Abstraction-Based Verification of Parameterized Networks

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Kai Baukus

Kiel

März 2003

1. Gutachter	Prof. Dr. Yassine Lakhnech
2. Gutachter	Prof. Dr. Willem-Paul de Roever
Datum der mündlichen Prüfung	28. Mai 2003

Contents

Preface

At the end of writing this thesis I notice that all this work would not have been possible without the help of many people. They have paved my way from the beginning of my computer science studies up to the end of my doctorate. First of all I thank Willem-Paul de Roever. It was him who awakened my interest in problems connected to concurrent and distributed computing. In his undergraduate course on theoretical computer science, lectured jointly with Wolfgang Thomas, I understood for the first time that the correctness of software is primarily a mathematical problem. In his graduate courses on the verification of concurrent programs I learned, with the help of Yassine Lakhnech, how to solve such mathematical problems with mathematical accuracy. My diploma thesis on the generation of auxiliary invariants, supervised by Yassine Lakhnech, got me involved in the verification of infinite state systems. Having finished my thesis and getting employed as research assistant at the chair of Willem-Paul this has remained my field of interest. Willem-Paul encourages his assistants to attend workshops and conferences, national and international, to get access to the verification community, and to build up one's own research network.

During a research visit of Saddek Bensalem to Kiel, Yassine proposed to verify parameterized systems with abstraction-based techniques. Inspired by the work of [ABJN99] we started together with Saddek to develop an abstraction-based approach which finally resulted in this thesis. During my visits to Grenoble, Saddek has always been a helpful contact. Yassine taught me how to work scientifically, how to formalize problems occurring during the work, how to tackle them, and how to present one's own ideas and solutions in papers or talks. Together with Karsten Stahl I have tried to substantiate Yassine's advice. We have published several papers which are partly the basis of this work. I thank Karsten for this fruitful collaboration.

Besides Karsten, I thank all my colleagues for an enjoyable working atmosphere. All of them were willing to discuss problems, including those not related to computer science. Especially, Martin Steffen was an enormous help because of his immense general knowledge concerning computer science.

He could always show a broader context of the topics discussed and hinted to related research areas. Ben Lukoschus was a brilliant consultant in LaTeX questions and helped me a lot in improving the layout of my thesis. Ralf Huuck always kept me up to date concerning the social events one had to attend. Finally, Marcel Kyas contributed an elaborate example of the verification of synchronous parameterized systems that he verified in his diploma thesis. I thank Änne Straßner for her management and coordination at the chair, and for being informed how to fight all the bureaucracy at a German university.

And, last but not least, I want to thank my family for always backing my plans. After all, it was my father who encouraged me to do my doctorate, because there was still enough time to do other work afterwards. Also, my sister has always attested me that life in the ivory tower is not that bad compared to working in industry. Sadly, my mother does not live to see the moment when I have definitely finished my studies. But I feel she will notice.

Chapter 1

Introduction

Software development is an error-prone task. This statement can be justified by several spectacular software failures. Professional programmers failed to correctly implement even the most basic software, the microcode running a processor, as the Intel Pentium flaw shows [Sti95]. When the German railway attempted to replace its long-established railway-switch tower at Hamburg-Altona by a fully computerized system, the central computer failed immediately after starting the new system [Meh95]. An error in the routine to handle stack overflows failed. The whole station had to be closed and for some days thousands of travelers had to be redirected. The first Ariane 5 launch vehicle destructed itself because of an integer overflow that led to a software shut-down [Gle96]. During the Gulf War a timing error in a United States' Patriot missile defense system located at Dhahran resulted in the wrong classification that a detected object was no missile [Wol92]. The Patriot system ignored the incoming Scud missile.

Software development is in particular more error-prone when considering distributed computing. Embedded software is invading more and more applications such as consumer electronics, avionics, process control, or medical systems. In many applications several embedded systems have to cooperate. For instance, in modern cars systems providing safety-critical control functions such as ABS braking systems and airbags have to be coordinated. A way to allow such coordination is to connect the different systems via a bus to a network. Protocols controlling these networks have to implement coordination and communication services reliably. For automotive applications we could mention the CAN-Bus (Controller Area Network). New Projects like FlexRay and Time-Triggered Protocol (TTP) develop an open framework that allows to integrate new components easily. To guarantee safety in critical distributed systems not only the correctness of each component is essential but also the correctness of the underlying communication protocols

is crucial.

