the generation lead to loosing behavior in the two-player game). Hence, there is another node v_T', with $l(v_T') = (D', \rho', \mathbf{t}')$, where $D'[t\rangle_2$ holds, also $i \in \texttt{gen}(v_T')$, and still $pre^U(t) \subseteq c(v_T')$ is given. Hence, transition t is added to $\mathscr{T}^\sigma$,

which yields a contradiction. In case b) we consider two cases:
Case $\neg D(\lambda^U(t)\rangle$: There must be a $p' \in pre^U(t) \cap \mathscr{P}_S^\sigma$ with $(\lambda^U(p'), \cdot, C) \in D$ with $\lambda^U(t) \notin C$. W.l.o.g. $p' = p$. As long as the token in p' is not moved, no new commitment set can be chosen. Hence, $\lambda^U(p')$ forbids all instances of $\lambda^U(t)$ and thus, $t' \notin \mathscr{T}^\sigma$.
Case $D(\lambda^U(t)\rangle$: This means $D[\lambda^U(t)\rangle$ but is never fired. Thus, either there is an infinite path in $\sigma_{\texttt{inf}}$ that starts in $l(v_T)$ and is not taking any tokens of the preset of $\lambda^U(t)$, or there is a transition $t^\star$ taking a token from $pre^U(t)$, because otherwise a deadlock would be encountered and $\sigma_{\texttt{inf}}$ would not be winning. In the former case $\lambda^U(t)$ is enabled in every state $v'_T \in T$ of the infinite path and $pre^U(t) \subseteq c(v'_T)$ because no token of the preset is taken. Hence, $t' \notin \mathscr{T}^\sigma$ because $p \in pre^U(t') \cap pre^U(t)$. In the latter case $t^\star \neq t'$ because there cannot be two instances of the same transition $\lambda^U(t)$ enabled in the same cut of an unfolding of a 1-bounded Petri net. If $t^\star$ takes the token of p, transition $t' \notin \mathscr{T}^\sigma$. Otherwise, $t^\star$ takes a token residing in another place $q \neq p \in pre^U(t)$. If $q \in \mathscr{P}_S^U$ then D would be nondeterministic and thus, $\sigma_{\texttt{inf}}$ not winning. Contradiction. If no system token is taken, thus $q \in \mathscr{P}_E^U$, then this node would correspond to a player 1 state. Thus, all postset edges would be considered in $\sigma_{\texttt{inf}}$ and therewith $t \in \mathscr{T}^\sigma$ holds, which also is a contradiction.

Determinism: To show that the strategy σ is *deterministic*, we show that there is no cut in the strategy with a system place where two different transitions are enabled, i.e., $\forall p \in \mathscr{P}_S^\sigma, C \in \mathscr{R}(\mathcal{N}^\sigma) : p \in C \implies \exists^{\leq 1} t \in post^\sigma(p) : pre^\sigma(t) \subseteq C$. Assume, there is such a place $p \in \mathscr{P}_S^\sigma$ and a cut $C \in \mathscr{R}(\mathcal{N}^\sigma)$ with $p \in C$ and two different transitions $t_1, t_2 \in post^\sigma(p)$ with $t_1 \neq t_2$, $pre^\sigma(t_1) \subseteq C$, and $pre^\sigma(t_2) \subseteq C$. Due to the construction of σ in Definition 59, there must be two nodes $v_{T_1}, v_{T_2} \in T$ with $pre^\sigma(t_1) \subseteq c(v_{T_1})$ and $pre^\sigma(t_2) \subseteq c(v_{T_2})$ and, two edges $e_1 = (v_{T_1}, \lambda^\sigma(t_1), v'_{T_1}) \in E_T$ and $e_2 = (v_{T_2}, \lambda^\sigma(t_2), v'_{T_2}) \in E_T$, because $t_1, t_2 \in \mathscr{T}^\sigma$ must have been added to the strategy. Let $l(v_{T_1}) = (D_1, \curlywedge_1, \lightning_1)$ and $l(v_{T_2}) = (D_2, \curlywedge_2, \lightning_2)$. Furthermore, we know that if t_x for $x \in \{1, 2\}$ is `type-2`-firable, i.e., $D_x[\lambda^\sigma(t_x)\rangle_2$, then the corresponding generation $i_x \in \{1, \ldots, \texttt{max}_S\}$ is annotated to the node v_{T_x}, i.e., $i_x \in \texttt{gen}(v_{T_x})$.
Case $e_1 < e_2$: Let e_1 be a predecessor edge of e_2. Since t_1 has fired first and $p \in pre^\sigma(t_1)$, the token on p is removed and can never be put back because σ is a branching process. Hence, t_2 cannot fire in any future of e_1. Thus, $t_2 \notin \mathscr{T}^\sigma$. *Contradiction.* The other direction ($e_1 > e_2$) is analog.
Case $e_1 \mid\mid e_2$: Let e_1 be in a different branch than e_2. The nodes v_{T_1} and v_{T_2} must be different because otherwise $D[t_1\rangle$ and $D[t_2\rangle$ holds, and thus, D would be nondeterministic and therewith $\sigma_{\texttt{inf}}$ not winning. Thus, $v_{T_1} \neq v_{T_2}$. Let $v_{T_E} \in T$ be the last common predecessor of v_{T_1} and v_{T_2} with $l(v_{T_E}) \in V_1$. Furthermore, let $t'_1 \in \mathscr{T}^\sigma$ be the environment transition leaving v_{T_E} and eventually leading to v_{T_1} and $t'_2 \in \mathscr{T}^\sigma$ be the one eventually leading to v_{T_2}. They must be environment transitions because $\texttt{SYS}_2$ and `LOOPS` cannot branch. *Case t_1 is of `type-1`, t_2 of `type-2`:* There is a *contradiction* because $p \in pre^\sigma(t_1) \cap pre^\sigma(t_2)$ cannot be of `type-1` and of `type-2`. *Case t_1 and t_2 are of `type-2`:* Both transitions can only move tokens from the same generation, lets say $i \in \{1, \ldots, \texttt{max}_S\}$, because otherwise $p \in pre^\sigma(t_1) \cap pre^\sigma(t_2)$ must belong to two different generations which is not possible. Thus, $i \in \texttt{gen}(v_{T_1}) \cap \texttt{gen}(v_{T_2})$.

But in their common predecessor v_{T_E} correspond to a player 1 state, i.e., $l(v_{T_E}) \in V_1$, and the construction in Definition 59 chooses exactly one successor branch for carrying the generation i. *Contradiction.* *Case t_1 and t_2 are of* `type-1`: Since $l(v_{T_E}) \in V_1$, transition t'_1 and t'_2 move an environment token $p_E \in \mathscr{P}_E^\sigma \cap pre^\sigma(t'_1) \cap pre^\sigma(t'_2)$. All `type-1` transitions in the branch starting with t'_1 causally depend on the transition t'_1, and all `type-1` transitions in the branch starting with t'_2 causally depend on t'_2. Especially, transition t_1 is dependent on t'_1 and t_2 is dependent on t'_2. Since $v_{T_1} \neq v_{T_2}$, we have $t_1 \neq t'_1$ and $t_2 \neq t'_2$. Hence, $t'_1 < t_1$ and $t'_2 < t_2$ and therewith $(t_1 \sharp t_2)$. But two conflicted transitions can never be enabled in the same cut when they are not the source of the conflict (which they are not due to the $<$). *Contradiction.*

Deadlock-Avoiding: For showing that σ is *deadlock-avoiding* we have to show that for every reachable marking, if there is a transition enabled in the unfolding, there must also be a transition enabled in the corresponding marking of $\mathscr{N}^\sigma$: $\forall C \in \mathscr{R}(\mathscr{N}^\sigma) : (\exists t^U \in \mathscr{T}^U : pre^U(t^U) \subseteq C) \implies (\exists t^\sigma \in \mathscr{T}^\sigma : pre^\sigma(t^\sigma) \subseteq C)$. Definition 59 constructs $\mathscr{N}^\sigma$ inductively by traversing T_σ in breadth-first order and associates each node with the corresponding cut. Thus, for each cut $C \in \mathscr{R}(\mathscr{N}^\sigma)$ there is either a node $v_T \in T$ with $c(v_T) = C$ or there must be a transition $t^\star \in \mathscr{T}^\sigma$ that takes a token from a place $p \in C$ before all places of C are created in the scheduling induced by $\sigma_{\texttt{inf}}$. In the latter case $pre^\sigma(t^\star) \subseteq C$, because all places of C are still created by the construction and C is a cut. This means their creation cannot be causally dependent on $t^\star$. In the two-player game $G(\mathscr{A})$ each state has a successor and so has the strategy tree T_σ. Let $(v_T, t, v'_T) \in E_T$ be the existing edge in T_σ and $e = ((D, \mathscr{r}, \mathscr{t}), t, (D', \mathscr{r}', \mathscr{t}')) \in E$ the corresponding edge in $G(\mathscr{A})$.

Case $t = \tau$: So, $e \in$ `LOOPS` or $e \in \texttt{LOOPS}_{\lightning}$. If $e \in$ `LOOPS`, then $D \in$ `TERM`, which means that there is also no transition enabled in the unfolding. If $e \in \texttt{LOOPS}_{\lightning}$, then $l(v_T) \in \mathbf{B} \cup \{v_{\lightning}\}$. This means that this play ends up in an infinite loop of nodes v''_T, with $l(v''_T) = v_{\lightning}$. This contradicts that $\sigma_{\texttt{inf}}$ is winning.

Case $t = \top$: So, $e \in \texttt{SYS}_\top$ and $c(v_T) = c(v'_T)$. There cannot be infinitely many consecutive successor states labeled with $\top$ because the successors for edges in $\texttt{SYS}_\top$ cannot contain any $\top$. Thus, we consider v'_T with cut C in another case.

Case $t \in \mathscr{T}$: If $D[t\rangle_2 \Rightarrow i \in \texttt{gen}(v_T)$ holds for the corresponding $i \in \{1, \ldots, \texttt{max}_S\}$ with $(p, i, \cdot) \in D$ for all $p \in pre^{\mathscr{A}}(t)$, then Definition 59 creates a transition $t^\sigma \in \mathscr{T}^\sigma$ which preset is contained in the current cut C. Hence, t^σ is enabled in C. Otherwise, t is a transition only moving tokens of `type-2` typed places of generation $i \neq 0$, but $i \notin \texttt{gen}(v_T)$. Since once a generation $i \neq 0$ is chosen (only possible via an edge in $\texttt{SYS}_\top$), it can never be changed again. Thus, there must be a node $v''_T \in T$ in the past of v_T that is a successor of an edge in $\texttt{SYS}_\top$ and $i \in \texttt{gen}(v''_T)$. Since Definition 59 only does not copy the generation to all successors in cases the node corresponds to a V_1 state, there must be a node $v^0_T \in T$ in between v''_T and v_T with $l(v^0_T) \in V_1$ and where the next node on the way to v_T does not contain i but $i \in \texttt{gen}(v^0_T)$. This node must have another child $v^c_T \in T$ which is not in the past of v_T but $i \in \texttt{gen}(v^c_T)$ due to Definition 59 chooses exactly one successor for the generation in V_1 states. Since t with $D[t\rangle_2$ is taken by an edge in E_T and $\sigma_{\texttt{inf}}$ is winning, it proved to be correctly typed. This means that all tokens residing in those places of C that correspond to $pre^{\mathscr{A}}(t)$ will indeed never synchronize directly or

indirectly with the environment again (only with tokens from generation i). Hence, they are independent of the firing of any environment transition in a V_1 state. So, regardless of how many branching points follow v_T^c, there is one branch where the nodes contain the generation i and the tokens corresponding to generation i are independent of the branching. Thus, either t itself must still be enabled in this branch or at least another transition t' moving tokens of generation i from the places in C must be enabled. Any of these transitions must eventually be taken by an edge in E_T, because $\sigma_{\texttt{inf}}$ is winning and thus, has no deadlock states, and the round robin procedure consecutively schedules the transitions of the ordered set $\langle \mathscr{T} \rangle$. Hence, eventually a transition $t^\sigma \in \mathscr{T}^\sigma$ is added to $\mathscr{N}^\sigma$ that moves the tokens of generation i of cut C. So, $pre^\sigma(t^\sigma) \subseteq C$. □

We now show that also the other direction holds. This means we prove that the function $\sigma_{\texttt{inf}}$ constructed in Definition 61 from a deadlock-avoiding strategy σ of the system players in a Petri game with transits actually is a winning strategy of player 0 in the corresponding information flow game.

Proof (Theorem 8: From Petri Game to Information Flow Game): The function $\sigma_{\texttt{inf}}$ is a *strategy* because it defines successors for V_0 states and in all cases the definition either fits to the successor definition of the edges $\texttt{SYS}_\top$, $\texttt{SYS}$, or $\texttt{LOOPS}_{\lightning}$, or an arbitrary successor is chosen. For a play of $G(\mathscr{A})$, in each case such an edge must always exist for V_0 states.

We show that $\sigma_{\texttt{inf}}$ is *winning*. Let $\pi_{\texttt{inf}}$ be a play of $G(\mathscr{A})$ conforming to $\sigma_{\texttt{inf}}$. The sequence $fs(\pi_{\texttt{inf}})$ indeed corresponds to an initial firing sequence in $\mathscr{A}$ because all edges either keep the marking or the successor marking corresponds to the firing of a transition. Thus, each state of the play corresponds to a cut of an unfolding of $\mathscr{A}$. Even more, due to Definition 61 choosing the successors according to the firing of a transition $t^\sigma \in \mathscr{T}^\sigma$, each state corresponds to a cut $\mathscr{C}$ of $\mathscr{N}^\sigma$. Since σ is deterministic and all plays conforming to $\sigma_{\texttt{inf}}$ choose every possible commitment set according to the postsets of the places in the corresponding cut $\mathscr{C}_i$, no state can have a decision set $D \in \texttt{NDET}$.

We show that for each $v_i \in V_0 \setminus \mathbf{B}$ of $\pi_{\texttt{inf}}$ with successor $v_{i+1} = \sigma_{\texttt{inf}}(v_i) = (D', \mathscr{r}', \mathscr{t}')$ the decision set $D' \notin \texttt{DL}$ by contradiction. *Assume* $D' \in \texttt{DL}$. Thus, it exists a transition $t \in \mathscr{T}$ such that $\mathscr{M}(D')[t\rangle$ but for all $t' \in \mathscr{T}$ we have $\neg D'[t'\rangle$. Hence, $D' \notin \texttt{TERM}$. Since σ is deadlock-avoiding, there must be a transition $t^\sigma \in \mathscr{T}^\sigma$ with $\mathscr{C}_{i+1}[t^\sigma\rangle$ for the corresponding cut $\mathscr{C}_{i+1}$. Hence, $D'(\lambda^\sigma(t^\sigma)\rangle$, because in both cases of Definition 61 the commitment sets of the successor v_{i+1} are chosen corresponding to all postset transitions of the places in $\mathscr{C}_{i+1}$. Thus, the only possibility for $\neg D'[\lambda^\sigma(t^\sigma)\rangle$ is that there is some place $p \in pre^{\mathscr{A}}(\lambda^\sigma(t^\sigma))$ having another generation than the others in D'. Since generations are only chosen by $\texttt{SYS}_\top$ edges in a V_0 state and can never be changed back when once chosen to be unequal to zero, the different generations have to be chosen by the first case in Definition 61 for some predecessor node of v_{i+1}. However, whenever a $\top$-symbol is resolved, all places in the successor decision set can choose a new generation in case they had been of generation zero before. For the constructed strategy $\sigma_{\texttt{inf}}$ this choice is uniformly done according to $\mathscr{N}^\sigma$. Thus, the places in $pre^{\mathscr{A}}(\lambda^\sigma(t^\sigma))$ cannot belong to different generations. *Contradiction.* Since $I \in V_0 \setminus \mathbf{B}$ due to the $\top$-symbol, and all successors of V_1 states that are reached while moving system tokens have a $\top$-symbol,

thus are in $V_0 \setminus \mathbf{B}$, we know that no state with a decision set in $\texttt{DL}$ can occur in $\pi_{\texttt{inf}}$. Thus, $\pi_{\texttt{inf}}$ avoids the non-accepting sink $v_{\lightning} \in V_0$.

Assume $\pi_{\texttt{inf}}$ still does not visit infinitely many V_1 states. In terminating situations we loop infinitely long with $\texttt{LOOPS}$ in V_1 states and the edges $\texttt{ENV}$ and $\texttt{SYS}_2$ start from V_1 states. Moreover, since the edges in $\texttt{SYS}_\top$ resolve all $\top$-symbols and the edges in $\texttt{SYS}$ do not introduce any $\top$-symbol, there must be infinitely many consecutive $\texttt{SYS}$ edges to not infinitely often visit states of V_1. This means from some state v_i in $\pi_{\texttt{inf}}$ on, infinitely many pure system transitions are fired and all places in the preset of the transitions must belong to generation zero. Thus, for all these places there must be a state $v_j \in V_0 \setminus \mathbf{B}$ with $j < i$, where the corresponding $\top$-symbol is resolved and generation zero is assigned to the current place of the corresponding token. Due to Definition 61, this means that $\texttt{type}(p^\sigma) \neq 2$ for the place p^σ of the corresponding cut $\mathscr{C}_j$. However, since all $\texttt{SYS}$ edges choose the commitment sets according to $\mathscr{N}^\sigma$ and no state V_1 is visited again, meaning no environment token is moved, $\texttt{type}(p^\sigma) = 2$. *Contradiction.* □

For the detailed proofs of the transit automata section (Sec. 12.2), we start by showing that the existential transit automata behave as expected. The acceptance is independent of the concrete choice of the covering firing sequence and the winning property of the plays correspond to the acceptance of the automata.

Proof (Lemma 22: Expediency of the ETA): Let π be a play of a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$ with an *existential* local winning condition $\texttt{WIN} = \exists\text{-}W(\mathscr{W})$ with $W \in \{\texttt{SAFE}, \texttt{REACH}, \texttt{BUCHI}, \texttt{COBUCHI}, \texttt{PARITY}\}$.

Property *(i)*: Let $\zeta \in Z(\pi)$ be a covering firing sequence of π with $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_\exists^{\texttt{WIN}}(\mathscr{G}))$ for the transit automaton $\Lambda_\exists^{\texttt{WIN}}(\mathscr{G})$ corresponding to $\mathscr{G}$ and $\zeta' \in Z(\pi)$ another covering firing sequence of π. Since π is a run, all nondeterminism is fixed, and ζ and ζ' only differ in the ordering of concurrent transitions. Especially, either both are finite or both are infinite. Hence, $\sigma_{\mathscr{T}}(\zeta)$ is padding ζ with τ^ω iff $\sigma_{\mathscr{T}}(\zeta')$ does this for ζ'. Let $\sigma_{\mathscr{T}}(\zeta) = t_0 t_1 \cdots$. For $\texttt{WIN} \in \{\exists\text{-}\texttt{SAFE}(\mathcal{S}), \exists\text{-}\texttt{COBUCHI}(\mathcal{S})\}$, the initial state s_0 is added to the co-Büchi states of the automaton. Furthermore, for $\texttt{WIN} = \exists\text{-}\texttt{PARITY}(\Omega)$, the initial state s_0 of the automaton gets an odd priority and for $\texttt{WIN} \in \{\exists\text{-}\texttt{REACH}(\mathcal{S}), \exists\text{-}\texttt{BUCHI}(\mathcal{S})\}$, the initial state cannot be part of $\mathcal{S}$. Hence, no run can be accepted that never leaves s_0. Since $\sigma_{\mathscr{T}}(\zeta)$ is accepted, there must be an index $i \in \mathbb{N}$ and a state $s \in S$ such that $(\triangleright, s) \in \Upsilon(t_i)$. Let $t_i^R \in \mathscr{T}^R$ be the corresponding transition in the play π. Since π is a run and ζ and ζ' are covering firing sequences, there must also be an index $j \in \mathbb{N}$ at which $t_i^R \in \mathscr{T}^R$ occur in ζ'. Let $t_j \in \mathscr{T}$ be the corresponding transition in $\sigma_{\mathscr{T}}(\zeta')$. Thus, also for the input word $\sigma_{\mathscr{T}}(\zeta')$ the automaton $\Lambda_\exists^{\texttt{WIN}}(\mathscr{G})$ has a run leaving the initial state s_0 and move to state s. For all transitions $t^R \in \mathscr{T}^R$ that causally depend on t_i^R, i.e., $t^R > t_i^R$, we know that the corresponding transition $t \in \mathscr{T}$ has to occur in $\sigma_{\mathscr{T}}(\zeta)$ with an index greater i and with an index greater j in $\sigma_{\mathscr{T}}(\zeta')$ because they are both traces of covering firing sequences of the same run. Each transition extending the started chain must be causally dependent on t_i^R, because $s \in \mathit{pre}(t)$ must hold for any $(s, s') \in \Upsilon(t)$. Either no transition ever extends the chain and the accepting run for $\sigma_{\mathscr{T}}(\zeta)$ stays in s by using the *total* and possibly the τ-edges from *stutter* or the state s is left using a *transit* edge due to a

transition extending the chain. In the first case, the automaton can also never leave s for the run $\sigma_{\mathscr{T}}(\zeta')$, because a state can only be left by using a transition extending the chain and thus, would be causally dependent on t_i^R and therewith would have to occur in $\sigma_{\mathscr{T}}(\zeta)$ after t_i. In the second case, the automaton can eventually also leave s with the same edge from *transit* for $\sigma_{\mathscr{T}}(\zeta')$ reaching the same successor because of the causally dependency. This argument can be repeated for all further transitions. Thus, the state changes are exactly the same and the infinite looping is done in the same states for both words $\sigma_{\mathscr{T}}(\zeta)$ and $\sigma_{\mathscr{T}}(\zeta')$. The corresponding runs only differ in the finite looping in states for concurrent transitions. Since the acceptance conditions are not dependent on finite looping or concrete time points for the state changes, $\sigma_{\mathscr{T}}(\zeta') \in \mathscr{L}(\Lambda_{\exists}^{\texttt{WIN}}(\mathscr{G}))$ because $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\texttt{WIN}}(\mathscr{G}))$.