These communication protocols are designed to establish fault-tolerant communication between the components and to provide general services for coordination issues. Such services can be fault detection, leader election, group membership, mutual exclusion for shared resources, distributed consensus, or distributed shared memory. These algorithms are not only needed in embedded software. Computers connected via the Internet use these algorithms to establish reliable communication, to maintain routing tables to deliver messages, or to enable access to central data archives. Even inside a single computer synchronization algorithms are used to coordinate processes that are executed concurrently on that system.

All these algorithms have one thing in common: they are expected to run for an arbitrary number of systems demanding such services. The number may be known at system start-up and be static, may change during computation, or may be even unknown. Whenever the actual number of participating processes is needed in a protocol, it is often specified as a parameter of the protocol. Dozens of parameterized distributed algorithms can be found in textbooks like [Lyn96, AW98]. In principle, to establish correctness for these so-called *parameterized systems* one has to prove every single protocol instance to be correct. Hence, an infinite family of protocol instances has to be verified. This leads naturally to the subject of this thesis: an algorithmic verification approach to parameterized networks.

1.1 Contribution

We present a framework to model and to verify parameterized networks

$$\{\mathcal{S}(0,n) \,\|\, \cdots \,\|\, \mathcal{S}(n-1,n) \mid n \in \omega\}$$

that consist of finite state processes. The single processes are characterized by a process template $\mathcal{S}(i,n)$ that may use i as a process identifier to distinguish the processes of the network and that may use n to indicate the network size. The $\|$ operator denotes the parallel execution of the processes. We define classes of networks that may be executed synchronously, as well as asynchronously. Our framework allows to uniformly model networks organized in a linear, ring, or tree topology, or fully connected networks.

For the defined classes of networks we present algorithmic methods to establish that a linear-time temporal property ψ holds for all network instances. Therefore, we have to prove

$$\mathcal{S}(0,n) \,\|\, \cdots \,\|\, \mathcal{S}(n-1,n) \models \psi \ ,$$

for every $n \in \omega$. That means, we have to prove that every computation of an arbitrary instance $\mathcal{S}(0,n) \parallel \cdots \parallel \mathcal{S}(n-1,n)$ of the network satisfies ψ. The properties that we are able to verify include safety as well as liveness properties. Therefore, we specify assumptions about the concrete scheduling in the network as additional fairness conditions $\mathcal{F}$. In such a case every *fair* computation of the network has to satisfy ψ. Moreover, we allow ψ to describe the behavior of single processes. Then, ψ has the form $\forall p_1, \ldots, p_k : \varphi(p_1, \ldots, p_k)$, where φ itself is a linear-time temporal formula. Such *universal properties* state that every collection $p_1, \ldots, p_k$ of processes satisfies φ.

Apt and Kozen show in [AK86] that parameterized verification is undecidable even for networks of finite state processes. Since we cannot find a sound and complete method, we have to give up one of these requirements. The natural choice of course is to keep soundness. We present a so-called *semi-automatic method*, i.e., our method automatically tries to establish properties of the systems, it is sound when successful, and in case of failure some user-interaction is needed to refine the verification.

1.2 Approach

In this thesis we apply the verification by abstraction approach [CGL94, DGG94]. An abstraction focuses on the core functionality of a given system and abstracts from details that are unimportant with respect to the desired properties. Formally, an abstraction relation maps the states of the concrete system to an abstract state space. The transitions of the abstract system are defined such that each step of the concrete system can be simulated at the abstract level. Hence, the behavior of the abstract system over-approximates the concrete one. If one can show that the abstract computations satisfy certain constraints, the computations of the concrete system will do so as well. This allows to transfer verification results from the abstract to the concrete system. The benefit of using an abstraction-based verification approach is that the resulting abstract system is simpler or even finite-state. The problems of this approach are to find an adequate abstraction relation and to construct the abstract system characterized by that relation.

To have feasible methods that solve the problems when applying abstraction to parameterized networks we introduce some restrictions: as already mentioned, we require each process in the network to be finite-state and, moreover, we assume the transitions of the processes to be expressible in a restricted logic AF(n). In principle AF(n) is the first-order fragment of WS1S, the Weak Second-order logic of One Successor. WS1S allows quantification over first-order and second-order variables. The first-order variables

are interpreted over the natural numbers whereas the second-order variables are interpreted over finite subsets of the natural numbers. WS1S is a decidable logic. Moreover, it allows to construct automata that recognize all models satisfying a given WS1S formula. Hence, our idea is as follows:

1. Represent the network's configuration, i.e., the states the processes of the network reside, as subsets of the natural numbers and characterize the execution steps of the network as WS1S formulae.