Property *(ii)*:
Completeness (direction "⇒"): Since $\pi \in \texttt{WIN}$, there is a flow chain $\xi \in \Xi(\pi)$ satisfying the corresponding condition. Thus, there must be a transition $t \in \mathscr{T}$ and a place $p \in \mathscr{P}$ with $(\triangleright, p) \in \Upsilon(t)$ such that the corresponding elements $t^R \in \mathscr{T}^R$ and $p^R \in \mathscr{P}^R$ of π start the chain. Let $\zeta = t_0^R t_1^R \cdots \in Z(\pi)$ be a covering firing sequence and denote the elements of the trace with $\sigma_{\mathscr{T}}(\zeta) = t_0 t_1 \cdots$. Since ζ is covering π, all transitions occurring in ξ must appear. Due to the definition of a flow chain, all transitions of ξ must be causally related. Hence, they must also appear in the same order in ζ, only possibly padded with concurrent transitions. Thus, we can consider the run for the input word $\sigma_{\mathscr{T}}(\zeta)$ that loops in s_0 until the transition starting the chain (i.e., the transition corresponding to t^R) is reached and then changes into the successor state p. For every other state, the run loops with *total* for every t_i that does not belong to ξ and for each transition t_j^R of ξ, the run performs the corresponding state change implied by the successor place in ξ. For finite chains the run of the automaton loops with *stutter* in a state corresponding to the last place of the chain due to the τ^ω suffix. For all but the $\exists$-$\texttt{SAFE}(\mathscr{S})$ condition, the type of the corresponding acceptance condition of the automaton is the same as the type of the winning condition and also the set $\mathscr{S}$ or the parity function maximally differ by additionally covering the initial state s_0. Since the run of the automaton stays only finitely long in s_0, this difference has no impact on the acceptance. Apart from the initial state, the run of the automaton visits exactly the states corresponding to the places of ξ and, in between, may only loop finitely long in all but the last state. These finite loops does not have any impact on the acceptance and thus, since ξ satisfies the winning property, $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\texttt{WIN}}(\mathscr{G}))$ for these conditions. For the $\exists$-$\texttt{SAFE}(\mathscr{S})$ condition, the situation is quite similar. We only change to a co-Büchi acceptance condition for the automaton. But since we adapted the transition relation of the automaton in the way that we can only loop in all states $p \in \mathscr{S}$, the satisfaction of the winning property for ξ and the acceptance of the run again coincide and thus, also $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\texttt{WIN}}(\mathscr{G}))$ holds.
Soundness (direction "⇐"): Let $\zeta \in Z(\pi)$ be a covering firing sequence with $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\exists}^{\texttt{WIN}}(\mathscr{G}))$. Since the initial state s_0 is neither in the reachable, nor in the Büchi states of the acceptance conditions of the automaton, it is especially added to the co-Büchi states, and has an odd color for the parity condition, it has to be left eventually for $\sigma_{\mathscr{T}}(\zeta)$ to be accepted. Thus, ζ must eventually start a flow chain. For each state, the run of the

automaton accepting $\sigma_{\mathscr{T}}(\zeta)$ either loops finitely and afterwards uses an edge of *transit* to proceed to a successor state, or eventually loops infinitely long in its last state. Each state change corresponds to a transition extending the flow chain. Since *total* is only adding edges to states where no edge with a label for the same transition is available, the run cannot skip any transition extending the current chain. Thus, collecting the places corresponding to the states actually yields a flow chain of π. Since reaching a bad place for the $\exists$-$\texttt{SAFE}(\mathcal{S})$ condition enforces that the run can only loop in these places, the co-Büchi acceptance condition also allows for the acceptance of chains that only reach a bad state once. Hence, for all conditions the corresponding winning property for the created chain is satisfied and so, $\pi \in \texttt{WIN}$. □

For the universal transit automata, we show that they are dual to the existential transit automata, so that the expediency of the UTA amounts to the expediency of the ETA.

Proof (Lemma 23: Dualization): By dualizing the transition relation of $\Lambda^{\texttt{WIN}}_{\forall}(\mathscr{G})$, we obtain $\widetilde{start} = \{(s_0, t, s_0) \in S \times \mathscr{T} \times S\} \cup \{(s_0, t, s) \in S \times \mathscr{T} \times S \mid (\triangleright, s) \in \Upsilon(t)\}$ because the conjunction over an empty set yields *true* and $\widetilde{transit} = \{(s, t, s') \in S \times \mathscr{T} \times S \mid (s, s') \in \Upsilon(t)\}$ because the sets are disjoint. Furthermore, $\widetilde{total} = total$ and $\widetilde{stutter} = stutter$ because the *total* and *stutter* sets are disjoint to all other sets and are deterministic. These are exactly the sets for the transition relation of $\Lambda^{\widetilde{\texttt{WIN}}}_{\exists}(\mathscr{G})$, however the loop for each transition $t \in \mathscr{T}$ is not in the *start* set but in the *total* set. Since $\widetilde{\forall\text{-}\texttt{REACH}}(\mathcal{S}) = \exists\text{-}\texttt{SAFE}(\mathcal{S})$, also the additional case for the edge relation $\rightarrow$ fits. For the acceptance condition we can also simply calculate the result. For example, for the winning condition $\forall\text{-}\texttt{BUCHI}(\mathcal{S})$ the acceptance condition is $\texttt{BUCHI}(\{s_0\} \cup \mathcal{S})$. Thus, the dual is $\texttt{COBUCHI}(\{s_0\} \cup \mathcal{S})$, which is exactly the acceptance condition for the ETA for the winning condition $\widetilde{\forall\text{-}\texttt{BUCHI}}(\mathcal{S}) = \exists\text{-}\texttt{COBUCHI}(\mathcal{S})$. The second equation easily follows from this because the dualization function is self-inverse: $\Lambda^{\widetilde{\texttt{WIN}}}_{\forall}(\mathscr{G}) = \Lambda^{\widetilde{\widetilde{\texttt{WIN}}}}_{\forall}(\mathscr{G}) = \Lambda^{\widetilde{\widetilde{\texttt{WIN}}}}_{\exists}(\mathscr{G}) = \Lambda^{\widetilde{\texttt{WIN}}}_{\exists}(\mathscr{G})$.□

For showing the expediency of the FTA, we use the ideas for showing the expediency of the ETA and show that the automata checking the single local Flow-LTL subformulas are triggered only for the places of the desired chain.

Proof (Lemma 25: Expediency of the FTA): Let π be a play of a Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{Flow-LTL}(\varphi))$ with a local Flow-LTL formula φ with only places as atomic propositions.

Property *(i)*: Let $\zeta \in Z(\pi)$ be a covering firing sequence of π with $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\text{F-}LTL}(\mathscr{G}))$ and $\zeta' \in Z(\pi)$ another covering firing sequence of π. Since the acceptance of $\Lambda^{\text{F-}LTL}(\mathscr{G})$ boils down to the acceptance of the single Flow-LTL transit automata $\Lambda^{\mathbb{E}LTL}_{\psi_i}(\mathscr{G})$, we show the corresponding property for these automata. Since each accepting run must start a flow chain, there must be a flow chain ξ due to the acceptance of $\sigma_{\mathscr{T}}(\zeta)$. Again, the two covering firing sequences ζ and ζ' only differ in the ordering of transitions not extending the chain and the ordering of the transitions extending the chain are preserved. The acceptance of $\Lambda^{\mathbb{E}LTL}_{\psi_i}(\mathscr{G})$ depends on the acceptance of A_{ψ_i}. Since A_{ψ_i} is

only triggered by places corresponding to the firing of transitions extending chain, the acceptance of $\sigma_{\mathscr{T}}(\zeta')$ can be shown with the analogous arguments as for the ETA. Thus, the accepting run for $\sigma_{\mathscr{T}}(\zeta')$ loops in (s_0, q_0) without triggering A_{ψ_i} as long as the transition starting the chain is reached. With the second set of the edges in *init*, the first place of the chain is tracked and A_{ψ_i} is triggered accordingly. Only the edges in *transit* and *potInfy* can be used for transitions extending the chain correctly, and those edges are also triggering A_{ψ_i} correspondingly. Since ζ and ζ' are both covering firing sequences, and these transitions are causally related, we trigger A_{ψ_i} in the same way. All transitions in between does not trigger A_{ψ_i} for both firing sequences. When for $\sigma_{\mathscr{T}}(\zeta)$ the automaton $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ switches into the $\mathfrak{s}$-mode (using only states of $S_{\mathfrak{s}}$ for the first component), the chain must be finite because every transition extending the chain would lead to the non-acceptance of $\sigma_{\mathscr{T}}(\zeta)$. In this case we can also eventually switch for $\sigma_{\mathscr{T}}(\zeta')$ in this mode when the last place of the chain is reached. Here in both cases A_{ψ_i} is triggered with the last place of the chain. Hence, for $\sigma_{\mathscr{T}}(\zeta')$ the automaton A_{ψ_i}, that is responsible for the acceptance, is triggered in the same way as for $\sigma_{\mathscr{T}}(\zeta)$, and so, $\sigma_{\mathscr{T}}(\zeta') \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$, when $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$. The intersection, union, or complementation of the automata is not dependent on the firing sequence. Thus, $\sigma_{\mathscr{T}}(\zeta') \in \mathscr{L}(\Lambda^{\text{F-}LTL}(\mathscr{G}))$.

Property *(ii)*: We first show that $\pi \in \texttt{Flow-LTL}(\mathbb{E}\,\psi_i) \iff \forall \zeta \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$, before lifting this result to the complete local Flow-LTL logics.
Completeness (direction "⇒"): Let $\pi \in \texttt{Flow-LTL}(\mathbb{E}\,\psi_i)$ and $\zeta \in Z(\pi)$. There must be a flow chain ξ of π with $\widetilde{\sigma}(\xi) \models \psi_i$. Since ζ is covering π, the transition creating and all transitions extending the chain must occur. Thus, as in the case of the ETA, there is the run of $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ that starts the chain and also triggers A_{ψ_i} with the first place of the chain. In case ξ is finite because ζ is finite, we can stutter with the τ-edges of *stutter* in the last place of the chain and trigger A_{ψ_i} accordingly. Note that we use the $\mathfrak{s}$-mode for the constructed run of $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ in case the last transition in ζ is not the transition extending the chain to its finite place and not the c-mode because there the stuttering would not lead to an accepting run. If ξ is finite due to infinitely many transitions not extending the chain are firing, we choose with the edges in *guessFin* to switch into the $\mathfrak{s}$-mode for the first of these transitions and do not trigger A_{ψ_i}. We stay in this mode and use the edges in *fin* to stutter on the last place of the chain while triggering A_{ψ_i} accordingly. For all transitions in ζ occurring before this case happens and in case of an infinite chain ξ, we can either use the edges in *transit* to track the next place of ξ and trigger A_{ψ_i} accordingly, or first switch into the c-mode with the edges in *guessFin* without triggering A_{ψ_i} for a transition not extending the chain. In the latter case, we stay in the c-mode with the first set of the edges in *potInfy* without triggering A_{ψ_i} for transitions in ζ not extending ξ until a transition extending ξ occur that brings the automaton back into the normal mode, tracks the next place of ξ, and triggers A_{ψ_i} accordingly. Hence, for finite or infinite chains ξ, we track the places of the chain and trigger A_{ψ_i} according to the places of ξ. Thus, $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$.
Soundness (direction "⇐"): Let $\zeta \in Z(\pi)$ with $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$. Since no run can be accepted by $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ that loops infinitely long in states with s_0 in the first component, there must be a start of a flow chain ξ and this start also triggers A_{ψ_i}. The

successor state is in $S \times Q$. When the accepting run uses edges in *transit*, we can extend ξ and also this edge triggers A_{ψ_i} and results in a state in $S \times Q$. When a *guessFin* edge is used, A_{ψ_i} is not triggered and we switch either in the mode $\mathfrak{s}$ or c. In mode c, when the first set of edges in *potInfy* are used, we do not extend ξ and also A_{ψ_i} is not triggered. In this case we stay in mode c. Since the corresponding run of $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ is accepted, it cannot use these edges infinitely often in a row because the S_c states are explicitly withdrawn from the first component of the accepting Büchi states. Thus, the run eventually leaves the c-mode into an $S \times Q$ state with an edge in *potInfy*. The used edge triggers A_{ψ_i} and corresponds to a transition extending the chain. Hence, in this case we also extend ξ. In mode $\mathfrak{s}$, the run can infinitely long use the edges in *fin*. In this case we do not extend ξ. This stuttering is done by considering $\widetilde{\sigma}(\xi)$ for the finite chain and perfectly fits the triggering of A_{ψ_i} with the last place of the chain by the edges in *fin*. An edge from *error* cannot be taken because then the run could only loop infinitely long in state $s_\lightning$, which is not accepting. Thus, the mode has been correctly guessed and the chain is never extended again. When ζ is finite, the τ-edges in *stutter* are used by the accepting run. These also trigger A_{ψ_i} with the last place of the chain. Since the triggering of A_{ψ_i} is exactly done for the places of the flow chain ξ, and A_{ψ_i} accepts all words satisfying ψ_i, $\pi \in \texttt{Flow-LTL}(\mathbb{E}\,\psi_i)$ holds.

We now compose these results to show Property *(ii)* for φ. W.l.o.g., we consider φ in the form after applying Step 1 of Sec. 12.2.3. For the *completeness* let $\pi \in \texttt{Flow-LTL}(\varphi)$ and $\zeta \in Z(\pi)$. For all subformulas $\mathbb{E}\,\psi_i$ of φ not preceded by negation, the completeness result above yields $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$, when $\pi \in \texttt{Flow-LTL}(\mathbb{E}\,\psi_i)$ is satisfied. For all subformulas $\neg\mathbb{E}\,\psi_j$ of φ, we use the contraposition of the soundness result above, when $\neg\mathbb{E}\,\psi_j$ is satisfied by π. Hence, $\exists \zeta' \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta') \notin \mathscr{L}(\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G}))$. The contraposition of Property *(i)* yields $\forall \zeta' \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta') \notin \mathscr{L}(\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G}))$. Thus, $\sigma_{\mathscr{T}}(\zeta) \notin \mathscr{L}(\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G}))$, and hence, $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\overline{\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G})})$. Since the composition of the automata $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ and $\overline{\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G})}$ exactly fit the formula composition, $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\text{F-}LTL}(\mathscr{G}))$.

For the *soundness* let $\zeta \in Z(\pi)$ be a covering firing sequence with $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda^{\text{F-}LTL}(\mathscr{G}))$. The construction of $\Lambda^{\text{F-}LTL}(\mathscr{G})$ builds the intersections and unions of the $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ and $\overline{\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G})}$ automata corresponding the conjunctions and disjunctions of the subformulas $\mathbb{E}\,\psi_i$ and $\neg\mathbb{E}\,\psi_j$. Consider only those automata that are responsible for the acceptance of $\sigma_{\mathscr{T}}(\zeta)$. Thus, $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$ for subformula $\mathbb{E}\,\psi_i$ and $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\overline{\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G})})$ for subformula $\neg\mathbb{E}\,\psi_j$. Property *(i)* for the SFTA yields $\forall \zeta' \in Z(\pi) : \sigma_{\mathscr{T}}(\zeta') \in \mathscr{L}(\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G}))$ and so the soundness result above yields $\pi \in \texttt{Flow-LTL}(\mathbb{E}\,\psi_i)$. The contraposition of the completeness result above yields $\pi \notin \texttt{Flow-LTL}(\mathbb{E}\,\psi_j)$ because $\sigma_{\mathscr{T}}(\zeta) \notin \mathscr{L}(\Lambda_{\psi_j}^{\mathbb{E}LTL}(\mathscr{G}))$. Thus, $\pi \in \texttt{Flow-LTL}(\neg\mathbb{E}\,\psi_j)$. Since the conjunctions and disjunctions fit the intersections and unions in the construction of $\Lambda^{\text{F-}LTL}(\mathscr{G})$, we have $\pi \in \texttt{Flow-LTL}(\varphi)$. □

Combining the results for the information flow game and the results for the transit automata yields the correctness of the construction of the product game and therewith the final decision procedure for Petri games with transits and local winning objectives presented in Sec. 12.3.

Proof (Lemma 26: Reduction): Given a Petri game with transits $\mathcal{G} = (\mathcal{A}, \texttt{WIN})$ with an arena $\mathcal{A}$ and a local winning condition $\texttt{WIN}$. We construct the product game $\mathbb{G}(\mathcal{G}) = G(\mathcal{A}) \mid\mid \Lambda(\mathcal{G})$ of the information flow game $G(\mathcal{A}) = (A(\mathcal{A}), \texttt{BUCHI}(V_1))$ with arena $A(\mathcal{A}) = (V, V_0, V_1, E, I)$ and the deterministic transit automaton $\Lambda(\mathcal{G}) = (\mathcal{T} \cup \{\top, \tau\}, S, s_0, \rightarrow, \texttt{PARITY}(\Omega))$ according to Definition 65.

Case completeness ("if"): Let σ be a deadlock-avoiding winning strategy of $\mathcal{G}$. Theorem 8 yields that $\sigma_{\texttt{inf}} = \sigma_{\texttt{pg2inf}}(\sigma) : V^*V_0 \to V$ is a winning strategy for player 0 of the information flow game $G(\mathcal{A})$. We define $\sigma_{||} : (V \times S)^*(V_0 \times S) \to (V \times S)$ by using $\sigma_{\texttt{inf}}$ to define the successor $v \in V$ of the first component and by adding the successor of the second component according to the deterministic decision of $\Lambda(\mathcal{G})$ with respect to the label used in $G(\mathcal{A})$ to reach the successor v. Thus, given a word $w v \in (V \times S)^*(V_0 \times S)$ with $w = (v^0, s^0) \cdots (v^n, s^n)$ and $v = (v, s)$, we consider the state $v' = \sigma_{\texttt{inf}}(v^0 \cdots v^n v) = (D, \nu, t)$. In case $v^0 \cdots v^n v$ is no play of $G(\mathcal{A})$, we can map wv to an arbitrary successor because we are only interested in plays of $\mathbb{G}(\mathcal{G})$ and, in this case, wv cannot be a play of $\mathbb{G}(\mathcal{G})$. Since $\Lambda(\mathcal{G})$ has a successor for every $t \in \mathcal{T} \cup \{\top, \tau\}$, there is an edge (s, t, s') and we can define $\sigma_{||}(wv) = (v', s')$. Note that as long wv is a play of $\mathbb{G}(\mathcal{G})$, the word $wv(v', s')$ is a play of $\mathbb{G}(\mathcal{G})$.

We now show that $\sigma_{||}$ is winning for player 0 in $\mathbb{G}(\mathcal{G})$. Given a play $w \in (V \times S)^\omega$ conforming to $\sigma_{||}$. Due to the construction of $\sigma_{||}$, we know that the projection on the first component is a play $\pi_{\texttt{inf}}$ of $G(\mathcal{A})$ that conforms to $\sigma_{\texttt{inf}}$ and thus, is winning for player 0 in $G(\mathcal{A})$. This means there are infinitely many V_1 states in the first component of w. Hence, w satisfies the first conjunct of the winning condition $\texttt{CONJ}(\Omega_1, \Omega_2)$, i.e., $w \in \texttt{PARITY}(\Omega_1)$. Furthermore, due to Definition 61, we know that $fs(\pi_{\texttt{inf}})$ belongs to a covering firing sequence of a maximal play π that conforms to σ because $\pi_{\texttt{inf}}$ is a play of $G(\mathcal{A})$ that conforms to $\sigma_{\texttt{inf}}$. Since σ is winning, $\pi \in \texttt{WIN}$ holds. Depending on the winning condition, either Lemma 22, Lemma 24, or Lemma 25, yields that all traces of covering firing sequences $\zeta \in Z(\pi)$ are accepted by the corresponding automaton $\Lambda(\mathcal{G})$, i.e., $\sigma_{\mathcal{T}}(\zeta) \in \mathcal{L}(\Lambda(\mathcal{G}))$. Note that $\sigma_{\mathcal{T}}(\zeta)$ is padded with τ^ω when $fs(\pi_{\texttt{inf}})$ is a finite word. In this case, w must also have a loop with infinitely many τ-edges at the end. These edges at the end of the word cannot result from edges in $\texttt{LOOPS}_{\lightning}$ of $G(\mathcal{A})$ because $\pi_{\texttt{inf}}$ is winning and $v_{\lightning} \notin V_1$. Thus, the possible τ-edges can only stem from the loops in terminating states and that fits the padding of $\sigma_{\mathcal{T}}(\zeta)$. The sequence of edge labels of w may only differ from $\sigma_{\mathcal{T}}(\zeta)$ by finitely many $\top$-labels. The acceptance of the automaton $\Lambda(\mathcal{G})$ is independent of finite $\top$-edges due to the deterministic $\top$-loop at every state (for the FTA, the automaton checking the LTL formula is not triggered for $\top$-edges). Since $fs(\pi_{\texttt{inf}})$ corresponds to such a firing sequence and the sequence of edge labels of w may only differ in finitely many $\top$-labels and the τ-looping, the projection of the second component satisfies the second conjunct of the winning condition $\texttt{CONJ}(\Omega_1, \Omega_2)$ because this corresponds to the acceptance of triggering the deterministic automaton $\Lambda(\mathcal{G})$. Hence, $w \in \texttt{PARITY}(\Omega_2)$ and thus, $w \in \texttt{CONJ}(\Omega_1, \Omega_2)$.

Case soundness ("only if"): Let $\sigma_{||} : (V \times S)^*(V_0 \times S) \to (V \times S)$ be a winning strategy of player 0 in the product game $\mathbb{G}(\mathcal{G})$. We create a function $\sigma_{\texttt{inf}} : V^*V_0 \to V$ by using the mapping $\sigma_{||}$ for the first component, when composing the deterministic decision of

the automaton to a given play conforming to $\sigma_{\texttt{inf}}$. Let $w \in V^*V_0$ with $w = v_0v_1 \cdots v_n$. Since we are only interested in words corresponding to prefixes of plays in $G(\mathscr{A})$, we can map all other words to an arbitrary successor. We know for all plays $\pi_{\texttt{inf}} = v'_0v'_1 \cdots$ of $G(\mathscr{A})$ that $v'_0 = I$ and that for all $i \in \mathbb{N}$ there is a label $l_i \in \mathscr{T} \cup \{\top, \tau\}$ such that $(v'_i, l_i, v'_{i+1}) \in E$. We create the word $\mathscr{w} = (I, s_0)(v_1, s_1) \cdots (v_n, s_n)$ by adding the deterministic decisions of $\Lambda(\mathscr{G})$ to the second component, i.e., $(s_i, l_i, s_{i+1}) \in \rightarrow$ for all $i \in \{0, \ldots, n-1\}$ and $(v_i, l_i, v_{i+1}) \in E$. Note that the successor s_{i+1} is unique because $\Lambda(\mathscr{G})$ is deterministic. Since $\Lambda(\mathscr{G})$ has an edge for each $l \in \mathscr{T} \cup \{\top, \tau\}$ and w is a prefix of a play of $\Lambda(\mathscr{G})$, the word $\mathscr{w}$ is a prefix of a play of $\mathbb{G}(\mathscr{G})$. Hence, we obtain a meaningful successor $\sigma_{||}(\mathscr{w}) = (v, s)$ and define $\sigma_{\texttt{inf}}(w) = v$. Since $\sigma_{||}$ is winning for player 0 in $\mathbb{G}(\mathscr{G})$ and especially the first component of a play conforming to $\sigma_{||}$ visits infinitely many V_1 states, $\sigma_{\texttt{inf}}$ is winning for player 0 in $G(\mathscr{A})$ because we can extend each play $\pi_{\texttt{inf}} \in V^\omega$ conforming to $\sigma_{\texttt{inf}}$ as above by adding the second component with respect to the deterministically triggering of $\Lambda(\mathscr{G})$. This play then conforms to $\sigma_{||}$ due to the construction of $\sigma_{\texttt{inf}}$, hence, visits also infinitely many V_1 states, and thus is winning. Theorem 7 yields that $\sigma = \sigma_{\texttt{inf2pg}}(\sigma_{\texttt{inf}})$ is a deadlock-avoiding strategy for the system players of $\mathscr{G}$.

We now show that σ is winning for the system players of $\mathscr{G}$. Definition 59 creates σ by traversing the strategy tree of $\sigma_{\texttt{inf}}$ in breadth-first order and adding the new transition and successor places to the associated cut of σ. This means that each play π conforming to σ has a covering firing sequence ζ that has the same ordering of the concurrent transitions as $fs(\pi_{\texttt{inf}})$ of a play $\pi_{\texttt{inf}}$ conforming to $\sigma_{\texttt{inf}}$. Extending $\pi_{\texttt{inf}}$ deterministically with the second component as above, we again obtain a play that conforms to $\sigma_{||}$ and thus, is winning. This means the sequence of labels of $\pi_{\texttt{inf}}$ is accepted by $\Lambda(\mathscr{G})$. The sequence $\sigma_{\mathscr{T}}(\zeta)$ only differs from this sequence by finitely many $\top$-labels. Since for a $\top$-label the automaton $\Lambda(\mathscr{G})$ only loops in the current state and the acceptance condition is not sensitive for finite stuttering (for the FTA, the automaton checking the LTL formula is not triggered for $\top$-edges), $\sigma_{\mathscr{T}}(\zeta) \in \mathscr{L}(\Lambda(\mathscr{G}))$. Thus, either Lemma 22, Lemma 24, or Lemma 25 yield that π is winning and hence, σ is winning. □

Finally, we show the complexity of the reduction by looking at the sizes of the information flow game and the transit automata, and by investigating the costs for solving the conjunction of the Büchi winning condition of the information game and the different acceptance conditions of the transit automata.

Proof (Theorem 9): The complexity of solving the Petri game with transits $\mathscr{G} = (\mathscr{A}, \texttt{WIN})$ depends on the size of the product game $\mathbb{G}(\mathscr{G})$ and the complexity needed for solving $\mathbb{G}(\mathscr{G})$. The size of the product game is the multiplication of the size of the information flow game $G(\mathscr{A})$ and the size of the transit automaton. The size of $G(\mathscr{A})$ is single-exponential in $|\mathscr{A}|$ due to Lemma 21.

Place-based conditions: The size of the ETA and the size of the UTA is single-exponential in $|\mathscr{A}|$ due to the determinization step. Thus, the size of $\mathbb{G}(\mathscr{G})$ is *single-exponential* in $|\mathscr{A}|$ for all place-based winning conditions.

We show that the conjunction of the Büchi condition of $G(\mathscr{A})$ and the corresponding acceptance condition of the transit automaton result in a condition that is easier to solve

than the generalized parity condition. For $\texttt{WIN} = \exists\text{-}\texttt{SAFE}(\mathcal{S})$ and $\texttt{WIN} = \exists\text{-}\texttt{COBUCHI}(\mathcal{S})$, we have the conjunction of a Büchi and a co-Büchi condition, i.e., a Rabin condition with one pair. This can be expressed by a parity condition with three colors [CHVB18]. Thus, $\mathbb{G}(\mathcal{G})$ can be solved in *single-exponential time*. For $\texttt{WIN} = \forall\text{-}\texttt{SAFE}(\mathcal{S})$, we have the conjunction of a Büchi and a safety condition. By adapting the game arena such that each state that is not safe only allows to succeed in a non-accepting sink, we can solve $\mathbb{G}(\mathcal{G})$ with a single Büchi condition and thus, in *single-exponential time*. For $\texttt{WIN} = \exists\text{-}\texttt{REACH}(\mathcal{S})$, we have the conjunction of a Büchi and a reachability condition. We again adapt the game arena, to obtain a single Büchi condition. This time we equip each state with an additional Boolean flag for stating whether a state of the desired reachable states has already been reached. Thus, we start with the flag set to *false* and keep it until we reach a state from the desired set. Then, we switch to *true* and can never change the flag back. Restricting the set of Büchi states to those having a *true* reachable flag, allows to solve $\mathbb{G}(\mathcal{G})$ in *single-exponential time*. For $\texttt{WIN} = \forall\text{-}\texttt{REACH}(\mathcal{S})$ and $\texttt{WIN} = \forall\text{-}\texttt{BUCHI}(\mathcal{S})$, we have a generalized Büchi condition that can be solved in polynomial time [BCGH+10; CH14; CDHL16]. Thus, solving $\mathbb{G}(\mathcal{G})$ is *single-exponential*. For $\texttt{WIN} \in \{\exists\text{-}\texttt{BUCHI}(\mathcal{S}), \forall\text{-}\texttt{COBUCHI}(\mathcal{S}), \exists\text{-}\texttt{PARITY}(\Omega), \forall\text{-}\texttt{PARITY}(\Omega)\}$, we have the conjunction of a Büchi and a parity condition. By enriching the states with a Boolean flag for each color, we can express the conjunction as a single parity condition. Initially, all flags are set to *true* and whenever a Büchi state is reached, all flags are reset to *true*. Whenever a state with a color c is visited, the corresponding flag of the successor state is set to *false* (if the successor is not a Büchi state). In all other cases, the flags for the successor state are identical to the ones of the predecessor. Thus, a flag set to *true* indicates that this color has not been visited after the last occurrence of a Büchi state. By this we know that for a run where a color occurs infinitely often with the flag set to *true*, that also infinitely many Büchi states must have been visited. So we can define the single parity function

$$\Omega'((v,s)) = \begin{cases} \Omega(s) & \text{if } t((v,s),\Omega(s)) = \mathit{true} \vee \Omega(s) \text{ odd} \\ max\{\Omega(s) \mid s \in S \wedge \Omega(s) \text{ even}\} + 1 & \text{otherwise} \end{cases}$$

for a state (v,s), the function $t : (V \times S) \times \mathbb{N} \to \mathbb{B}$ returning the current value of the flag corresponding to the given color, and the parity condition Ω of the transit automaton. Thus, for odd colors we keep the coloring, but for even colors we keep the coloring only in case the corresponding flag in the state is set to *true*. Otherwise, we set the color to an odd number that is greater than all occurring even numbers. Thus, a play is still loosing, when it was lost due to the parity condition because we only change the coloring for even colors and the change worsens the winning possibilities. But now the play is also loosing when it wins the parity condition, but does not visit infinitely many Büchi states because it sets the flag infinitely often to *false*, but the flag is not set back to *true* infinitely often. This means, we are infinitely often in the second branch of Ω'. This applies also for all larger even colors, if these colors would then be winning. If the play visits infinitely many Büchi states, we are infinitely often in the first branch and still are winning. Parity games can be solved polynomially in the size of the arena and exponentially in the number of colors. The number of colors stem either from the

input coloring of $\mathscr{A}$ or from the determinization step. In both cases the number is linear in $|\mathscr{A}|$. Since we enrich the states of $A(\mathscr{A})$ only exponentially in the number of colors, we obtain the *single-exponential time* upper-bound.

Since solving Petri games is EXPTIME-complete [FO17] and Petri games can be easily expressed with Petri games with transits, solving Petri games with transits is EXPTIME-hard. The single-exponential upper-bound for all place-based conditions yield the EXPTIME-completeness result for the Petri games with transits with place-based conditions.

Local Flow-LTL: The size of the FTA depends on the local Flow-LTL formula φ. W.l.o.g., we consider φ in the form described in Step 1 in Sec. 12.2.3. For each flow subformula $\mathbb{E}\,\psi_i$ the NBA A_{ψ_i} is single-exponential in $|\psi_i|$ [VW94]. If this subformula is not under a negation, i.e., is an existential flow subformula, the NBA $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$ is single-exponential in $|\psi_i|$ and linear in $|\mathscr{A}|$. Otherwise the subformula is a universal subformula and the NBA $\overline{\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})}$ is double-exponential in $|\psi_i|$ and single-exponential in $|\mathscr{A}|$ due to the step building the complement [Mic88; Saf88]. Since building the unions and intersections of these NBAs can be done with an quadratic or linear blow-up [Cho74], $\Lambda_{\varphi}^{\text{F-}LTL}(\mathscr{G})$ is single-exponential in $|\varphi|$ and polynomial in $|\mathscr{A}|$ when only existential requirements are used, and double-exponential in $|\varphi|$ and single-exponential in $|\mathscr{A}|$ otherwise. The determinization of the NBA $\Lambda_{\varphi}^{\text{F-}LTL}(\mathscr{G})$ yields another exponent [Pit07; Sch09]. Thus, $\Lambda^{\text{F-}LTL}(\mathscr{G})$ is double-exponential in $|\varphi|$ and single-exponential in $|\mathscr{A}|$ when only existential requirements are used, and triple-exponential in $|\varphi|$ and double-exponential in $|\mathscr{A}|$ otherwise. For the special case that φ only consists of universal requirements, we can first apply De Morgan to put the negation at the beginning of the formula while interchanging the conjunctions and disjunctions. We then build the corresponding intersections and unions of the NBA $\Lambda_{\psi_i}^{\mathbb{E}LTL}(\mathscr{G})$, again in quadratic or linear time. Afterwards we determinize the resulting NBA to a DPA at the cost of one exponent, and finally complement the DPA without any blow-up. Thus, in this case $\Lambda^{\text{F-}LTL}(\mathscr{G})$ is also only double-exponential in $|\varphi|$ and single-exponential in $|\mathscr{A}|$. The size of the product game $\mathbb{G}(\mathscr{G})$ matches the sizes stated above for all cases because $G(\mathscr{A})$ is single-exponential in $|\mathscr{A}|$. Since the conjunction of a Büchi condition and a parity condition can be expressed as a single parity condition (see above), the complexity stated in the theorem is obtained. The reason for not obtaining another exponent even though solving parity games is in NP ∩ co-NP, is that most algorithms are only exponential in the number of colors, but polynomial in the number of states [Tho02]. However, the number of colors resulting from the determinization is only linear in size of the NBA [Sch09]. Thus, in this case, the time complexities for solving the game are subsumed by the size of the arena.

All these games admit a memoryless strategy [Zie98; SS09]. Hence, the winning strategy can be represented as a finite graph and the size of the graph is bounded by the size of $\mathbb{G}(\mathscr{G})$. Thus, Definition 59 already yields the possibility to obtain a finite representation of the strategy of the Petri game with transits in single-exponential time by using this finite graph rather than the infinite strategy tree. □

13 AdamSYNT – A Synthesis Tool for Petri Games

In this chapter we present the tool AdamSYNT for the synthesis of asynchronous distributed systems modeled with Petri games [FO17] (cp. Sec. 10.2). The implementation focuses on the solving algorithms for safe Petri games without any mixed communication and with one environment player, an arbitrary but bounded number of system players and a local safety objective. However, the source code also contains algorithms in an early development state for Petri games with transits for special subclasses like games that do not create infinitely many flow chains.

Similarly to the reduction presented in Chap. 12, AdamSYNT synthesizes winning strategies of Petri games in three steps: First, the problem is reduced to a synthesis problem of a two-player game over a finite graph. Second, a strategy of the two-player game is created (if existent). Finally, the two-player game strategy for player 0 is translated into a Petri game strategy for the system players. To tackle the immense state space of the problem, the two-player game is encoded using Binary Decision Diagrams (BDDs) [Lee59; Bry18]. Section 13.1 gives a brief insight in the encoding. In Sec. 13.2 we present the results for applying AdamSYNT to several scalable benchmark families of essential building blocks for modeling manufacturing and workflow scenarios.

AdamSYNT is open source[1] (GPL-3.0 License) and is integrated into the Adam framework (cp. Chap. 15). For the paper [FGO15], we additionally submitted a virtual machine for applying AdamSYNT on a subset of the scalable benchmark families. This artifact achieved the artifact evaluation badge of the artifact evaluation committee of CAV stating that the artifact is *consistent*, *easy to use*, *replicable*, *well-documented*, and *complete*. More technical details to the framework can be found in Chap. 15. This chapter is based on [FGO15; FGHO17].

13.1 Symbolic Encoding

Similar to the decision procedure presented in Chap. 12, in AdamSYNT, a Petri game is reduced to a two-player game over a finite graph. The states of the graph are enriched markings of the underlying Petri net. In this section, we only focus on the symbolic representation of the two-player game. More details about the reduction can be found in [FO14; FO17; FGO15; FGHO17]. The number of markings of a safe Petri net is exponential in its number of places (e.g., [Esp96]). The enrichment of these markings

[1] https://github.com/adamtool/adamsynt

does not change the theoretical complexity [FO14], but still has an impact for practical examples [FGHO17].

Binary Decision Diagrams (BDDs) [Lee59; Mor82; Jr78; Bry18] have proven to be a very effective data structure to represent large state spaces [BCMD+90; BCL91; BCMD+92; McM93]. A BDD compactly represents a Boolean function and is most commonly accompanied with an ordering constraint yielding *Ordered Binary Decision Diagrams* (OBDDs) [Bry18]. OBDDs enable the efficient application of standard operations like conjunction and negation [Bry86; DS01]. For more details about BDDs see for example [Bry92; Bry95; DS01; Bry18].

For the encoding of the two-player game, we represent the edge relation as a BDD. For each edge, we encode the source and target state as a bitvector. The representation of a state, i.e., the representation of an enriched marking of the Petri net, is organized by the tokens (a *token-based* encoding), rather than the places (a *place-based* encoding). This is motivated by the fact that the number of tokens in a Petri net is usually much smaller than the number of places. Hence, it is cheaper to encode to each token the id of the currently occupied place instead of simply providing variables for each place to represent the (arbitrary) subsets of places corresponding to the markings. A state is composed of several subvectors, one for each token. Figure 13.1 depicts such a subvector for a system token i. The first part of the bitvector encodes the place p_i the token is currently residing on and its type (`type`-1 vs. `type`-2). The second part encodes the decision of the strategy. The bit t_{i_j} for $j \in \{1, \ldots, m\}$ is set iff the player represented by token i chooses to allow for the firing of the corresponding transition of the Petri net. The $\top$-bit is set to indicate that the player is allowed and has to choose a new set of transitions (cp. Sec. 12.1). For the special case of the environment token, we only need to encode the place, without the type, $\top$-, and transition flags.

Due to the additional information necessary for the system players, the number of variables in the BDD grows significantly with the number of players in the Petri game. For optimizing the size of the BDD and the number of operators needed to encode the game, we partition the system places $\mathscr{P}_S$ into $k \in \mathbb{N}$ disjunct sets $\bigcup_{i \in \{1,\ldots,k\}} \mathscr{P}_S^i = \mathscr{P}_S$, such that for every reachable marking of the underlying Petri net $\mathscr{N}$, each place of the marking belongs uniquely to exactly one of the sets $\mathscr{P}_S^i$, i.e., $\forall M \in \mathscr{R}(\mathscr{N}), \forall i \in \{1, \ldots, k\} : |M \cap \mathscr{P}_S^i| \leq 1$. By that, we can restrict the number of variables used for the transitions for each token i to those occurring in the postset of some place in $\mathscr{P}_S^i$. Hence, we define $\mathscr{T}_i = \bigcup_{p \in \mathscr{P}_S^i} post(p)$ for all $i \in \{1, \ldots, k\}$. In general, the number k is the number of system processes in the game. As a result we have that

$$2 \cdot \left(\mathtt{log}_2(\,|\mathscr{P}_E|\,) + \sum_{i=1}^{k} \left(\mathtt{log}_2(\,|\mathscr{P}_S^i|\,) + 2 + |\mathscr{T}_i| \right) \right)$$

calculates the total number of variables used for the BDD encoding of the two-player game.

The sensitivity of BDDs to the *ordering* of the variables holds a great potential for optimizations. The size of a BDD can vary from linear to exponential in the number of input variables depending on their ordering [EFD05]. Finding an optimal ordering is NP-

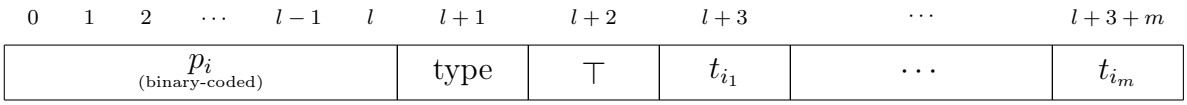


Fig. 13.1: *Bitvector representation of a cut. The subvector encodes the ith system token.*

hard [BW96], and [Sie98] shows that unless P=NP, there is no polynomial-time algorithm for finding an ordering within a constant factor of the optimum [Bry18]. Rather than finding the optimal ordering, another approach is to use heuristics and rules of thumb to find a reasonably sufficient ordering. For example, [MT98; RK08] list some of those heuristics and optimizations. In [MP15; Kam15; ADBG+17] experimental results for several variable orderings are given considering Petri net models. For our experimental results we decided for an easy ordering where the variables of the source and the target state of the encoded edge relation are interleaved. By doing so, we achieve that more frequently compared variables (due to the predecessor and successor encoding of the transitions) are closer together.

The algorithms in AdamSYNT are restricted to 1-bounded Petri games. Even though several well-known straightforward techniques like *place splitting* and *pipelining* exists for turning a k-bounded Petri net into a 1-bounded one, these techniques only preserve the language of the Petri net and not its causality structure [BDW07]. However, this structure is crucial in our setting of Petri games. In [BDW07; BW00] a technique is presented to transform a k-bounded Petri net into a safe one while preserving the causality structure. However, this technique results in an excessive blow-up in the size of the net. Hence, future work should aim to directly encode the problem for k-bounded Petri games with techniques for non-Boolean domains [MR18] like *multi-terminal binary decision diagrams (MTBDDs)* [FMY97] (also called *algebraic decision diagrams (ADDs)* [BFGH+93; BFGH+97], [Bry18]) or antichains and not aim to transform k-bounded Petri games into safe ones.

13.2 Benchmarks

We introduce five structurally different parameterized benchmark families to show the scalability of AdamSYNT. The benchmarks represent essential building blocks for modeling various manufacturing and workflow scenarios that can be analyzed automatically by synthesizing winning strategies for the system players:

Alarm System (AS) [FO14; FGHO17]: There are n geographically distributed locations. Every location is secured by an alarm system. A burglar, modeled by the environment, can intrude an arbitrary location. The alarm systems can inform each other about burglaries. The goal is that no alarm system is triggered without an intrusion and all alarm systems indicate the correct intrusion point in case of a burglary.
Parameters: n alarm systems.