2. Define an abstraction relation in terms of WS1S predicates. Each predicate splits the concrete state space into a set of states that satisfy the predicate and a set of states that do not satisfy it. An abstraction relation defined in such a way is called *predicate abstraction*. Choose a finite set of such WS1S predicates.

3. Exploit decidability of WS1S to automatically compute the transitions of the abstract system. For a finite set of predicates used to define the abstraction relation the resulting abstract system is finite-state.

4. Apply model-checking techniques to check whether the abstract system satisfies the abstract properties, i.e., the properties required for the concrete system have to be adapted to specify the abstract system.

We identify a generic class of WS1S predicates that proves to be adequate to define a powerful abstraction relation. Then, the four steps can be conducted automatically and a positive answer establishes the parameterized network to satisfy its specification. Because of the undecidability result we cannot expect the required properties to hold for the abstract system in all cases even if the concrete network is correct. The abstraction may be to coarse to establish the correctness property. Therefore, we sometimes have to refine the abstraction relation which makes our method semi-automatic.

We now elaborate the sketched approach of abstraction-based, algorithmic verification of parameterized networks.

WS1S Systems We introduce several classes of parameterized networks, synchronous or asynchronous, with or without global variables, that can be modeled in our framework. A network of processes is described by a template process. The process template $\mathcal{S}(i, n)$ may use the network size n and its own identifier i to define its initial state and its transitions. We restrict the processes to be finite-state and the transition relation to be characterized in the restricted logic AF(n).

For these classes of networks we give translations into *WS1S systems*. These systems manipulate second-order variables which we can use to model

sets of processes that are in the same state. This allows a uniform representation of configurations of arbitrary network instances. Each process in the network is in one of finitely many states, a configuration of the corresponding WS1S system has finitely many sets. We define the translation such that each set corresponds to a process state and contains the processes that are actually in that state. Hence, we represent the infinite family of networks $\{\mathcal{S}(0,n) \parallel \cdots \parallel \mathcal{S}(n-1,n) \mid n \in \omega\}$ as a single WS1S transition system $\mathcal{W}$ whose variables range over finite subsets of ω. To model asynchronous transitions the WS1S translation requires the existence of a process that has to perform a step. Synchronous transitions are modeled such that all processes have to perform a step simultaneously. Then we can prove that a given network and its translation behave similarly; every computation of a network instance is represented inside the WS1S system.

Proving Safety Properties The set systems are standard transition systems without parameters. We can now apply abstraction to verify the WS1S systems. We first consider *safety properties*. Intuitively a safety property states that something "bad" is never going to happen where "bad" describes a certain system configuration that must not occur. Since an abstraction over-approximates the original system, by absence of the corresponding "bad" configuration in the abstract system we can conclude its absence also for the concrete one.

We define the abstraction relation as a predicate abstraction. Each predicate maps the states of the WS1S transition system $\mathcal{W}$ to one boolean abstract variable. We give a generic class of WS1S predicates that we prove to be sufficient for verification of parameterized systems. For a finite set of such predicates the abstraction relation defines a finite abstract system.

We introduce two types of abstraction which we call *global* and *local abstractions*. To define a global abstraction relation we choose closed WS1S predicates. Such predicates describe the overall network configuration. In contrast, WS1S formulae with free variables, for instance $p \in X$, allow to keep track of arbitrary processes. These formulae are used to define local abstractions. Local abstractions are used to prove universal properties. If the abstract system satisfies a universal property for an arbitrary process, the property holds for all processes in the concrete network.

Having defined an abstraction relation, we exploit decidability of WS1S to construct the abstract system $\mathcal{S}_A$ automatically. Hence, for a given abstraction relation construction and verification of the abstract system is fully automatic. To verify the abstract system we also have to find an abstract

formula ψ_A such that if $\mathcal{S}_A$ satisfies ψ_A, one can deduce

$$\mathcal{S}(0,n) \,\|\, \cdots \,\|\, \mathcal{S}(n-1,n) \models \psi \ ,$$

for every $n \in \omega$, using the preservation results of [CGL94, DGG94]. We do so by replacing the state formulae of ψ by corresponding abstract variables.