Concurrent Machines (CM) [FGO15]: There are n machines which should process m orders. The orders can be processed concurrently, but no machine is allowed to process more than one order. The hostile environment chooses one machine to be defective. The goal is that finally all orders are processed.
Parameters: n machines / m orders.

Job Processing (JP) [FGO15]: A job requires processing by an ordered subset of n processors. In ascending order, the processor of the subset have to work off the job. The subset is chosen nondeterministically by the environment.
Parameters: n processors.

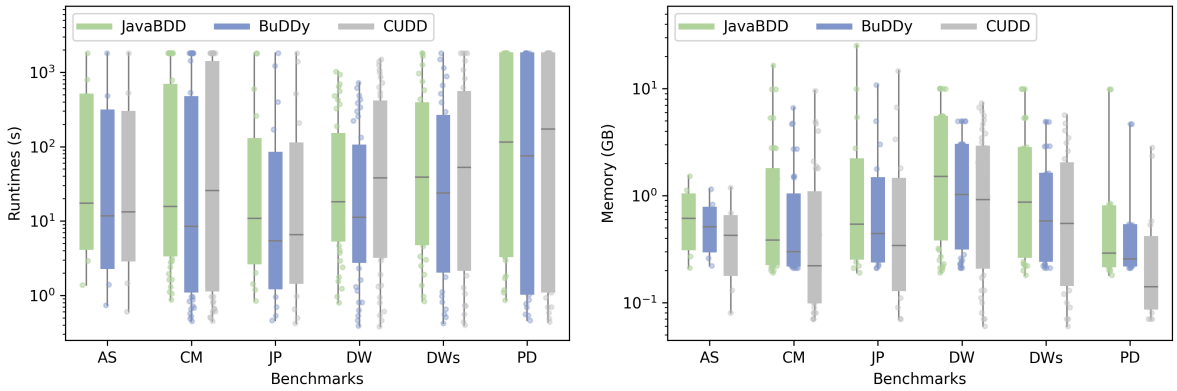
Document Workflow (DW) and (DWs) [FGO15]: There are n clerks endorsing or rejecting a document. The document is circularly passed on by the clerks. The environment decides which clerk receives the document first. The goal is that all clerks take an unanimous decision. For the simple variant (DWs), all clerks have to endorse the document.
Parameters: n clerks.

Package Delivery (PD) [GOW20]: There are n drones that should deliver m packages. Each drone can only transport one package at a time. The packages get uniquely assigned to the drones. The hostile environment lets an arbitrary drone crash. Drones get informed of the crash and can decide on recovering the package. The system's goal is to deliver all packages.
Parameters: n drones / m packages.

More details and especially a succinct high-level representation of the benchmark families can be found in [GO21; GOW20].

Table 13.1 summarizes the results using AdamSYNT with BuDDy [Lin] as BDD library for synthesizing local controllers for the system of the benchmark families above. For each benchmark of a benchmark family *Ben*, the table shows the number of tokens $\#Tok$ as well as the numbers places $|\mathscr{P}|$ and transitions $|\mathscr{T}|$ of the input Petri game for the corresponding parameters *Par*. Furthermore, we give the resulting number of BDD variables $\#Var$ used for symbolically encoding the two-player game and the elapsed CPU *time* in seconds as well as the used *memory* in GB for solving the synthesis problem. The answer to the realizability question is shown in the *ex. strat.* column by displaying a ✓ if a strategy exists and an ✗ if not. In case of the existence of a strategy, the size of the synthesized strategy is given by the number of places $|\mathscr{P}^\sigma|$ and the number of transitions $|\mathscr{T}^\sigma|$. As a result we can see that we are able to synthesize local controllers of asynchronous distributed systems for up to 59 processes. However, this number still strongly depends on the structure of the distributed system.

AdamSYNT supports the usage of different packages for the BDD representations and calculations. There is a large body of BDD libraries (e.g., CUDD [Som], BuDDy [Lin], BiDDy [Meo12], PJBDD [BFH21], CacBDD [LSX13], JDD [Vah], Sylvan [DP15; DP17], BeeDeeDee [LMS14], MEDDLY [BM10a]) and research comparing different libraries in various scenarios (e.g., in the context of probabilistic symbolic model



(a) *Comparing the runtimes.* **(b)** *Comparing the memory usage.*

Fig. 13.2: *Comparison of the runtimes and memory usage of* AdamSYNT *with the* JavaBDD *library (green), the* BuDDy *library (blue), and the* CUDD *library (gray) for the BDD calculations. We use box plots representing the median value with the horizontal gray lines, the area where the median 50% of the data is located as colored boxes, the range of the minimal and maximal values by the vertical gray lines, and the data points itself as small circles.*

checking [DHJL+15], formal concept analysis [RZS09], AI-planning problems [Jen02], and analyzing feature models [PLP11]).

For the benchmarks presented above, Fig. 13.2 shows a comparison of the time and memory usage of three different BDD libraries:

JavaBDD: [Wha]: A BDD library originally written by John Whaley in Java. It also provides an interface for calling C/C++ libraries using the Java Native Interface (JNI).

BuDDy [Lin]: A library for manipulating BDDs originally developed by Jørn Lind-Nielsen as a Ph. D. project. The implementation is written in C.

CUDD [Som]: A BDD package written in C and originally developed by Fabio Somenzi.

The results show that for all benchmark families BuDDy generally outperforms the other libraries with respect to the running times, whereas CUDD requires the least memory. Especially for experiments with smaller state spaces the CUDD library needs significantly less memory than the others. This however comes with the cost of often being the slowest library.

Tab. 13.1: *Benchmark data for* AdamSYNT *using the BDD library BuDDy. The results are obtained on an Intel i7-2700K CPU with 3.50 GHz, 32 GB RAM, and a timeout (TO) of 30 minutes. The runtimes are given in seconds, the memory in GB.*

Ben	*Par*	*#Tok*	$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{T}\rvert$	*#Var*	*time*	*memory*	$\lvert\mathscr{P}^\sigma\rvert$	$\lvert\mathscr{T}^\sigma\rvert$	ex. strat.
AS	2	4	94	17	28	1.14	.24	18	12	✓
	3	5	206	27	72	1.70	.30	32	21	✓
	4	6	402	39	152	4.11	.55	50	32	✓
	⋮		...			...		...		...
	7	9	1710	87	728	535.96	1.80	128	77	✓
	8	10	2466	107	1072	TO	TO	?	?	?
CM	2/1	4	52	12	10	.64	.22	12	8	✓
	2/2	5	82	19	16	.58	.21	-	-	✗
	2/3	6	112	26	22	.64	.22	-	-	✗
	⋮		...			...		...		...
	2/7	10	232	54	46	135.07	1.47	-	-	✗
	2/8	11	262	61	52	TO	TO	?	?	?
	⋮		...			...		...		...
	7/1	9	156	37	35	2.70	.33	72	28	✓
	7/2	10	240	59	56	9.32	.68	94	42	✓
	7/3	11	324	81	77	60.74	1.77	116	56	✓
	7/4	12	408	103	98	TO	TO	?	?	?
	⋮		...			...		...		...
	10/1	12	216	52	50	535.30	6.74	132	40	✓
	10/2	13	330	83	80	TO	TO	?	?	?
JP	2	3	46	12	13	.71	.22	16	13	✓
	3	4	76	18	23	1.07	.24	34	28	✓
	⋮		...			...		...		...
	15	16	1614	168	299	561.46	5.02	1602	1040	✓
	16	17	1904	187	335	1591.22	10.88	1906	1224	✓
	17	18	2226	207	373	TO	TO	?	?	?
DW	1	3	46	11	10	.68	.22	9	6	✓
	2	4	72	18	16	.77	.23	21	15	✓
	⋮		...			...		...		...
	31	33	782	221	190	317.82	3.05	2979	2016	✓
	32	34	808	228	196	TO	TO	?	?	?
DWs	1	3	38	11	6	.57	.21	8	3	✓
	2	5	70	21	12	.68	.22	23	10	✓
	⋮		...			...		...		...
	29	59	940	291	174	1419.64	6.99	3452	1711	✓
	30	61	972	301	180	TO	TO	?	?	?
PD	1/1	3	44	11	9	.47	.21	-	-	✗
	1/2	4	76	17	16	.53	.21	-	-	✗
	⋮		...			...		...		...
	1/7	9	236	47	51	207.68	4.67	-	-	✗
	1/8	10	268	53	58	TO	TO	?	?	?
	2/1	4	78	16	18	.56	.21	-	-	✗
	2/2	5	134	24	32	1.49	.24	26	18	✓
	2/3	6	190	32	46	10.33	.53	-	-	✗
	2/4	7	246	40	60	1170.00	8.57	-	-	✗
	2/5	8	302	48	74	TO	TO	?	?	?
	⋮		...			...		...		...
	7/1	9	292	41	93	1515.60	2.76	-	-	✗
	7/2	10	520	59	172	TO	TO	?	?	?

14 Related Work

In the literature, there is a large variety of algorithms solving the synthesis problem that differ in the choice of the architecture of the input system and the specification logic. For example, for implementations that do not interact with the environment (the *closed synthesis problem*) and only consists of a single process, the synthesis problem for LTL specifications is solved in [MW81], for CTL specifications in [CE81], and for the modal μ-calculus in [Koz83]. The *open synthesis problem*, i.e., the synthesis problem for implementations that do interact with their environment, has been solved for architectures with a single system process and linear-time specifications in an asynchronous setting in [PR89b; Var95b], and for branching-time specifications in a synchronous setting in [KV00b]. The paper [KV00a] presents a solution for the μ-calculus in a synchronous setting [Sch08].

However, the *distributed synthesis problem* [PR90; MT01; KV01a], i.e., the synthesis of systems where multiple processes interact with their environments, introduces another level of complexity resulting from the possibly different level of informedness of the individual processes. In [PR90], Pnueli and Rosner introduce a model for the synthesis of distributed systems in a *synchronous* setting, where the processes exchange their knowledge via one slot communication channels and show that in this setting the distributed synthesis problem is in general undecidable. Finkbeiner and Schewe identify information forks in the communication architecture as a sufficient criterion for the undecidability of such systems [FS05]. Unfortunately the complexity for the decidable architectures, including e.g., pipelines [PR90], two-way pipeline and one-way ring architectures [KV01a], is nonelementary [FS05]. In the *asynchronous* setting several solutions for the distributed synthesis problem are based on *Zielonka's asynchronous automata* [Zie87]. Such a model consists of a product of finite-state processes that are synchronized on shared actions. All non-shared actions can be taken individually by each process independent from the other processes. A causal memory model is used for the knowledge of the processes which is completely exchanged during the synchronizations.

In relation to the synthesis problem is the *control problem* that goes back to Ramadge and Wonham's *supervisory control* [RW89]. There, it is searched for a controller of a given plant, such that the product of the controller and the plant satisfies a given specification. The plant can be seen as an incomplete implementation of the system and the aim is to control the nondeterministic behavior of the implementation in such a way that the thereby completed implementation satisfies the given specification. Going from synthesizing implementations solely from a given specifications to the controller synthesis is somewhat fluid [BCJ18]. Generally, implementations can also be expressed in the specification language and likewise specifications can also be represented in the

implementation language [BCJ18]. In *decentralized supervisory control* [RW92; YL02], local controllers are searched for specifications given as global constraints. Considering *local specifications*, i.e., each process has its own specification, allowed to extend the decidability result for the distributed synthesis problem for pipeline architectures [PR90] to pipeline architectures allowing inputs on both sides for the controller synthesis problem [MT01].

In [GLZ04; GGMW13; MW14; Gim17] the distributed control problem is considered, where the plants and the controllers are Zielonka automata. Even though the decidability of the control problem for Zielonka automata is in general still open, there exist several decidable architectures. For instance, in [GLZ04] decidability for *series-parallel* systems with causal memory and for specifications that are recognizable on finite behavior is shown. Furthermore, in [GGMW13] a decision procedure for reachability objectives in *tree* architectures is presented that is extended in [MW14] for ω-regular local specifications for *acyclic communication* architectures. The complexity of the decision procedures is nonelementary for architectures of depth greater one. Another decidable architecture can be achieved by enforcing the processes to frequently synchronize on common actions. So-called *connectedly communicating processes* [MTY05] either communicate (directly or indirectly) within a certain bound of parallel steps or never synchronize again. The distributed control problem of such systems is also decidable [MTY05]. In [Gim17] the decidability for *decomposable games* is shown. This incorporates the previous results by showing that acyclic games are structurally decomposable, connectedly communicating games are process decomposable, and series-parallel games are action decomposable.

Another framework for the distributed synthesis is the model on which the second part of this thesis is built (cp. Sec. 10.2). *Petri games* have been introduced in [FO14] and the corresponding journal version [FO17]. There, Finkbeiner and Olderog introduce an EXPTIME-complete decision procedure for Petri games with one environment, a bounded number of system players, and a local safety condition based on avoiding dedicated bad places. Furthermore, the undecidability of unbounded Petri games is established. The decision procedure is implemented in [FGO15] with an accompanying web interface [GHY21]. In [Spr15] preliminary work for Petri games with bad markings is presented and [Hec21; FGHO22] present a decision procedure for one environment player, a bounded number of system players, and a global safety condition based on avoiding bad markings with 2-EXPTIME complexity. For one system player, a bounded number of environment players, and a global safety condition based on avoiding bad markings [Göl17; FG17] introduce a EXPTIME-complete decision procedure. In [Han19] a decision procedure for two environment players, two system players, and a global safety condition based on avoiding bad markings with a synchronization condition restricting the communication possibilities of the players is introduced. For extending the memory model of Petri games, there is some preliminary work introducing *forgetful places* [Buh19], i.e., places in which the players loose their memory. Furthermore, preliminary work for making arbitrary bounded Petri games *concurrency-preserving*, i.e., no players can be created or terminate, without information leakage, is presented in [Sch19].

A *bounded synthesis* approach for Petri games is introduced in [Fin15] and an implementation that encodes the problem in *quantified Boolean formulas (QBF)* using

QuAbS [Ten16; HT18] as a solver is provided in [FGHO17]. Bounded synthesis algorithms for Petri games with a *true concurrency semantics*, i.e., subsuming several interleavings of concurrent transitions by a single one whenever possible, is introduced in [Met17; HM19]. High-level Petri games based on high-level Petri nets are introduced in [GO21] and solving algorithms exploiting the symmetries in the system are developed in [GOW20; GW21a; Wür21]. Petri games can be translated into the *control games* based on Zielonka's asynchronous automata with an exponential blow-up [Beu19; BFH19]. The same applies for the other direction [Beu19; BFH19].

Different to the reduction method for Petri games presented in [FO14; FO17], in the reduction presented in this thesis, we do not outsource the `type`-2 case into a precalculation, but integrate it into the information flow game. This is necessary for handling the new winning conditions based on the local data flow. To integrate the `type`-2 case in the information flow game, we use the generations and the round robin procedure instead of the three valued flag for each system player and the additional memory for the repeating marking used in [Hec21; FGHO22], because the winning condition defined on the local data flow may need additional unrollings of loops and we cannot directly exploit the idea of useless repetitions for Petri games with transits as in [Hec21; FGHO22] for Petri games. To deal with the nondeterminism of the strategy, we use the restriction to Petri games without mixed communication instead of the backward moves from [Hec21; FGHO22], to avoid the additional complexity. Moreover, this global view would thwart the idea of the locality of the new winning conditions.

There is a large body of work regarding *Petri net synthesis* [DR96b; BBD15], i.e., the quest for a structural description of a concurrent system, given a behavioral one. Originally, the problem has been formulated as the search for a Petri net with a reachability graph that is isomorphic to a given transition system. This problem has been introduced and solved by Ehrenfeucht and Rozenberg using the theory of *regions* of transition systems [ER90a; ER90b]. A significant part of the application of this work either concerns the synthesis without any environment [Dar07; BBD15] and/or the control problem of a single-process [HKG97; RXG00]. In [Dar05; DR12; BBD15], however, another distributed version of the Ramadge and Wonham setting is considered using region theory. There, the events are distributed over locations with one local controller per location that observes the observable and controls the controllable events of its location [BBD15].

15 ADAM – Analyzer of Distributed Asynchronous Models

In this chapter we present the combination of the two tools developed accompanying this thesis. The tool is named ADAM in honor of Carl Adam Petri, who laid the foundations of Petri nets. Conveniently, however, the name is also an acronym for **A**nalyzer of **D**istributed **A**synchronous **M**odels. We have constructed a single *framework* to deal with the common features for the model checking and the synthesis approaches developed in ADAMMC (cp. Chap. 7) and ADAMSYNT (cp. Chap. 13). The framework is designed modularly, so that individual parts can be build separately and used as libraries in other projects. In Sec. 15.1, we give a brief overview on the structure of this framework.

Furthermore, we have implemented two user interfaces for ADAM. The *web interface* ADAMWEB[1] [GHY20; GHY21] allows for an intuitive, visual definition of Petri nets with transits and Petri games. The corresponding model checking and synthesis problems are solved directly on a server. In the interface, implementations, counterexamples, and all intermediate steps can be analyzed and simulated. Stepwise simulations and interactive state space generation supports the user in detecting modeling errors in the distributed system to tackle the validation problem. The *command-line interface* offers considerably more possibilities to tweak the algorithms of the different approaches because not all parameters of the algorithms are yet provided in the web interface. In general, the command-line tool is more suitable for benchmarking purposes and can be used more easily for the comparison with other tools. All tools are available online and open source.

This chapter is based on [GHY21].

15.1 Framework

We have developed a framework covering the approaches for model checking and synthesis presented in this thesis. The focus of the implementation is on the adaptability, ease to use, and generality of the data structures and algorithms, rather than on their speed, as the computationally intensive work is done by the external tool ABC [Ber; BM10b] or the BDD libraries [Lin; Som]. In general, the reduction algorithms take up a fraction of the total computation time and also the overhead for calling the C libraries is negligible [Iri13]. However, for the model checking approach with Flow-CTL specifications, the automata reduction may already require a lot of computing time. Currently, these

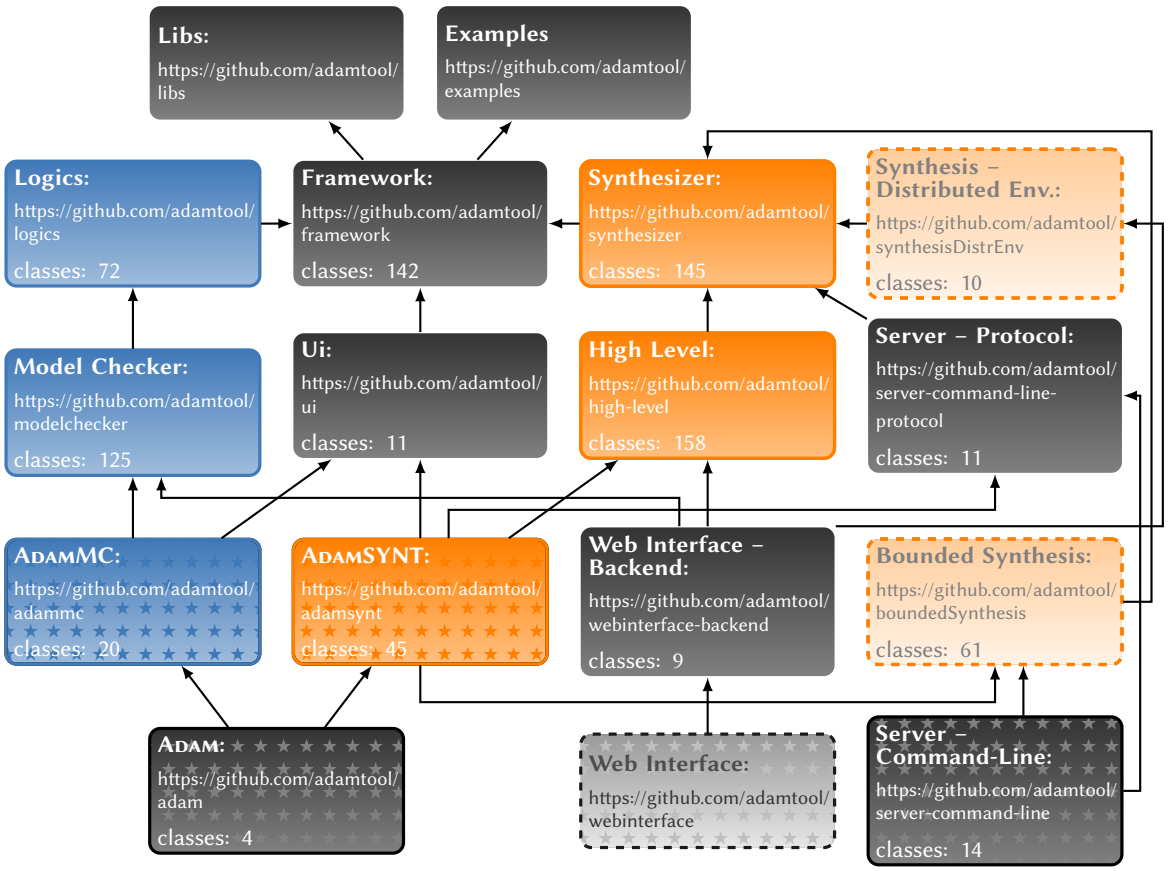


Fig. 15.1: *Overview of the repositories corresponding to the* ADAM *framework. Blue colors indicate repositories mainly corresponding to the model checking approach, whereas oranges colors belong to the synthesis approach. The star pattern identifies repositories corresponding to executables and the dashed lighter packages mark the repositories with other leading developers.*

algorithms are only used as proof of concept, are not optimized, and are still in an early state of development.

We base our data structure for Petri nets with transits and Petri games on the data structures for Petri nets provided by [Uni12]. Figure 15.1 gives an overview of the repositories that make up the ADAM project[2]. The framework has also been used in projects not developed by the author of this thesis. Jesko Hecking-Harbusch and Niklas Metzger developed bounded synthesis algorithms for Petri games with a global safety objective [FGHO17; HM19] in the repository "**Bounded Synthesis**", and Lukas Panneke implemented the synthesis approach for Petri games with one system player, a bounded number of environment players, and a global safety objective [FG17] in the repository "**Synthesis – Distributed Env.**". The front-end of the web interface and the web server (repository "**Web Interface**") is developed by Ann Yanich. The repository "**Adam**" and

[1] http://adam.informatik.uni-oldenburg.de
[2] https://github.com/adamtool/

"**Web Interface**" have their dependencies integrated as git submodules and the other repositories provide scripts to automatically checkout all necessary dependencies (also on specific versions). All repositories offer scripts for properly building the libraries or tools.

15.2 Web Interface

In this section we give some insights into the *web interface* for the tools AdamMC and AdamSYNT with its possibilities to simulate or interactively explore implementations, counterexamples, and parts of the created state space for both applications. The web interface offers the input of Petri nets with transits and Petri games, where the user can interactively creates places, transitions, and their connections with a few inputs.

A screenshot for the model checking approach is given in Fig. 15.2 and one for the synthesis approach in Fig. 15.3 with the tool bars on the left to interactively create the input models.

15.2.1 The Model Checking Approach

As back-end for the *model checking* approach, the algorithms of AdamMC are used to model check Petri nets with transits against a given Flow-LTL formula as specification. Internally, the problem is reduced to the model checking problem of Petri nets against LTL. Both, the input Petri net with transits and the constructed Petri net can be visualized, simulated, and manually arranged in the web interface. Automatic layout techniques may be applied to avoid the overlapping of nodes. In addition, a so-called *physics control*, which modifies the repulsion, link, and gravity strength of nodes, can be used to minimize the overlapping of edges. Heuristics generate coordinates for the constructed Petri net by using the coordinates of the input Petri net with transits to obtain a similar layout of corresponding parts.

For a positive result, the web interface allows the user to follow the control flow of the combined system and the data flow of the components of simulated runs of the net by visualizing the data flow trees corresponding to the input firing sequence. For a negative result, the web interface allows the user to simulate the counterexample both in the Petri net with transits and in the Petri net from the reduction. The witness of the counterexample for each flow subformula and the run violating the requirement on the global behavior can be displayed by the web interface. This functionality is helpful when developing an encoding of a problem into Petri net with transits to ensure that a counterexample is not an error in the encoding. The constructed Petri net can be exported into a standard format for Petri net model checking (PNML) and the constructed LTL formula can be displayed. A complete documentation with the features and example workflows is available online[3].

[3] https://github.com/adamtool/webinterface/tree/master/doc/mc#readme

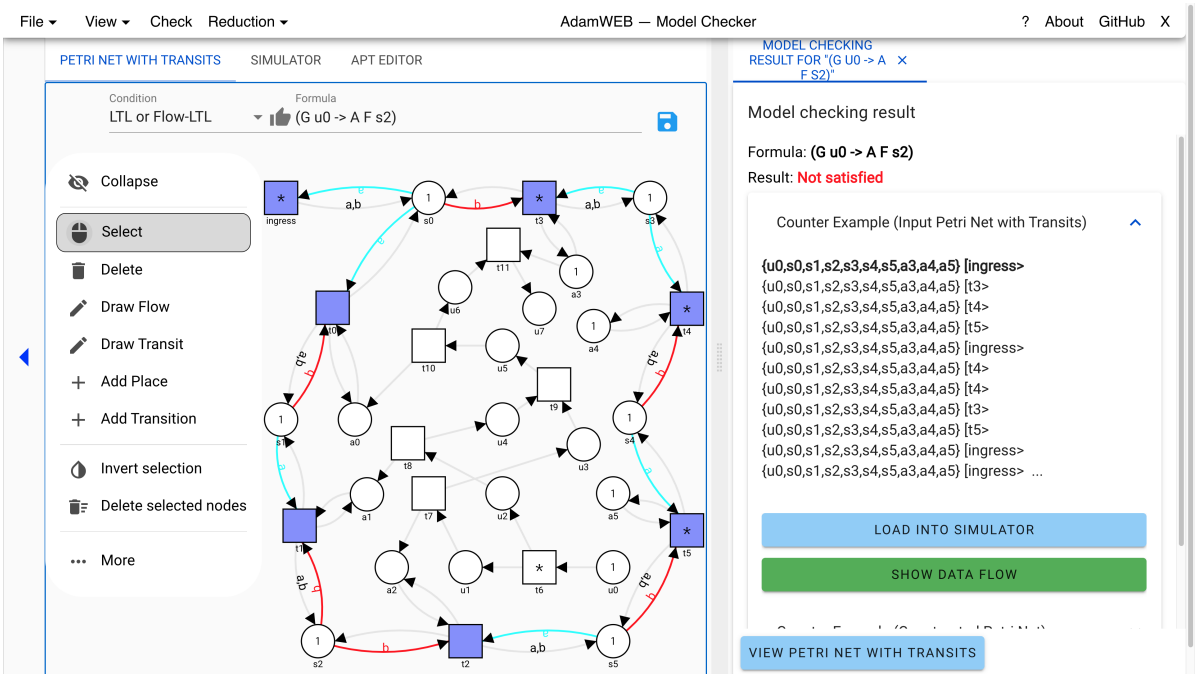


Fig. 15.2: *Screenshot from the web interface for the model checking approach. The input Petri net with transits and the specification is given on the left, whereas the result and a textual representation of the counterexample is presented on the right. The transits are depict as colored arcs or with letters equipped to the arcs. The purple color of a transition indicates the weak fairness property of this transition.*

15.2.2 The Synthesis Approach

As a back-end for the *synthesis* approach, the algorithms of ADAMSYNT solve a given Petri game with a local safety specification. Internally, the synthesis problem for Petri games with a bounded number of system players, one environment player, and a local safety objective is reduced to the synthesis problem for finite-state two-player games with complete information.

For a positive result, a winning strategy in the two-player game is translated into a winning strategy for the Petri game. Both can be visualized in the web interface. Here, the web interface provides the same features for visualizing, manipulating, and automatically laying out the elements as for model checking. It uses the order of nodes of the Petri game to heuristically provide a positioning of the strategy and allows for the simulation of runs of the strategy. The winning strategy of the two-player game provides an additional view on the implementation to check if it is not bogus due to a forgotten case in the Petri game or the specification. For a negative result, i.e., an unrealizable synthesis problem, the web interface allows for interactively analyzing the underlying two-player game via a stepwise creation of strategies. This guides the user towards changes to make the problem realizable. A complete documentation with the

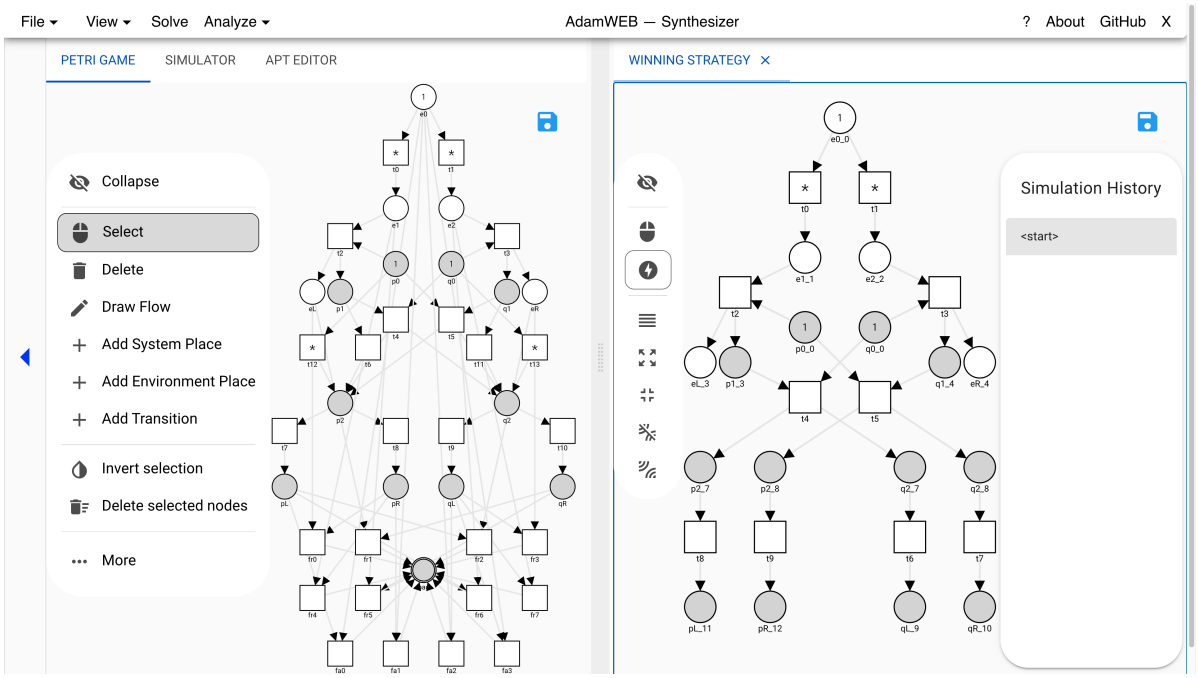


Fig. 15.3: *Screenshot from the web interface for the synthesis approach. The input Petri game is presented on the left, whereas the corresponding winning strategy is depicted on the right.*

features and example workflows is available online[4].

15.2.3 Implementation Details

The server is implemented using the Sparkjava micro-framework [Spa] for incoming HTTP and WebSocket connections. The client is a single-page application written in Javascript using Vue.js [Vuea], D3 [D3], and the Vuetify component library [Vueb]. Both components are implemented by Ann Yanich. Libraries are constructed out of the tools AdamMC and AdamSYNT and one interface has been implemented handling both libraries.

15.3 Command-line Interface

We provide three tools with a *command-line interface* for the analysis of distributed asynchronous systems: AdamMC for the model checking algorithms (cp. Chap. 7), AdamSYNT for the synthesis algorithms (cp. Chap. 13), and Adam combining the features of both tools. Since the tools are based on the same repositories for the program logic and data structures, they offer the same main functionalities as the web interface.

In addition, the command-line interfaces provide a broad variety of parameters for tweaking the different approaches. These are listed in the help dialogue of the individual

[4] https://github.com/adamtool/webinterface/tree/master/doc/synt#readme

commands. If available, the input and output objects can be visualized using Graphviz and the dot language [EGKN+04]. Generators for constructing scalable Petri nets with transits and Petri games are available as well as features to automatically generate the function for the bash-completion of the current state of the tool.

More details can be found in the documentations available in the repositories.

16 Conclusion

This final chapter of the thesis summarizes the results and addresses some interesting directions for future work.

16.1 Summary

This thesis contributes to the *correct* development of *asynchronous distributed systems* following two complementing approaches: *model checking* and *synthesis*. For both approaches, we introduced new *modeling* and *specification* formalisms that enable requirements on the unbounded data flow in asynchronous distributed systems. This thesis provided *solving algorithms* for the corresponding model checking and synthesis problems with a reasonable complexity, despite the unbounded features of the data flow and the incomplete knowledge of the system's components about the system's environment in asynchronous distributed systems. The *implementations* of these algorithms deal with the state explosion problem, resulting from the different schedules of the asynchronous components in a distributed system, via a reduction to a hardware model checking problem or by using symbolic algorithms, respectively.

In the first part, a new framework for *model checking* asynchronous distributed systems has been introduced. The model (*Petri nets with transits*) and the specification language (*Flow-CTL**) allow for a clear separation of the control part of the system and the local data flow of its processes. Both formalisms are extensions of well-established concepts which have proven to be particularly suitable in such contexts. The separation allows us to individually focus on the respective aspects of the system. Controlling the behavior of the distributed system is done by the Petri nets whereas the correctness is stated on the unbounded local data flow of the processes. This allows us to select specific runs of the system, e.g., those adhering to fair schedulings, and to only check the correctness of the selected runs. As a result, a specification can be composed of individual requirements for the data flow depending on different control runs. Different algorithms with affordable costs have been introduced to automatically check whether such a system satisfies a given specification. For the Flow-LTL fragment of Flow-CTL*, two optimized algorithms have been introduced: a sequential and a parallel one. Despite having a worse theoretical complexity in general, the experimental results show a superiority of the parallel approach in the scenarios considered. Finally, the problem has been reduced to a hardware model checking problem in order to use the state-of-the-art algorithms and toolboxes provided in this setting. An implementation for subclasses of

the framework is available with AdamMC[1] and shows the practical applicability of the presented algorithms.

The second part of the thesis introduced a new framework for the *synthesis* of asynchronous distributed systems with *causal memory*. The model (*Petri games with transits* with *existential*, *universal*, and *local Flow-LTL* winning conditions) has been introduced, in which the correctness of the system can be stated at the level of the unbounded local data flow of the processes. This allows for the first time to address the synthesis problem for asynchronous distributed systems with Petri games, where the specification goes beyond safety requirements. A uniform reduction method for all winning conditions has been presented for systems with one environment player and a bounded number of system players that do not allow any mixed communications. The resulting two-player game over a finite graph with complete information can be solved by established state-of-the-art algorithms. As a result, local controllers for the processes can be synthesized with affordable costs for the new existential and universal safety, reachability, Büchi, co-Büchi, and parity, as well as for the local Flow-LTL specifications. The presented solving algorithm reduces the intricate causal memory model of the players in the Petri game with transits to a complete information model detached from the winning property of the strategy. This enabled us to reduce the general existence of a strategy for the system players in the Petri game with transits to the solving of a two-player Büchi game, while the winning property of the strategy is reduced to the acceptance by suitably constructed automata. This separation accounts for the generality of the presented solving algorithm and facilitates simpler extensibility to different winning conditions. To show the practical applicability, another tool, AdamSYNT[2], has been implemented such that local controllers for subclasses of the framework can be fully automatically synthesized. A web interface is online at `http://adam.informatik.uni-oldenburg.de` to provide easy and interactive access to a selection of the features of both tools.

16.2 Future Work

There are several interesting aspects and starting points on how to build upon the results of this thesis. In the following we list some of them.

Extending the Specification Language: For Flow-CTL* we consider *all* runs, *all* firing sequences, and *all* data flow chains (or an *existing* run with an *existing* firing sequence, and an *existing* data flow chain, respectively) to check whether a formula is satisfied. We can extend this logic by also considering other combinations of the quantors for the runs (**R**), the firing sequences (**F**), and the data flow chains (**D**). However, the $\exists\mathbf{R}\forall\mathbf{F}Q\mathbf{D}$-fragments, with $Q \in \{\forall, \exists\}$ (or the $\forall\mathbf{R}\exists\mathbf{F}Q\mathbf{D}$-fragments, respectively), are most likely undecidable. A reduction to the Post correspondence problem (PCP) should be possible by introducing one independent process for each of the list of words similarly to [FGHO22]. Each process can nondeterministically decide for a word of its list and

[1] `https://github.com/adamtool/adammc`
[2] `https://github.com/adamtool/adamsynt`

outputs the index followed by the corresponding letters. Here we have to use helping processes to select the letters because we do not have labels for the transitions of the Petri net. A selected concurrent run ($\exists\mathbf{R}$) then contains the sequence of indices and the concatenated words. Since we consider *all* firing sequences of the run ($\forall\mathbf{F}$), we can select with the formula the firing sequence in which the indices of both processes alternate and check their correspondence. Similarly, we can select the firing sequence in which the letters alternate and check their correspondence. Thus, each of the firing sequence can be seen as a synchronized progression of the processes similar to setting of the undecidability proof by Pnueli and Rosner [PR90].

Reduction Techniques: To tackle the state-explosion problem, AdamSYNT already contains algorithms exploiting the symmetries in the system [GOW20; GW21a], and uses symbolic algorithms by representing the two-player game with BDDs [FGO15; FGHO17]. A further interesting step is to investigate whether other reduction techniques for the state space of Petri nets can also be applied for Petri net with transits or Petri games. There is a large body of work regarding model checking and *partial order reductions* (see e.g. [CGP01; Pel18]). For example [Tiu94] investigates how to use *stubborn sets* [Val89; Val90; Val92; KV98] together with BDDs for the reachability analysis in safe Petri nets. Another approach is to reduce the input Petri net before its analysis, while still preserving specific properties of the system. Such *structural reductions* are for example presented in [Mur89; Sta90; Thi20]. In [BDJJ+19] the combination of structural reductions and stubborn sets for the reachability analysis in Petri nets is investigated and [JSUV22] provides algorithms for using stubborn sets to improve LTL model checking on Petri nets. Some conscientious preliminary work for comparing structural reductions, stubborn sets, and BDDs, as well as some combinations of these techniques, is already done in [Pan21] together with an implementation for the reachability analysis in bounded Petri nets.

Decomposing the Input Problem: Another approach that enables the synthesis of large-scale asynchronous distributed systems is to decompose the input problem into smaller problems, solve these simpler problems, and, in the end, reassemble an original solution from the individual smaller solutions. This would enable us to solve large Petri games with transits which cannot be solved as a whole but for which each independent component can be solved. However, most Petri games with transits cannot simply be decomposed and analyzed individually without information about the surrounding in which the component is embedded. At this point, the *rely-guarantee paradigm* [Sta85; AL89; AL93] can be helpful to identify constraints a component game relies on and that can be guaranteed by the system. For making such guarantees formal, a knowledge operator [FHMV95] may be used for expressing the level of informedness of the specific players at the beginning of the component game.

Memory Model: In Petri games *causal memory* is used to represent the knowledge of the players. In this model, all players involved in a joint transition exchange their *complete* knowledge and the players can never forget any information. Thus, no player (especially the environment) can hide any information while interacting with the other

players. Lifting these restrictions and introducing *partial observation* [CD10; CDH13] is an interesting direction for further work but bears the risk of making an already computationally hard problem even harder. One direction of extending the memory model is to introduce labels on the transitions of the Petri game with transits and let equally labeled transitions become indistinguishable to the players. Another direction is to introduce *forgetful places* where the causal histories are merged in the strategy, i.e., the causal past of the token reaching such a place is lost and thus cannot be communicated in the next joint transition. However, adding the concept of forgetful places enables us to enforce a synchronization between processes without information leakage. Thus, most likely the general setting is undecidable due to Pnueli and Rosner [PR90] and makes it especially interesting for the bounded synthesis approaches [Fin15; FGHO17; Hec21]. A good starting point for the syntactical and semantic definition of forgetful places is given in [Buh19].

Another possibility for the environment to hide certain information is to introduce more environment players. A good starting point for this topic is [Han19], where a decision procedure for two environment and two system players with a bound on the communication depth until the new information is propagated to all players is introduced.

Implementations with Side Constraints: A solution to the synthesis problem is generally not unambiguous. An interesting question is how to guide the search for strategies satisfying given side constraints. For example, bounded synthesis approaches [SF07; FS13] are tailored to find small strategies. For Petri games it could be of special interest to find strategies using the least communication between the players to keep the single components as independent as possible and minimize the possible errors due to communication problems. A related and not less interesting question is where do we have to add communication in the system to provide a winning strategy in case the current model does not have any winning strategy.

Bibliography

[AL89] Martín Abadi and Leslie Lamport. "Composing Specifications". In: *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*. 1989, pp. 1–41. DOI: 10.1007/3-540-52559-9_59.

[AL93] Martín Abadi and Leslie Lamport. "Composing Specifications". In: *ACM Trans. Program. Lang. Syst.* 15.1 (1993), pp. 73–132. DOI: 10.1145/151646.151649.

[ABd03] Parosh Aziz Abdulla, Ahmed Bouajjani, and Julien d'Orso. "Deciding Monotonic Games". In: *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*. 2003, pp. 1–14. DOI: 10.1007/978-3-540-45220-1_1.

[ADBG+17] Elvio Gilberto Amparore, Susanna Donatelli, Marco Beccuti, Giulio Garbi, and Andrew S. Miner. "Decision Diagrams for Petri Nets: Which Variable Ordering?" In: *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'17), co-located with the 38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and the 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 25-30, 2017*. 2017, pp. 31–50.

[ADG20] Elvio Gilberto Amparore, Susanna Donatelli, and Francesco Gallà. "A CTL* Model Checker for Petri Nets". In: *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020, Paris, France, June 24-25, 2020, Proceedings*. 2020, pp. 403–413. DOI: 10.1007/978-3-030-51831-8_21.

[AQRX04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. "Zing: Exploiting Program Structure for Model Checking Concurrent Software". In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. 2004, pp. 1–15. DOI: 10.1007/978-3-540-28644-8_1.

[AG11] Krzysztof R. Apt and Erich Grädel, eds. *Lectures in Game Theory for Computer Scientists*. Cambridge University Press, 2011.

[AC88] André Arnold and Paul Crubillé. "A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems". In: *Inf. Process. Lett.* 29.2 (1988), pp. 57–66. DOI: 10.1016/0020-0190(88)90029-4.

[BBCP+09] Souheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. "The GreatSPN tool: recent enhancements". In: *SIGMETRICS Perform. Evaluation Rev.* 36.4 (2009), pp. 4–9. DOI: 10.1145/1530873.1530876.

[BM10a] Junaid Babar and Andrew S. Miner. "Meddly: Multi-terminal and Edge-Valued Decision Diagram LibrarY". In: *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010*. 2010, pp. 195–196. DOI: 10.1109/QEST.2010.34.