Because of the undecidability for parameterized systems we have to accept *false negatives*, i.e., counterexamples found for the abstract system that do not correspond to any concrete computation. Such false negatives are used to refine the abstraction relation.

Proving Liveness Properties The preservation results of [CGL94] and [DGG94] also hold for *liveness properties* ψ: if one can establish that the abstract system satisfies ψ_A, one can conclude that $\mathcal{S}(0,n) \,\|\, \cdots \,\|\, \mathcal{S}(n-1,n) \models \psi$. Nevertheless, things are more complex when considering liveness.

A *liveness property* states that eventually something "good" happens. To prove a liveness property one has to show that all executions eventually lead to "good" configurations. That makes verification of liveness properties by abstraction difficult, since the abstraction introduces more computations at the abstract level. An abstraction merges states of the original system and may introduce cycles not present in the concrete system. These cycles cause a new type of false negatives: infinite paths that never reach a "good" configuration. If the concrete system always reaches a "good" configuration, there exist ranking functions that characterize the progress. In principle, they specify the number of remaining steps that are needed to reach the "good" configuration. In our method we augment the abstract system with special variables representing de- or increase of these ranking values. This allows us to constrain the computations of the abstract system by requiring the ranking value not to be decreased infinitely often without increasing it infinitely often. We give heuristics to define ranking predicates such that the corresponding fairness requirement excludes such counterexamples. The generated fairness conditions are *guaranteed* to hold for the parameterized network and, hence, can be added safely.

Fairness requirements may also be stated for the concrete network to express assumptions on a fair scheduling of the processes. If such assumptions are needed to guarantee progress at the concrete level, we have to express these fairness conditions also for the abstract system. We demonstrate how to choose the abstraction relation such that we can lift fairness requirements from the concrete to the abstract level.

1.3 Applicability

We prove applicability of our method by verifying several protocols. In the first parts of the thesis we analyze three mutual exclusion algorithms: Szymanski's algorithm that communicates via multi-reader shared variables, an algorithm inspired by Dijkstra's algorithm, and an algorithm to implement mutual exclusion in a token ring.

We use Szymanski's algorithm as a running example to illustrate the verification steps. We present global and local abstractions for it, and for both we prove mutual exclusion to hold. Moreover, we augment the abstract system to respect generated fairness requirements in order to prove accessibility of the critical sections and linear waiting.

Dijkstra's algorithm uses a global variable to control access to the critical section. We establish mutual exclusion and accessibility for the algorithm. The liveness property depends on several fairness requirements attached to the algorithm. We show how to lift them to the abstract level.

As a third mutual exclusion example we verify mutual exclusion in a token ring. The results establish the mutual exclusion property and the liveness property that each process will eventually enter its critical section. Moreover, verification of the algorithm illustrates how to tackle token rings in general.

Beside the practical applicability we prove some theoretical results about our framework. For restricted classes of parameterized systems the model-checking problem is decidable. We show that the set of generic abstraction and ranking predicates is sufficient to verify those systems applying our method.

Whereas the mutual exclusion algorithms which we verify are already finite-state, many real life examples are based on infinite-state processes. In our motivation we mentioned the time-triggered protocol. It provides several services to the processes connected to the network. One feature is a group membership protocol that enables each process to maintain the set of connected, non-faulty processes. We prove the algorithm to be correct by incremental verification; we first prove two non-faulty processes to have the same set, then we prove that set to be correct. Both steps are performed inside our framework.

As another motivating example we mentioned cache-coherence protocols. These protocols have two parameters: the number of participating processes and the addresses of the memory space. We first prove a result that allows under certain circumstances to reduce the verification of two-dimensional parameterized systems to one-dimensional parameterized systems. Additionally, we use the theorem prover PVS to justify further simplifications. Afterwards, we can apply our approach and establish sequential correctness

of the protocol.

Marcel Kyas also applied our approach and presents an elaborate example for the verification of a synchronous system in [Kya01].

Another direction to generalize the systems is to take other topologies into account. With WS1S systems we primarily attack networks with broadcast communication, since we have communication by shared variables. Moreover, linear and ring topologies can be modeled by constraining the processes exclusively to communicate with their direct neighbors. There we exploit the fact that we have the successor function in WS1S. We show how to adapt our approach to the decidable logics WS$\boldsymbol{n}$S, for Weak Second-order logic with n Successors. These allow to model arbitrary tree topologies.