[BKRS12] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. "LTL to Büchi Automata Translation: Fast and More Deterministic". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 95–109. DOI: 10.1007/978-3-642-28756-5_8.

[BBD15] Éric Badouel, Luca Bernardinello, and Philippe Darondeau. *Petri Net Synthesis*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2015. DOI: 10.1007/978-3-662-47967-4.

[BFGH+93] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic decision diagrams and their applications". In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*. 1993, pp. 188–191. DOI: 10.1109/ICCAD.1993.580054.

[BFGH+97] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 171–206. DOI: 10.1023/A:1008699807402.

[BBS11] Yu Bai, Jens Brandt, and Klaus Schneider. "Data-Flow Analysis of Extended Finite State Machines". In: *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*. 2011, pp. 163–172. DOI: 10.1109/ACSD.2011.22.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BS93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. "An Iterative Approach to Language Containment". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 1993, pp. 29–40. DOI: 10.1007/3-540-56922-7_4.

[BBGI+14] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. "VeriCon: towards verifying controller programs in software-defined networks". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, pp. 282–293. DOI: 10.1145/2594291.2594317.

[BCRR+09] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. "Real life challenges in access-control management". In: *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*. 2009, pp. 899–908. DOI: 10.1145/1518701.1518838.

[BMJA+01] Michel Beaudouin-Lafon, Wendy E. Mackay, Mads Jensen, Peter Andersen, Paul Janecek, Henry Michael Lassen, Kasper Lund, Kjeld Høyer Mortensen, Stephanie Munck, Anne V. Ratzer, Katrine Ravn, Søren Christensen, and Kurt Jensen. "CPN/Tools: A Tool for Editing and Simulating Coloured Petri Nets ETAPS Tool Demonstration Related to TACAS". In: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 2001, pp. 574–577. DOI: 10.1007/3-540-45319-9_39.

[Ber] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Version 1.01 81030.

[BBDF+01] S. Bernardi, C. Bertoncello, S. Donatelli, G. Franceschinis, G. Gaeta, M. Gribaudo, and A. Horvàth. *GreatSPN in the new Millenium*. Tools of Aachen 2001, International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems 760/2001. Universitat Dortmund (Germany), 2001, pp. 17–23.

[BRV04] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. "The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets". In: *International journal of production research* 42.14 (2004), pp. 2741–2756.

[BV06] Bernard Berthomieu and François Vernadat. "Time Petri Nets Analysis with TINA". In: *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. 2006, pp. 123–124. DOI: 10.1109/QEST.2006.56.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. DOI: 10.1007/978-3-662-07964-5.

[BDW07] Eike Best, Philippe Darondeau, and Harro Wimmel. "Making Petri Nets Safe and Free of Internal Transitions". In: *Fundam. Informaticae* 80.1-3 (2007), pp. 75–90.

[BF88] Eike Best and César Fernández. *Nonsequential Processes - A Petri Net View*. Vol. 13. EATCS Monographs on Theoretical Computer Science. Springer, 1988. DOI: 10.1007/978-3-642-73483-0.

[BW00] Eike Best and Harro Wimmel. "Reducing k-Safe Petri Nets to Pomset-Equivalent 1-Safe Petri Nets". In: *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding*. Ed. by Mogens Nielsen and Dan Simpson. Vol. 1825. Lecture Notes in Computer Science. Springer, 2000, pp. 63–82. DOI: 10.1007/3-540-44988-4_6.

[Beu19] Raven Beutner. "Translating Asynchronous Games for Distributed Synthesis". Bachelor's Thesis. Saarland University, Saarbrücken, Germany, 2019.

[BFH19] Raven Beutner, Bernd Finkbeiner, and Jesko Hecking-Harbusch. "Translating Asynchronous Games for Distributed Synthesis". In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. 2019, 26:1–26:16. DOI: 10.4230/LIPIcs.CONCUR.2019.26.

[BFH21] Dirk Beyer, Karlheinz Friedberger, and Stephan Holzner. "PJBDD: A BDD Library for Java and Multi-Threading". In: *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*. 2021, pp. 144–149. DOI: 10.1007/978-3-030-88885-5_10.

[BCCF+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". In: *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*. 1999, pp. 317–320. DOI: 10.1145/309847.309942.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[BCRZ99] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. "Verifying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. 1999, pp. 60–71. DOI: 10.1007/3-540-48683-6_8.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, July 2011.

[BCHK+03] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. "The Petri Net Markup Language: Concepts, Technology, and Tools". In: *Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings*. 2003, pp. 483–505. DOI: 10.1007/3-540-44919-1_31.

[BCGH+10] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. "Robustness in the Presence of Liveness". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 410–424. DOI: 10.1007/978-3-642-14295-6_36.

[BCJ18] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. "Graph Games and Reactive Synthesis". In: *Handbook of Model Checking*. 2018, pp. 921–962. DOI: 10.1007/978-3-319-10575-8_27.

[BGJP+07a] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. "Automatic hardware synthesis from specifications: a case study". In: *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*. 2007, pp. 1188–1193.

[BGJP+07b] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. "Specify, Compile, Run: Hardware from PSL". In: *Electron. Notes Theor. Comput. Sci.* 190.4 (2007), pp. 3–16. DOI: 10.1016/j.entcs.2007.09.004.

[BGHK+12] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. "Synthesizing Robust Systems with RATSY". In: *Proceedings First Workshop on Synthesis, SYNT 2012, Berkeley, California, USA, 7th and 8th July 2012*. 2012, pp. 47–53. DOI: 10.4204/EPTCS.84.4.

[BJPP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) designs". In: *J. Comput. Syst. Sci.* 78.3 (2012), pp. 911–938. DOI: 10.1016/j.jcss.2011.08.007.

[BBFJ+12] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. "Acacia+, a Tool for LTL Synthesis". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, pp. 652–657. DOI: 10.1007/978-3-642-31424-7_45.

[Bok18] Udi Boker. "Why These Automata Types?" In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. 2018, pp. 143–163. DOI: 10.29007/c3bj.

[BK09] Udi Boker and Orna Kupferman. "Co-ing Büchi Made Tight and Useful". In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. 2009, pp. 245–254. DOI: 10.1109/LICS.2009.32.

[BW96] Beate Bollig and Ingo Wegener. "Improving the Variable Ordering of OBDDs Is NP-Complete". In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122.

[BDJJ+19] Frederik M. Bønneland, Jakob Dyhr, Peter Gjøl Jensen, Mads Johannsen, and Jirí Srba. "Stubborn versus structural reductions for Petri nets". In: *J. Log. Algebraic Methods Program.* 102 (2019), pp. 46–63. DOI: 10.1016/j.jlamp.2018.09.002.

[BW18] Julian C. Bradfield and Igor Walukiewicz. "The mu-calculus and Model Checking". In: *Handbook of Model Checking*. 2018, pp. 871–919. DOI: 10.1007/978-3-319-10575-8_26.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[BHPV00] Guillaume Brat, Klaus Havelund, Seungjoon Park, and Willem Visser. "Java PathFinder - Second Generation of a Java Model Checker". In: *In Proceedings of the Workshop on Advances in Verification*. 2000.

[BM10b] Robert K. Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[Bru21] Roberto Bruttomesso. "Intrepid: A Scriptable and Cloud-Ready SMT-Based Model Checker". In: *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*. 2021, pp. 202–211. DOI: 10.1007/978-3-030-85248-1_13.

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[Bry92] Randal E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams". In: *ACM Comput. Surv.* 24.3 (1992), pp. 293–318. DOI: 10.1145/136035.136043.

[Bry95] Randal E. Bryant. "Binary decision diagrams and beyond: enabling technologies for formal verification". In: *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*. 1995, pp. 236–243. DOI: 10.1109/ICCAD.1995.480018.

[Bry18] Randal E. Bryant. "Binary Decision Diagrams". In: *Handbook of Model Checking*. 2018, pp. 191–217. DOI: 10.1007/978-3-319-10575-8_7.

[BL69] J. Richard Buchi and Lawrence H. Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311.

[Buh19] Moritz Buhr. "Forgetful Petri Games – Synthesizing Distributed Systems with Partially Observable Causal Memory". Bachelor's Thesis. University of Oldenburg, Oldenburg, Germany, 2019.

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. "Symbolic Model Checking with Partitioned Transistion Relations". In: *VLSI 91, Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, Edinburgh, Scotland, 20-22 August, 1991*. 1991, pp. 49–58.

[BCMD90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. "Sequential Circuit Verification Using Symbolic Model Checking". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*. 1990, pp. 46–51. DOI: 10.1145/123186.123223.

[BCMD+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. 1990, pp. 428–439. DOI: 10.1109/LICS.1990.113767.

[BCMD+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In: *Inf. Comput.* 98.2 (1992), pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A.

[BJS09] Joakim Byg, Kenneth Yrke Jørgensen, and Jirí Srba. "TAPAAL: Editor, Simulator and Verifier of Timed-Arc Petri Nets". In: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. 2009, pp. 84–89. DOI: 10.1007/978-3-642-04761-9_7.

[CJKL+17] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. "Deciding parity games in quasipolynomial time". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. 2017, pp. 252–263. DOI: 10.1145/3055399.3055409.

[CVPK+12] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. "A NICE Way to Test OpenFlow Applications". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. 2012, pp. 127–140.

[CFG14] Martín Casado, Nate Foster, and Arjun Guha. "Abstractions for software-defined networks". In: *Commun. ACM* 57.10 (2014), pp. 86–95. DOI: 10.1145/2661061.2661063.

[CHJS18] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. "Extending Automata Learning to Extended Finite State Machines". In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*. 2018, pp. 149–177. DOI: 10.1007/978-3-319-96562-8_6.

[CFJM16] Pavol Cerný, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. "Optimal Consistent Network Updates in Polynomial Time". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 2016, pp. 114–128. DOI: 10.1007/978-3-662-53426-7_9.

[CG18] Sagar Chaki and Arie Gurfinkel. "BDD-Based Symbolic Model Checking". In: *Handbook of Model Checking*. 2018, pp. 219–245. DOI: 10.1007/978-3-319-10575-8_8.

[CD10] Krishnendu Chatterjee and Laurent Doyen. "The Complexity of Partial-Observation Parity Games". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*. 2010, pp. 1–14. DOI: 10.1007/978-3-642-16242-8_1.

[CDH13] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. "A survey of partial-observation stochastic parity games". In: *Formal Methods Syst. Des.* 43.2 (2013), pp. 268–284. DOI: 10.1007/s10703-012-0164-2.

[CDHL16] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Veronika Loitzenbauer. "Conditionally Optimal Algorithms for Generalized Büchi Games". In: *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*. 2016, 25:1–25:15. DOI: 10.4230/LIPIcs.MFCS.2016.25.

[CH12] Krishnendu Chatterjee and Monika Henzinger. "An $O(n^2)$ time algorithm for alternating Büchi games". In: *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*. 2012, pp. 1386–1399. DOI: 10.1137/1.9781611973099.109.

[CH14] Krishnendu Chatterjee and Monika Henzinger. "Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-Component Decomposition". In: *J. ACM* 61.3 (2014), 15:1–15:40. DOI: 10.1145/2597631.

[CHP07] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. "Generalized Parity Games". In: *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings*. 2007, pp. 153–167. DOI: 10.1007/978-3-540-71389-0_12.

[CHP08] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. "Algorithms for Büchi Games". In: *CoRR* abs/0805.2620 (2008). arXiv: 0805.2620.

[CJH03] Krishnendu Chatterjee, Marcin Jurdzinski, and Thomas A. Henzinger. "Simple Stochastic Parity Games". In: *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*. 2003, pp. 100–113. DOI: 10.1007/978-3-540-45220-1_11.

[CEP95] Allan Cheng, Javier Esparza, and Jens Palsberg. "Complexity Results for 1-Safe Nets". In: *Theor. Comput. Sci.* 147.1&2 (1995), pp. 117–136. DOI: 10.1016/0304-3975(94)00231-7.

[CK93] Kwang-Ting Cheng and A. S. Krishnakumar. "Automatic Functional Test Generation Using the Extended Finite State Machine Model". In: *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*. 1993, pp. 86–91. DOI: 10.1145/157485.164585.

[CK96] Kwang-Ting Cheng and A. S. Krishnakumar. "Automatic generation of functional vectors using the extended finite state machine model". In: *ACM Trans. Design Autom. Electr. Syst.* 1.1 (1996), pp. 57–79. DOI: 10.1145/225871.225880.

[CFGR95] Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, and Marina Ribaudo. "GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets". In: *Perform. Evaluation* 24.1-2 (1995), pp. 47–68. DOI: 10.1016/0166-5316(95)00008-L.

[Cho74] Yaacov Choueka. "Theories of Automata on omega-Tapes: A Simplified Approach". In: *J. Comput. Syst. Sci.* 8.2 (1974), pp. 117–141. DOI: 10.1016/S0022-0000(74)80051-6.

[CJK97] Søren Christensen, Jens Bæk Jørgensen, and Lars Michael Kristensen. "Design/CPN - A Computer Tool for Coloured Petri Nets". In: *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*. 1997, pp. 209–223. DOI: 10.1007/BFb0035390.

[Chu57] A. Church. "Applications of recursive arithmetic to the problem of circuit synthesis". In: *Summaries of the Summer Institute of Symbolic Logic*. Vol. 1. Cornell Univ., Ithaca, NY, 1957, pp. 3–50.

[Chu62] Alonzo Church. "Logic, Arithmetic, and Automata". In: *Proceedings of International Congress of Mathematicians*. Institut Mittag-Leffler, Djursholm, 1962, pp. 23–35.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: A New Symbolic Model Checker". In: *Int. J. Softw. Tools Technol. Transf.* 2.4 (2000), pp. 410–425. DOI: 10.1007/s100090050046.

[CFIP+13] Koen Claessen, Jasmin Fisher, Samin Ishtiaq, Nir Piterman, and Qinsi Wang. "Model-Checking Signal Transduction Networks through Decreasing Reachability Sets". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 85–100. DOI: 10.1007/978-3-642-39799-8_5.

[Cla08] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking - History, Achievements, Perspectives*. 2008, pp. 1–26. DOI: 10.1007/978-3-540-69850-0_1.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. 1981, pp. 52–71. DOI: 10.1007/BFb0025774.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. 1983, pp. 117–126. DOI: 10.1145/567067.567080.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[CFJ93] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. "Exploiting Symmetry In Temporal Logic Model Checking". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 1993, pp. 450–462. DOI: 10.1007/3-540-56922-7_37.

[CGJL+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[CGJL+01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Progress on the State Explosion Problem in Model Checking". In: *Informatics - 10 Years Back. 10 Years Ahead*. 2001, pp. 176–194. DOI: 10.1007/3-540-44577-3_12.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pp. 1512–1542. DOI: 10.1145/186025.186051.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. "Introduction to Model Checking". In: *Handbook of Model Checking*. 2018, pp. 1–26. DOI: 10.1007/978-3-319-10575-8_1.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. "Completeness and Complexity of Bounded Model Checking". In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. 2004, pp. 85–96. DOI: 10.1007/978-3-540-24622-0_9.

[CV03] Edmund M. Clarke and Helmut Veith. "Counterexamples Revisited: Principles, Algorithms, Applications". In: *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*. 2003, pp. 208–224. DOI: 10.1007/978-3-540-39910-0_9.

[CW14] Edmund M. Clarke and Qinsi Wang. "2^5 Years of Model Checking". In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. 2014, pp. 26–40. DOI: 10.1007/978-3-662-46823-4_2.

[CFKM+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15.

[CDGJ+08] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008, p. 262.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27.7 (2008), pp. 1165–1178. DOI: 10.1109/TCAD.2008.923410.

[D3] D3. URL: https://d3js.org/ (visited on 03/2022).

[Dar05] P. Darondeau. "Distributed implementations of Ramadge-Wonham supervisory control with Petri nets". In: *Proceedings of the 44th IEEE Conference on Decision and Control*. 2005, pp. 2107–2112. DOI: 10.1109/CDC.2005.1582472.

[Dar07] Philippe Darondeau. "Synthesis and Control of Asynchronous and Distributed Systems". In: *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic*. 2007, pp. 13–22. DOI: 10.1109/ACSD.2007.71.

[DR12] Philippe Darondeau and S. Laurie Ricker. "Distributed Control of Discrete-Event Systems: A First Step". In: *Trans. Petri Nets Other Model. Concurr.* 6 (2012), pp. 24–45. DOI: 10.1007/978-3-642-35179-2_2.

[DJJJ+12] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jirí Srba. "TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 492–497. DOI: 10.1007/978-3-642-28756-5_36.

[DK08] Christian Dax and Felix Klaedtke. "Alternation Elimination by Complementation (Extended Abstract)". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*. 2008, pp. 214–229. DOI: 10.1007/978-3-540-89439-1_16.

[DLS06] Stéphane Demri, François Laroussinie, and Philippe Schnoebelen. "A parametric analysis of the state-explosion problem in model checking". In: *J. Comput. Syst. Sci.* 72.4 (2006), pp. 547–575. DOI: 10.1016/j.jcss.2005.11.003.

[DR96a] Jörg Desel and Wolfgang Reisig. "Place/Transition Petri Nets". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*. 1996, pp. 122–173. DOI: 10.1007/3-540-65306-6_15.

[DR96b] Jörg Desel and Wolfgang Reisig. "The Synthesis Problem of Petri Nets". In: *Acta Informatica* 33.4 (1996), pp. 297–315. DOI: 10.1007/s002360050046.

[DHJL+15] Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. "A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking". In: *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings*. 2015, pp. 35–51. DOI: 10.1007/978-3-319-25942-0_3.

[DP15] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-Core Decision Diagrams". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 677–691. DOI: 10.1007/978-3-662-46681-0_60.

[DP17] Tom van Dijk and Jaco van de Pol. "Sylvan: multi-core framework for decision diagrams". In: *Int. J. Softw. Tools Technol. Transf.* 19.6 (2017), pp. 675–696. DOI: 10.1007/s10009-016-0433-2.

[DÁBB21] Oyendrila Dobe, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. "HyperProb: A Model Checker for Probabilistic Hyperproperties". In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. 2021, pp. 657–666. DOI: 10.1007/978-3-030-90870-6_35.

[DS01] Rolf Drechsler and Detlef Sieling. "Binary decision diagrams in theory and practice". In: *Int. J. Softw. Tools Technol. Transf.* 3.2 (2001), pp. 112–136. DOI: 10.1007/s100090100056.

[EFD05] Rüdiger Ebendt, Görschwin Fey, and Rolf Drechsler. *Advanced BDD optimization*. Springer, 2005. DOI: 10.1007/b107399.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. "Efficient implementation of property directed reachability". In: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*. 2011, pp. 125–134.

[Ehl11] Rüdiger Ehlers. "Unbeast: Symbolic Bounded Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 2011, pp. 272–275. DOI: 10.1007/978-3-642-19835-9_25.

[ER16] Rüdiger Ehlers and Vasumathi Raman. “Slugs: Extensible GR(1) Synthesis”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 2016, pp. 333–339. DOI: 10.1007/978-3-319-41540-6_18.

[ESK14] Rüdiger Ehlers, Sanjit A. Seshia, and Hadas Kress-Gazit. “Synthesis with Identifiers”. In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. 2014, pp. 415–433. DOI: 10.1007/978-3-642-54013-4_23.

[ER90a] Andrzej Ehrenfeucht and Grzegorz Rozenberg. “Partial (Set) 2-Structures. Part I: Basic Notions and the Representation Problem”. In: *Acta Informatica* 27.4 (1990), pp. 315–342. DOI: 10.1007/BF00264611.

[ER90b] Andrzej Ehrenfeucht and Grzegorz Rozenberg. “Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems”. In: *Acta Informatica* 27.4 (1990), pp. 343–368. DOI: 10.1007/BF00264612.

[EGKN+04] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. “Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools”. In: *Graph Drawing Software*. 2004, pp. 127–148. DOI: 10.1007/978-3-642-18638-7_6.

[EH86] E. Allen Emerson and Joseph Y. Halpern. “"Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic”. In: *J. ACM* 33.1 (1986), pp. 151–178. DOI: 10.1145/4904.4999.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. “Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)”. In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. 1991, pp. 368–377. DOI: 10.1109/SFCS.1991.185392.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. “On Model-Checking for Fragments of μ-Calculus”. In: *Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 1993, pp. 385–396. DOI: 10.1007/3-540-56922-7_32.

[EJS01] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. “On model checking for the μ-calculus and its fragments”. In: *Theor. Comput. Sci.* 258.1-2 (2001), pp. 491–522. DOI: 10.1016/S0304-3975(00)00034-7.

[EL85] E. Allen Emerson and Chin-Laung Lei. “Modalities for Model Checking: Branching Time Strikes Back”. In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*. 1985, pp. 84–96. DOI: 10.1145/318593.318620.

[EL87] E. Allen Emerson and Chin-Laung Lei. “Modalities for Model Checking: Branching Time Logic Strikes Back”. In: *Sci. Comput. Program.* 8.3 (1987), pp. 275–306. DOI: 10.1016/0167-6423(87)90036-0.

[ES93] E. Allen Emerson and A. Prasad Sistla. "Symmetry and Model Checking". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 1993, pp. 463–478. DOI: 10.1007/3-540-56922-7_38.

[Eng91] Joost Engelfriet. "Branching Processes of Petri Nets". In: *Acta Informatica* 28.6 (1991), pp. 575–591. DOI: 10.1007/BF01463946.

[Esp96] Javier Esparza. "Decidability and Complexity of Petri Net Problems - An Introduction". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*. 1996, pp. 374–428. DOI: 10.1007/3-540-65306-6_20.

[EH08] Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008. DOI: 10.1007/978-3-540-77426-6.

[EN94] Javier Esparza and Mogens Nielsen. "Decidability Issues for Petri Nets - a survey". In: *J. Inf. Process. Cybern.* 30.3 (1994), pp. 143–160.

[EWS05] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. "Fair Simulation Relations, Parity Games, and State Space Reduction for Bu"chi Automata". In: *SIAM J. Comput.* 34.5 (2005), pp. 1159–1175. DOI: 10.1137/S0097539703420675.

[FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995. DOI: 10.7551/mitpress/5803.001.0001.

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "BoSy: An Experimentation Framework for Bounded Synthesis". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17.

[Fin15] Bernd Finkbeiner. "Bounded Synthesis for Petri Games". In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. 2015, pp. 223–237. DOI: 10.1007/978-3-319-23506-6_15.

[Fin16] Bernd Finkbeiner. "Synthesis of Reactive Systems". In: *Dependable Software Systems Engineering*. 2016, pp. 72–98. DOI: 10.3233/978-1-61499-627-9-72.

[FGHO17] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Symbolic vs. Bounded Synthesis for Petri Games". In: *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. 2017, pp. 23–43. DOI: 10.4204/EPTCS.260.5.

[FGHO19a] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Data Flows in Concurrent Network Updates". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 2019, pp. 515–533. DOI: 10.1007/978-3-030-31784-3_30.

[FGHO19b] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Data Flows in Concurrent Network Updates (Full Version)". In: *CoRR* abs/1907.11061 (2019). arXiv: 1907.11061.

[FGHO20a] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. 2020, pp. 64–76. DOI: 10.1007/978-3-030-53291-8_5.

[FGHO20b] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL (Full Version)". In: *CoRR* abs/2005.07130 (2020). arXiv: 2005.07130.

[FGHO20c] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Branching Properties on Petri Nets with Transits". In: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. 2020, pp. 394–410. DOI: 10.1007/978-3-030-59152-6_22.

[FGHO20d] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Model Checking Branching Properties on Petri Nets with Transits (Full Version)". In: *CoRR* abs/2007.07235 (2020). arXiv: 2007.07235.

[FGHO22] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. "Global Winning Conditions in Synthesis of Distributed Systems with Causal Memory". In: *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*. 2022, 20:1–20:19. DOI: 10.4230/LIPIcs.CSL.2022.20.

[FGO15] Bernd Finkbeiner, Manuel Gieseking, and Ernst-Rüdiger Olderog. "Adam: Causality-Based Synthesis of Distributed Systems". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 433–439. DOI: 10.1007/978-3-319-21690-4_25.

[FG17] Bernd Finkbeiner and Paul Gölz. "Synthesis in Distributed Environments". In: *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*. 2017, 28:1–28:14. DOI: 10.4230/LIPIcs.FSTTCS.2017.28.

[FKPS19] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. "Temporal Stream Logic: Synthesis Beyond the Bools". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. 2019, pp. 609–629. DOI: 10.1007/978-3-030-25540-4_35.

[FO14] Bernd Finkbeiner and Ernst-Rüdiger Olderog. "Petri Games: Synthesis of Distributed Systems with Causal Memory". In: *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014*. 2014, pp. 217–230. DOI: 10.4204/EPTCS.161.19.

[FO17] Bernd Finkbeiner and Ernst-Rüdiger Olderog. "Petri games: Synthesis of distributed systems with causal memory". In: *Inf. Comput.* 253 (2017), pp. 181–203. DOI: 10.1016/j.ic.2016.07.006.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. "Algorithms for Model Checking HyperLTL and HyperCTL*". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 30–48. DOI: 10.1007/978-3-319-21690-4_3.

[FS05] Bernd Finkbeiner and Sven Schewe. "Uniform Distributed Synthesis". In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. 2005, pp. 321–330. DOI: 10.1109/LICS.2005.53.

[FS13] Bernd Finkbeiner and Sven Schewe. "Bounded synthesis". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 519–539. DOI: 10.1007/s10009-012-0228-z.

[FTFO12] William M. Fitzgerald, Fatih Turkmen, Simon N. Foley, and Barry O'Sullivan. "Anomaly analysis for Physical Access Control security configuration". In: *7th International Conference on Risks and Security of Internet and Systems, CRiSIS 2012, Cork, Ireland, October 10-12, 2012*. 2012, pp. 1–8. DOI: 10.1109/CRISIS.2012.6378953.

[FA73] Michael J. Flynn and Tilak Agerwala. "Comments on Capabilities, Limitations and Correctness of Petri Nets". In: *Proceedings of the 1st Annual Symposium on Computer Architecture, Gainesville, FL, USA, December 1973*. 1973, pp. 81–86. DOI: 10.1145/800123.803973.

[FMW16] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes". In: *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*. 2016, pp. 1–9. DOI: 10.1109/IFIPNetworking.2016.7497232.

[FHFM+11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. "Frenetic: a network programming language". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 279–291. DOI: 10.1145/2034773.2034812.

[Fra86] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986. DOI: 10.1007/978-1-4612-4886-6.

[FCS11] Robert Frohardt, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. "Access Nets: Modeling Access to Physical Spaces". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. 2011, pp. 184–198. DOI: 10.1007/978-3-642-18275-4_14.

[FMY97] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 149–169. DOI: 10.1023/A:1008647823331.

[GS53] David Gale and Frank M. Stewart. "Infinite games with perfect information". In: *Contributions to the Theory of Games, Annals of Mathematics Studies* 2 (1953), pp. 245–266.

[GLZ04] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. "Distributed Games with Causal Memory Are Decidable for Series-Parallel Systems". In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*. 2004, pp. 275–286. DOI: 10.1007/978-3-540-30538-5_23.

[GO01] Paul Gastin and Denis Oddoux. "Fast LTL to Büchi Automata Translation". In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. 2001, pp. 53–65. DOI: 10.1007/3-540-44585-4_6.

[GHKF19] Gideon Geier, Philippe Heim, Felix Klein, and Bernd Finkbeiner. "Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications". In: *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. 2019, pp. 138–146. DOI: 10.23919/FMCAD.2019.8894261.

[GGMW13] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. "Asynchronous Games over Tree Architectures". In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*. 2013, pp. 275–286. DOI: 10.1007/978-3-642-39212-2_26.

[GL02] Dimitra Giannakopoulou and Flavio Lerda. "From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*. 2002, pp. 308–326. DOI: 10.1007/3-540-36135-9_20.

[GH19] Manuel Gieseking and Jesko Hecking-Harbusch. *AdamMC: A Model Checker for Petri Nets With Transits and Flow-LTL (Artifact ATVA)*. 2019. DOI: 10.6084/m9.figshare.8313344.

[GH20] Manuel Gieseking and Jesko Hecking-Harbusch. *AdamMC: A Model Checker for Petri Nets with Transits against Flow-LTL (Artifact CAV)*. 2020. DOI: 10.6084/m9.figshare.11676171.

[GHY20] Manuel Gieseking, Jesko Hecking-Harbusch, and Ann Yanich. *AdamWEB: A Web Interface for Petri Nets with Transits and Petri Games (Artifact TACAS)*. 2020. DOI: 10.6084/m9.figshare.13089800.

[GHY21] Manuel Gieseking, Jesko Hecking-Harbusch, and Ann Yanich. "A Web Interface for Petri Nets with Transits and Petri Games". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. 2021, pp. 381–388. DOI: 10.1007/978-3-030-72013-1_22.

[GO21] Manuel Gieseking and Ernst-Rüdiger Olderog. "High-Level Representation of Benchmark Families for Petri Games". In: *Model Checking, Synthesis, and Learning - Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday*. 2021, pp. 115–137. DOI: 10.1007/978-3-030-91384-7_7.

[GOW20] Manuel Gieseking, Ernst-Rüdiger Olderog, and Nick Würdemann. "Solving high-level Petri games". In: *Acta Informatica* 57.3-5 (2020), pp. 591–626. DOI: 10.1007/s00236-020-00368-5.

[GW20] Manuel Gieseking and Nick Würdemann. *Canonical Representations for Direct Generation of Strategies in High-level Petri Games (Artifact ICATPN)*. 2020. DOI: 10.6084/m9.figshare.13697845.

[GW21a] Manuel Gieseking and Nick Würdemann. "Canonical Representations for Direct Generation of Strategies in High-Level Petri Games". In: *Application and Theory of Petri Nets and Concurrency - 42nd International Conference, PETRI NETS 2021, Virtual Event, June 23-25, 2021, Proceedings*. 2021, pp. 95–117. DOI: 10.1007/978-3-030-76983-3_6.

[GW21b] Manuel Gieseking and Nick Würdemann. "Canonical Representations for Direct Generation of Strategies in High-level Petri Games (Full Version)". In: *CoRR* abs/2103.10207 (2021). arXiv: 2103.10207.

[Gim17] Hugo Gimbert. "On the Control of Asynchronous Automata". In: *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*. 2017, 30:1–30:15. DOI: 10.4230/LIPIcs.FSTTCS.2017.30.

[GCH13] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. "Synthesis of AMBA AHB from formal specification: a case study". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 585–601. DOI: 10.1007/s10009-011-0207-9.

[Göl17] Paul Gölz. "Synthesis for Petri Games with One System Player". Bachelor's Thesis. Saarland University, Saarbrücken, Germany, 2017.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-36387-4.

[Han19] Paul Hannibal. "Entscheidbarkeit von Petri-Spielen mit 2 Umgebungs- und 2 System-Spielern". German. Master's Thesis. University of Oldenburg, Oldenburg, Germany, 2019.

[HP84] David Harel and Amir Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*. 1984, pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17.

[ETVV17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. "Network-Wide Configuration Synthesis". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 2017, pp. 261–281. DOI: 10.1007/978-3-319-63390-9_14.

[HP00] Klaus Havelund and Thomas Pressburger. "Model Checking JAVA Programs using JAVA PathFinder". In: *Int. J. Softw. Tools Technol. Transf.* 2.4 (2000), pp. 366–381. DOI: 10.1007/s100090050043.

[HS96] Klaus Havelund and Natarajan Shankar. "Experiments in Theorem Proving and Model Checking for Protocol Verification". In: *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*. 1996, pp. 662–681. DOI: 10.1007/3-540-60973-3_113.

[HS99] Klaus Havelund and Jens U. Skakkebæk. "Applying Model Checking in Java Verification". In: *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*. 1999, pp. 216–231. DOI: 10.1007/3-540-48234-2_17.

[HHFM+94] Jifeng He, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. "Provably Correct Systems". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS, Lübeck, Germany, September 19-23, Proceedings*. 1994, pp. 288–335. DOI: 10.1007/3-540-58468-4_171.

[Hec21] Jesko Hecking-Harbusch. "Synthesis of asynchronous distributed systems from global specifications". PhD thesis. Saarland University, Saarbrücken, Germany, 2021.

[HM19] Jesko Hecking-Harbusch and Niklas O. Metzger. "Efficient Trace Encodings of Bounded Synthesis for Asynchronous Distributed Systems". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 2019, pp. 369–386. DOI: 10.1007/978-3-030-31784-3_22.

[HT18] Jesko Hecking-Harbusch and Leander Tentrup. "Solving QBF by Abstraction". In: *Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018*. 2018, pp. 88–102. DOI: 10.4204/EPTCS.277.7.

[Hel99a] Keijo Heljanko. "Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 1999, pp. 240–254. DOI: 10.1007/3-540-49059-0_17.

[Hel99b] Keijo Heljanko. "Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets". In: *Fundam. Informaticae* 37.3 (1999), pp. 247–268. DOI: 10.3233/FI-1999-37304.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Software Verification with BLAST". In: *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*. 2003, pp. 235–239. DOI: 10.1007/3-540-44829-2_17.

[HKG97] Lawrence E. Holloway, Bruce H. Krogh, and Alessandro Giua. "A Survey of Petri Net Methods for Controlled Discrete Event Systems". In: *Discret. Event Dyn. Syst.* 7.2 (1997), pp. 151–190. DOI: 10.1023/A:1008271916548.

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[Hol05] Gerard J. Holzmann. "Software model checking with SPIN". In: *Adv. Comput.* 65 (2005), pp. 78–109. DOI: 10.1016/S0065-2458(05)65002-4.

[ID93] C. Norris Ip and David L. Dill. "Better Verification Through Symmetry". In: *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*. 1993, pp. 97–111.

[Iri13] Nazario Irizarry. *Mixing C and Java for high performance computing*. Tech. rep. MITRE Corporation, 2013.

[JVCS07] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-based software testing and analysis with C*. Cambridge University Press, 2007.

[JBCF+19] Swen Jacobs, Roderick Bloem, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Philipp J. Meyer, Thibaud Michaud, Mouhammad Sakr, Salomon Sickert, Leander Tentrup, and Adam Walker. "The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results". In: *CoRR* abs/1904.07736 (2019). arXiv: 1904.07736.

[Jen81] Kurt Jensen. "Coloured Petri Nets and the Invariant-Method". In: *Theor. Comput. Sci.* 14 (1981), pp. 317–336. DOI: 10.1016/0304-3975(81)90049-9.

[Jen97] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 3*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1997. DOI: 10.1007/978-3-642-60794-3.

[Jen] Kurt Jensen. *Design/CPN*. Computer Science Department, University of Aarhus, Denmark.

[JSUV22] Peter Gjøl Jensen, Jirí Srba, Nikolaj Jensen Ulrik, and Simon Mejlby Virenfeldt. "Automata-Driven Partial Order Reduction and Guided Search for LTL Model Checking". In: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*. 2022, pp. 151–173. DOI: 10.1007/978-3-030-94583-1_8.

[Jen02] Rune M. Jensen. *A Comparison Study between the CUDD and BuDDy OBDD Package Applied to AI-Planning Problems*. Tech. rep. CMU-CS-02-173. School of Computer Science, Carnegie Mellon University, 2002.

[JLGK+14] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. "Dynamic scheduling of network updates". In: *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*. 2014, pp. 539–550. DOI: 10.1145/2619239.2626307.

[JGWB07] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. "Anzu: A Tool for Property Synthesis". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, pp. 258–262. DOI: 10.1007/978-3-540-73368-3_29.

[Jr78] Sheldon B. Akers Jr. "Binary Decision Diagrams". In: *IEEE Trans. Computers* 27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141.

[Jur98] Marcin Jurdzinski. "Deciding the Winner in Parity Games is in UP ∩ co-Up". In: *Inf. Process. Lett.* 68.3 (1998), pp. 119–124. DOI: 10.1016/S0020-0190(98)00150-1.

[Jur11] Marcin Jurdzinski. "Algorithms for solving parity games". In: *Lectures in Game Theory for Computer Scientists*. 2011, pp. 74–98.

[Kah07] Reinhard Kahle. "Freek Wiedijk (Ed.), The Seventeen Provers of the World". In: *Stud Logica* 87.2-3 (2007), pp. 369–374. DOI: 10.1007/s11225-007-9093-2.

[Kam15] E Kamp. *Bandwidth, profile and wavefront reduction for static variable ordering in symbolic model checking*. Tech. rep. University of Twente, 2015.

[KRW13] Naga Praveen Katta, Jennifer Rexford, and David Walker. "Incremental consistent updates". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*. 2013, pp. 49–54. DOI: 10.1145/2491185.2491191.

[KPR98] Yonit Kesten, Amir Pnueli, and Li-on Raviv. "Algorithmic Verification of Linear Temporal Logic Specifications". In: *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. 1998, pp. 1–16. DOI: 10.1007/BFb0055036.

[KPRS06] Yonit Kesten, Amir Pnueli, Li-on Raviv, and Elad Shahar. "Model Checking with Strong Fairness". In: *Formal Methods Syst. Des.* 28.1 (2006), pp. 57–84. DOI: 10.1007/s10703-006-4342-y.

[KB17] Ayrat Khalimov and Roderick Bloem. "Bounded Synthesis for Streett, Rabin, and CTL*". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 2017, pp. 333–352. DOI: 10.1007/978-3-319-63390-9_18.

[KKV03] Victor Khomenko, Maciej Koutny, and Walter Vogler. "Canonical prefixes of Petri net unfoldings". In: *Acta Informatica* 40.2 (2003), pp. 95–118. DOI: 10.1007/s00236-003-0122-y.

[KN01] Bakhadyr Khoussainov and Anil Nerode. *Automata theory and its applications*. Vol. 21. Birkhäuser Basel, 2001.

[Kin76] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[Kla91] Nils Klarlund. "Progress Measures for Complementation of omega-Automata with Applications to Temporal Logic". In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. 1991, pp. 358–367. DOI: 10.1109/SFCS.1991.185391.

[Kla01] Hartmut Klauck. "Algorithms for Parity Games". In: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. 2001, pp. 107–129. DOI: 10.1007/3-540-36387-4_7.

[KNFB+11] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. "The Internet Topology Zoo". In: *IEEE J. Sel. Areas Commun.* 29.9 (2011), pp. 1765–1775. DOI: 10.1109/JSAC.2011.111002.

[KV21] Michalis Kokologiannakis and Viktor Vafeiadis. "GenMC: A Model Checker for Weak Memory Models". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.

[KW19] Jürgen König and Heike Wehrheim. "Data Independence for Software Transactional Memory". In: *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. 2019, pp. 263–279. DOI: 10.1007/978-3-030-20652-9_18.

[KBGH+21] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, J. Srba, Y. Thierry-Mieg, A. Walner, and K. Wolf. *Complete Results for the 2020 Edition of the Model Checking Contest.* June 2021. URL: http://mcc.lip6.fr/2021/results.php (visited on 03/2022).

[Koz83] Dexter Kozen. "Results on the Propositional mu-Calculus". In: *Theor. Comput. Sci.* 27 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6.

[KRVR+15] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proc. IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[Kri59] Saul Kripke. "A Completeness Theorem in Modal Logic". In: *J. Symb. Log.* 24.1 (1959), pp. 1–14. DOI: 10.2307/2964568.

[KV98] Lars Michael Kristensen and Antti Valmari. "Finding Stubborn Sets of Coloured Petri Nets Without Unfolding". In: *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN '98, Lisbon, Portugal, June 22-26, 1998, Proceedings.* 1998, pp. 104–123. DOI: 10.1007/3-540-69108-1_7.

[Kup18] Orna Kupferman. "Automata Theory and Model Checking". In: *Handbook of Model Checking.* 2018, pp. 107–151. DOI: 10.1007/978-3-319-10575-8_4.

[KV00a] Orna Kupferman and Moshe Y. Vardi. "μ-Calculus Synthesis". In: *Mathematical Foundations of Computer Science 2000, 25th International Symposium, MFCS 2000, Bratislava, Slovakia, August 28 - September 1, 2000, Proceedings.* 2000, pp. 497–507. DOI: 10.1007/3-540-44612-5_45.

[KV00b] Orna Kupferman and Moshe Y. Vardi. "Synthesis with Incomplete Informatio". In: *2nd International Conference on Temporal Logic (ICTL'97), Advances in Temporal Logic.* 2000, pp. 109–127. DOI: 10.1007/978-94-015-9586-5_6.

[KV01a] Orna Kupferman and Moshe Y. Vardi. "Synthesizing Distributed Systems". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings.* 2001, pp. 389–398. DOI: 10.1109/LICS.2001.932514.

[KV01b] Orna Kupferman and Moshe Y. Vardi. "Weak alternating automata are not that weak". In: *ACM Trans. Comput. Log.* 2.3 (2001), pp. 408–429. DOI: 10.1145/377978.377993.

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. "An automata-theoretic approach to branching-time model checking". In: *J. ACM* 47.2 (2000), pp. 312–360. DOI: 10.1145/333979.333987.

[Kur14] Robert P Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Vol. 302. Princeton university press, 2014.

[Küs01] Ralf Küsters. "Memoryless Determinacy of Parity Games". In: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. 2001, pp. 95–106. DOI: 10.1007/3-540-36387-4_6.

[LH00] Timo Latvala and Keijo Heljanko. "Coping With Strong Fairness". In: *Fundam. Informaticae* 43.1-4 (2000), pp. 175–193. DOI: 10.3233/FI-2000-43123409.

[Lee59] Chang-Yeong Lee. "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[LP85] Orna Lichtenstein and Amir Pnueli. "Checking That Finite State Concurrent Programs Satisfy Their Linear Specification". In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*. 1985, pp. 97–107. DOI: 10.1145/318593.318622.

[Lin] Jørn Lind-Nielsen. *BuDDy: Binary Decision Diagram package*. Department of Information Technology, Technical University of Denmark. URL: http://sourceforge.net/projects/buddy/ (visited on 03/2022).

[LMS14] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. "A Thread-Safe Library for Binary Decision Diagrams". In: *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*. 2014, pp. 35–49. DOI: 10.1007/978-3-319-10431-7_4.

[LSX13] Guanfeng Lv, Kaile Su, and Yanyan Xu. "CacBDD: A BDD Package with Dynamic Cache Management". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 229–234. DOI: 10.1007/978-3-642-39799-8_15.

[MT01] P. Madhusudan and P. S. Thiagarajan. "Distributed Controller Synthesis for Local Specifications". In: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*. 2001, pp. 396–407. DOI: 10.1007/3-540-48224-5_33.

[MT02] P. Madhusudan and P. S. Thiagarajan. "A Decidable Class of Asynchronous Distributed Controllers". In: *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*. 2002, pp. 145–160. DOI: 10.1007/3-540-45694-5_11.

[MTY05] P. Madhusudan, P. S. Thiagarajan, and Shaofa Yang. "The MSO Theory of Connectedly Communicating Processes". In: *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*. 2005, pp. 201–212. DOI: 10.1007/11590156_16.

[MKAC+11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. "Debugging the data plane with anteater". In: *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. 2011, pp. 290–301. DOI: 10.1145/2018436.2018470.

[MR18] Rupak Majumdar and Jean-François Raskin. "Symbolic Model Checking in Non-Boolean Domains". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 1111–1147. DOI: 10.1007/978-3-319-10575-8_31.

[MTW14] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. "Kuai: A model checker for software-defined networks". In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. 2014, pp. 163–170. DOI: 10.1109/FMCAD.2014.6987609.

[MILY+21] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark W. Barrett. "Pono: A Flexible and Extensible SMT-Based Model Checker". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. 2021, pp. 461–474. DOI: 10.1007/978-3-030-81688-9_22.

[MW81] Zohar Manna and Pierre Wolper. "Synthesis of Communicating Processes from Temporal Logic Specifications". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. 1981, pp. 253–281. DOI: 10.1007/BFb0025786.

[MHC17] Jedidiah McClurg, Hossein Hojjat, and Pavol Cerný. "Synchronization Synthesis for Network Programs". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 2017, pp. 301–321. DOI: 10.1007/978-3-319-63390-9_16.

[MABP+08] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. "OpenFlow: enabling innovation in campus networks". In: *Comput. Commun. Rev.* 38.2 (2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[McM92] Kenneth L. McMillan. "Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits". In: *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*. 1992, pp. 164–177. DOI: 10.1007/3-540-56496-9_14.

[McM93] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993. DOI: 10.1007/978-1-4615-3190-6.

[McM95] Kenneth L. McMillan. "A Technique of State Space Search Based on Unfolding". In: *Formal Methods Syst. Des.* 6.1 (1995), pp. 45–65. DOI: 10.1007/BF01384314.

[McM03] Kenneth L. McMillan. "Craig Interpolation and Reachability Analysis". In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 2003, p. 336. DOI: 10.1007/3-540-44898-5_18.

[McN65] Robert McNaughton. *Finite-state infinite games*. Project MAC Rep. MIT, Cambridge, Mass, 1965.

[McN93] Robert McNaughton. "Infinite Games Played on Finite Graphs". In: *Ann. Pure Appl. Logic* 65.2 (1993), pp. 149–184. DOI: 10.1016/0168-0072(93)90036-D.

[MP15] Jeroen Meijer and Jaco van de Pol. "Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Model Checking". In: *CoRR* abs/1511.08678 (2015). arXiv: 1511.08678.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithmen und Datenstrukturen im VLSI-Design: OBDD - Grundlagen und Anwendungen*. Springer, 1998.

[Meo12] Robert Meolic. "Biddy - a Multi-platform Academic BDD Package". In: *J. Softw.* 7.6 (2012), pp. 1358–1366. DOI: 10.4304/jsw.7.6.1358-1366.

[Met17] Niklas Metzger. "Bounded Synthesis of Petri Games with True Concurrency Semantics". Bachelor's Thesis. Saarland University, Saarbrücken, Germany, 2017.

[MSL18] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31.

[MC18] Thibaud Michaud and Maximilien Colange. "Reactive synthesis from LTL specification with Spot". In: *7th Workshop on Synthesis, SYNT@CAV*. 2018.

[Mic88] Max Michel. "Complementation is more difficult with automata on infinite words". In: *CNET, Paris* 15 (1988).

[Mis12] Dirk Missal. "Formal synthesis of safety controller code for distributed controllers". PhD thesis. Martin Luther University of Halle-Wittenberg, 2012.

[MH08] Dirk Missal and Hans-Michael Hanisch. "A Modular Synthesis Approach for Distributed Safety Controllers, Part A: Modelling and Specification". In: *IFAC Proceedings Volumes* 41.2 (2008). 17th IFAC World Congress, pp. 14473–14478.

[MH84] Satoru Miyano and Takeshi Hayashi. "Alternating Finite Automata on omega-Words". In: *Theor. Comput. Sci.* 32 (1984), pp. 321–330. DOI: 10.1016/0304-3975(84)90049-5.

[MRFR+13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. "Composing Software Defined Networks". In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. 2013, pp. 1–13.

[Mor82] Bernard M. E. Moret. "Decision Trees and Diagrams". In: *ACM Comput. Surv.* 14.4 (1982), pp. 593–623. DOI: 10.1145/356893.356898.

[Mos91] Andrzej Wlodzimierz Mostowski. *Games with Forbidden Positions*. Tech. rep. 78. Instytut Matematyki, Uniwersytet Gda'nski, Poland, 1991.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. "Weak Alternating Automata Give a Simple Explanation of Why Most Temporal and Dynamic Logics are Decidable in Exponential Time". In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 1988, pp. 422–427. DOI: 10.1109/LICS.1988.5139.

[MS87] David E. Muller and Paul E. Schupp. "Alternating Automata on Infinite Trees". In: *Theor. Comput. Sci.* 54 (1987), pp. 267–276. DOI: 10.1016/0304-3975(87)90133-2.

[Mur89] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143.

[MW14] Anca Muscholl and Igor Walukiewicz. "Distributed Synthesis for Acyclic Architectures". In: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*. 2014, pp. 639–651. DOI: 10.4230/LIPIcs.FSTTCS.2014.639.

[MPCE+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. "CMC: A Pragmatic Approach to Model Checking Real Code". In: *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. 2002.

[Mye04] Glenford J. Myers. *The art of software testing (2. ed.)* Wiley, 2004.

[NM10] Ukachukwu Ndukwu and Annabelle McIver. "An expectation transformer approach to predicate abstraction and data independence for probabilistic programs". In: *Proceedings Eighth Workshop on Quantitative Aspects of Programming Languages, QAPL 2010, Paphos, Cyprus, 27-28th March 2010*. 2010, pp. 129–143. DOI: 10.4204/EPTCS.28.9.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. "Petri Nets, Event Structures and Domains, Part I". In: *Theor. Comput. Sci.* 13 (1981), pp. 85–108. DOI: 10.1016/0304-3975(81)90112-2.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-45949-9.

[OR93] Ernst-Rüdiger Olderog and Stephan Rössig. "A Case Study in Transformational Design of Concurrent Systems". In: *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*. 1993, pp. 90–104. DOI: 10.1007/3-540-56610-4_58.

[PIKL+15] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. "Decentralizing SDN Policies". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 663–676. DOI: 10.1145/2676726.2676990.

[Pan21] Lukas Panneke. "Optimization Techniques for Reachability Analysis of Bounded Petri Nets". Implementation: https://github.com/Selebrator/bachelor. Bachelor's Thesis. University of Oldenburg, Oldenburg, Germany, 2021.

[Pel18] Doron Peled. "Partial-Order Reduction". In: *Handbook of Model Checking*. 2018, pp. 173–190. DOI: 10.1007/978-3-319-10575-8_6.

[PP04] Dominique Perrin and Jean-Eric Pin. *Infinite words - automata, semigroups, logic and games*. Vol. 141. Pure and applied mathematics series. Elsevier Morgan Kaufmann, 2004.

[Pet80] James L. Peterson. "A Note on Colored Petri Nets". In: *Inf. Process. Lett.* 11.1 (1980), pp. 40–43. DOI: 10.1016/0020-0190(80)90032-0.

[Pet62] Carl Adam Petri. "Kommunikation mit Automaten". PhD thesis. Bonn: Institut für instrumentelle Mathematik, 1962.

[Pit06] Nir Piterman. "From Nondeterministic Buchi and Streett Automata to Deterministic Parity Automata". In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. 2006, pp. 255–264. DOI: 10.1109/LICS.2006.28.

[Pit07] Nir Piterman. "From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata". In: *Log. Methods Comput. Sci.* 3.3 (2007). DOI: 10.2168/LMCS-3(3:5)2007.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) Designs". In: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. 2006, pp. 364–380. DOI: 10.1007/11609773_24.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PR89a] Amir Pnueli and Roni Rosner. "On the Synthesis of a Reactive Module". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 1989, pp. 179–190. DOI: 10.1145/75277.75293.

[PR89b] Amir Pnueli and Roni Rosner. "On the Synthesis of an Asynchronous Reactive Module". In: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. 1989, pp. 652–671. DOI: 10.1007/BFb0035790.

[PR90] Amir Pnueli and Roni Rosner. "Distributed Reactive Systems Are Hard to Synthesize". In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597.

[PLP11] Richard Pohl, Kim Lauenroth, and Klaus Pohl. "A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models". In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. 2011, pp. 313–322. DOI: 10.1109/ASE.2011.6100068.

[PB15] Mihai-Lica Pura and Didier Buchs. "Symbolic Model Checking of Security Protocols for Ad hoc Networks on any Topologies". In: *Trans. Petri Nets Other Model. Concurr.* 10 (2015), pp. 109–130. DOI: 10.1007/978-3-662-48650-4_6.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22.

[Rab69] Michael O Rabin. "Decidability of second-order theories and automata on infinite trees". In: *Transactions of the american Mathematical Society* 141 (1969), pp. 1–35.

[RS59] Michael O. Rabin and Dana S. Scott. "Finite Automata and Their Decision Problems". In: *IBM J. Res. Dev.* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114.

[RW89] Peter JG Ramadge and W Murray Wonham. "The control of discrete event systems". In: *Proceedings of the IEEE* 77.1 (1989), pp. 81–98.

[Rei98] Wolfgang Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer, 1998.

[Rei13] Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. DOI: 10.1007/978-3-642-33278-4.

[RFRS+12] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. "Abstractions for network update". In: *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*. 2012, pp. 323–334. DOI: 10.1145/2342356.2342427.

[RXG00] Nidhal Rezg, Xiaolan Xie, and Asma Ghaffari. "Supervisory Control in Discrete Event Systems Using the Theory of Regions". In: *Discrete Event Systems: Analysis and Control*. Ed. by R. Boel and G. Stremersch. Boston, MA: Springer US, 2000, pp. 391–398. DOI: 10.1007/978-1-4615-4493-7_41.

[Ric53] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[RK08] Michael Rice and Sanjay Kulhari. *A survey of static variable ordering heuristics for efficient BDD/MDD construction*. Tech. rep. University of California, 2008.

[RZS09] Andrei Rimsa, Luis Enrique Zárate, and Mark A. J. Song. "Evaluation of Different BDD Libraries to Extract Concepts in FCA - Perspectives and Limitations". In: *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*. 2009, pp. 367–376. DOI: 10.1007/978-3-642-01970-8_36.

[RSS21] Mauricio Rocha, Adenilso Simão, and Thiago Sousa. "Model-based test case generation from UML sequence diagrams using extended finite state machines". In: *Softw. Qual. J.* 29.3 (2021), pp. 597–627. DOI: 10.1007/s11219-020-09531-0.

[RB99] A. W. Roscoe and Philippa J. Broadfoot. "Proving Security Protocols with Model Checkers by Data Independence Techniques". In: *J. Comput. Secur.* 7.1 (1999), pp. 147–190.

[Roz16] Kristin Yvonne Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy". In: *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*. 2016, pp. 8–26. DOI: 10.1007/978-3-319-48869-1_2.

[RW92] K. Rudie and W.M. Wonham. "Think globally, act locally: decentralized supervisory control". In: *IEEE Transactions on Automatic Control* 37.11 (1992), pp. 1692–1708. DOI: 10.1109/9.173140.

[Saf88] Shmuel Safra. "On the Complexity of omega-Automata". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948.

[Sch08] Sven Schewe. "Synthesis of distributed systems". PhD thesis. Saarland University, Saarbrücken, Germany, 2008.

[Sch09] Sven Schewe. "Tighter Bounds for the Determinisation of Büchi Automata". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 167–181. DOI: 10.1007/978-3-642-00596-1_13.

[SF07] Sven Schewe and Bernd Finkbeiner. "Bounded Synthesis". In: *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*. 2007, pp. 474–488. DOI: 10.1007/978-3-540-75596-8_33.

[SV14] Sven Schewe and Thomas Varghese. "Determinising Parity Automata". In: *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*. 2014, pp. 486–498. DOI: 10.1007/978-3-662-44522-8_41.

[Sch00] Karsten Schmidt. “LoLA: A Low Level Analyser”. In: *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding*. 2000, pp. 465–474. DOI: 10.1007/3-540-44988-4_27.

[Sch19] Sanny Schmitt. “Generating Concurrency-preserving Petri Games”. Bachelor’s Thesis. Saarland University, Saarbrücken, Germany, 2019.

[Sch02] Philippe Schnoebelen. “The Complexity of Temporal Logic Model Checking”. In: *Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse, France, 30 September - 2 October 2002*. 2002, pp. 393–436.

[ST03] Roberto Sebastiani and Stefano Tonetta. “"More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking”. In: *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L’Aquila, Italy, October 21-24, 2003, Proceedings*. 2003, pp. 126–140. DOI: 10.1007/978-3-540-39724-3_12.

[SYGA+14] Ohad Shacham, Eran Yahav, Guy Golan-Gueta, Alex Aiken, Nathan Grasso Bronson, Mooly Sagiv, and Martin T. Vechev. “Verifying atomicity via data independence”. In: *International Symposium on Software Testing and Analysis, ISSTA ’14, San Jose, CA, USA - July 21 - 26, 2014*. 2014, pp. 26–36. DOI: 10.1145/2610384.2610402.

[Sie98] Detlef Sieling. “On the Existence of Polynomial Time Approximation Schemes for OBDD Minimization”. In: *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings*. 1998, pp. 205–215. DOI: 10.1007/BFb0028562.

[SC85] A. Prasad Sistla and Edmund M. Clarke. “The Complexity of Propositional Linear Temporal Logics”. In: *J. ACM* 32.3 (1985), pp. 733–749. DOI: 10.1145/3828.3837.

[SS09] Saqib Sohail and Fabio Somenzi. “Safety first: A two-stage algorithm for LTL games”. In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. 2009, pp. 77–84. DOI: 10.1109/FMCAD.2009.5351138.

[SSR08] Saqib Sohail, Fabio Somenzi, and Kavita Ravi. “A Hybrid Algorithm for LTL Games”. In: *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*. 2008, pp. 309–323. DOI: 10.1007/978-3-540-78163-9_26.

[Som] Fabio Somenzi. *CUDD: Colorado University Decision Diagram Package*. Department of Electrical and Computer Engineering, University of Colorado, Boulder. URL: https://github.com/ivmai/cudd (visited on 03/2022).

[SB00] Fabio Somenzi and Roderick Bloem. “Efficient Büchi Automata from LTL Formulae”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 248–263. DOI: 10.1007/10722167_21.

[Spa] Sparkjava. URL: http://sparkjava.com/ (visited on 03/2022).

[Spr15] Valentin Spreckels. "Petri-Spiele: Erweiterung der Sicherheitsgewinnbedingung auf Markierungen". German. Master's Thesis. University of Oldenburg, Oldenburg, Germany, 2015.

[Sta85] Eugene W. Stark. "A Proof Technique for Rely/Guarantee Properties". In: *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*. 1985, pp. 369–391. DOI: 10.1007/3-540-16042-6_21.

[Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Leitfäden und Monographien der Informatik. Teubner, 1990.

[Sto74] Larry Joseph Stockmeyer. "The complexity of decision problems in automata theory and logic." PhD thesis. Massachusetts Institute of Technology, 1974.

[TOHT06] Takashi Takenaka, Kozo Okano, Teruo Higashino, and Kenichi Taniguchi. "Symbolic model checking of extended finite state machines with linear constraints over integer variables". In: *Systems and Computers in Japan* 37.6 (2006), pp. 64–72. DOI: 10.1002/scj.20264.

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.

[Ten16] Leander Tentrup. "Non-prenex QBF Solving Using Abstraction". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 393–401. DOI: 10.1007/978-3-319-40970-2_24.

[TH96] P. S. Thiagarajan and Jesper G. Henriksen. "Distributed Versions of Linear Time Temporal Logic: A Trace Perspective". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*. 1996, pp. 643–681. DOI: 10.1007/3-540-65306-6_24.

[Thi15] Yann Thierry-Mieg. "Symbolic Model-Checking Using ITS-Tools". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 231–237. DOI: 10.1007/978-3-662-46681-0_20.

[Thi20] Yann Thierry-Mieg. "Structural Reductions Revisited". In: *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020, Paris, France, June 24-25, 2020, Proceedings*. 2020, pp. 303–323. DOI: 10.1007/978-3-030-51831-8_15.

[Tho90] Wolfgang Thomas. "Automata on Infinite Objects". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. 1990, pp. 133–191. DOI: 10.1016/b978-0-444-88074-1.50009-3.

[Tho97] Wolfgang Thomas. "Languages, Automata, and Logic". In: *Handbook of Formal Languages, Volume 3: Beyond Words*. 1997, pp. 389–455. DOI: 10.1007/978-3-642-59126-6_7.

[Tho02] Wolfgang Thomas. "Infinite Games and Verification (Extended Abstract of a Tutorial)". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. 2002, pp. 58–64. DOI: 10.1007/3-540-45657-0_5.

[Tho09] Wolfgang Thomas. "Facets of Synthesis: Revisiting Church's Problem". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 1–14. DOI: 10.1007/978-3-642-00596-1_1.

[Tiu94] Mikko Tiusanen. "Symbolic, Symmetry, and Stubborn Set Searches". In: *Application and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain, June 20-24, 1994, Proceedings*. 1994, pp. 511–530. DOI: 10.1007/3-540-58152-9_28.

[TDB16] Petar Tsankov, Mohammad Torabi Dashti, and David A. Basin. "Access Control Synthesis for Physical Spaces". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 443–457. DOI: 10.1109/CSF.2016.38.

[Tur37] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

[Uni12] University of Oldenburg. *APT – Analyse von Petri-Netzen und Transitionssystemen*. https://github.com/CvO-Theory/apt. 2012.

[Vah] Arash Vahidi. *JDD: A pure Java BDD and Z-BDD library*. URL: https://bitbucket.org/vahidi/jdd (visited on 03/2022).

[Val89] Antti Valmari. "Stubborn sets for reduced state space generation". In: *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*. 1989, pp. 491–515. DOI: 10.1007/3-540-53863-1_36.

[Val90] Antti Valmari. "A Stubborn Attack On State Explosion". In: *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*. 1990, pp. 156–165. DOI: 10.1007/BFb0023729.

[Val92] Antti Valmari. "A Stubborn Attack on State Explosion". In: *Formal Methods Syst. Des.* 1.4 (1992), pp. 297–322. DOI: 10.1007/BF00709154.

[Var95a] Moshe Y. Vardi. "Alternating Automata and Program Verification". In: *Computer Science Today: Recent Trends and Developments*. 1995, pp. 471–485. DOI: 10.1007/BFb0015261.

[Var95b] Moshe Y. Vardi. "An Automata-Theoretic Approach to Fair Realizability and Synthesis". In: *Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings*. 1995, pp. 267–278. DOI: 10.1007/3-540-60045-0_56.

[VW08] Moshe Y. Vardi and Thomas Wilke. "Automata: from logics to algorithms". In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. 2008, pp. 629–736.

[VW86] Moshe Y. Vardi and Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. 1986, pp. 332–344.

[VW94] Moshe Y. Vardi and Pierre Wolper. "Reasoning About Infinite Computations". In: *Inf. Comput.* 115.1 (1994), pp. 1–37. DOI: 10.1006/inco.1994.1092.

[Vuea] Vue.js. URL: https://vuejs.org/ (visited on 03/2022).

[Vueb] Vuetify. URL: https://vuetifyjs.com/ (visited on 03/2022).

[WTD16] Neil Walkinshaw, Ramsay Taylor, and John Derrick. "Inferring extended finite state machine models from software executions". In: *Empir. Softw. Eng.* 21.3 (2016), pp. 811–853. DOI: 10.1007/s10664-015-9367-7.

[WMLT+13] Anduo Wang, Salar Moarref, Boon Thau Loo, Ufuk Topcu, and Andre Scedrov. "Automated synthesis of reactive controllers for software-defined networks". In: *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*. 2013, pp. 1–6. DOI: 10.1109/ICNP.2013.6733666.

[Wha] John Whaley. *JavaBDD: Java binary decision diagram library*. URL: http://javabdd.sourceforge.net/ (visited on 03/2022).

[Wol18] Karsten Wolf. "Petri Net Model Checking with LoLA 2". In: *Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings*. 2018, pp. 351–362. DOI: 10.1007/978-3-319-91268-4_18.

[Wol86] Pierre Wolper. "Expressing Interesting Properties of Programs in Propositional Temporal Logic". In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 1986, pp. 184–193. DOI: 10.1145/512644.512661.

[Wür21] Nick Würdemann. "Exploiting symmetries of high-level Petri games in distributed synthesis". In: *it Inf. Technol.* 63.5-6 (2021), pp. 321–331. DOI: 10.1515/itit-2021-0012.

[YL02] Tae-Sic Yoo and Stéphane Lafortune. "A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems". In: *Discret. Event Dyn. Syst.* 12.3 (2002), pp. 335–377. DOI: 10.1023/A:1015625600613.

[Zie87] Wieslaw Zielonka. "Notes on Finite Asynchronous Automata". In: *RAIRO Theor. Informatics Appl.* 21.2 (1987), pp. 99–135. DOI: 10.1051/ita/1987210200991.

[Zie98] Wieslaw Zielonka. "Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees". In: *Theor. Comput. Sci.* 200.1-2 (1998), pp. 135–183. DOI: 10.1016/S0304-3975(98)00009-7.

Symbols

General

Model Checking

Synthesis

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg befolgt habe.

Oldenburg, den 23.03.2022