1.4 Related Work

Of course, we are not the first to verify parameterized networks. We present the related work covering the topic of verification of parameterized networks separated into three groups: work that is based on deductive methods, work that gives decidability results and presents algorithms to verify restricted classes of parameterized systems automatically, and work that presents the so-called regular model-checking approach. While the first two are of interest to get an overview on parameterized verification, we discuss the works related to the last group because of the close relationship of the systems modeled there and our WS1S systems. This close connection is caused by the fact that regular languages are representable in WS1S.

1.4.1 Deductive Methods

To prove correctness of parameterized networks one has to establish correctness for arbitrary sizes of the network. From a mathematical point of view there are at least two adequate methods: choose an arbitrary n and prove the property for an instance of size n, or prove inductively for all n that networks of size n and, hence, all instances are correct.

Applying the first method reduces verification of parameterized networks to the verification of concurrent systems which is quite well studied [MP95b, MP95a, dRdBH$^+$01].

The proof obligations needed to establish correctness are numerous. Thus, some approaches use machine assistance to construct the verification conditions and to keep track of their proof status. In [NN99] the Owicki & Gries method [OG76a, Owi75] is formulated as a PVS theory [ORSvH95] and generalized to the parameterized case. For given systems the verification

conditions are constructed automatically and their proofs are assisted by the system if not conducted completely automatically. [NN01] shows their generalized method to be complete.

In [BLO98b, Bau98] the focus is on the automatic generation of assertion networks. The guards and effects of transitions are used to construct assertions for certain locations that are guaranteed to hold by construction. The constructed assertion network is checked for inductiveness. Unproved subgoals are used to strengthen the assertions. The implemented tool is also based on PVS.

PVS as a back-end is also used in [LS97, BLO98a] where its theorem proving capabilities are used to construct an abstract system automatically. [LS97] does so explicitly for parameterized networks and gives heuristics how to define the abstraction relation.

The abstract system represents a network invariant that abstracts any arbitrary number of processes. The idea of finding such a network invariant is based on [KM89, WL89, BCG89, SG89, HLR92, LHR97].

While finding a network invariant originally addresses linear networks, it has been generalized in [CGJ95] to networks generated by context-free grammars. In [CGJ95], abstract transition systems are used to specify the invariant. An abstract transition system consists of abstract states specified by regular expressions and transitions between abstract states.

While a network invariant tries to describe the whole system, [JL98] constructs an abstraction that focuses on two processes in order to prove mutual exclusion for Burn's algorithm. The mutual exclusion property makes a statement about two arbitrary processes in the network, hence, they give an abstraction which preserves the external behavior of any two concrete processes running in the environment of all other processes. We use this idea in Chapter 5, as well.

1.4.2 Decidability Results

The proof of Apt & Kozen's undecidability result shows that parameterized systems are quite powerful, i.e., the single processes need not much capabilities to be able to express computations of a Turing-machine. Hence, rigorous restrictions are needed to obtain decidability results.

The systems considered in [GS92] consist of a unique control process and an arbitrary number of user processes with identical definitions. In particular, the processes have no process identifiers to distinguish them from each other. The transitions are labeled to allow synchronization. Other kind of guards are not allowed. [GS92] proves decidability of such systems by modeling them

as vector addition systems with states (VASS). For such systems decidability results are presented in [Kos82, May81].

In [EN96] the same model is used: a control process running in parallel with many user processes. In contrast, the processes run in a synchronous manner. Still the user processes behave totally symmetrically. Communication labels are not needed, since the processes already run synchronously. Instead, restricted guards are allowed. [EN96] gives a finite abstraction which is exact for the finite behavior of the system. To check for liveness behavior the authors give an algorithm to detect fair paths. We discuss their decidability result in more detail in Chapter 9.

In [EN95] token rings are analyzed. They show that for classes of ring networks of arbitrary size, there exists a natural number such that the verification of the parameterized ring can be reduced to the verification of networks of size up to that number. This is discussed also in more detail in Chapter 9.

Between the decidability results presented above and the regular model-checking approach coming next, lies the approach of [Mai01]. There, one tries to model-check safety properties for parameterized systems. This is done by constructing a proof tree backwards starting from a bad state and applying the predecessor operation. [Mai01] gives conditions under which this construction terminates and obtains a characterization of a subclass for which the problem is decidable.

1.4.3 Regular Model-Checking

An alternative approach to the verification of parameterized systems is *regular model-checking*. It is based on the idea of representing sets of states of parameterized networks by regular languages. One could think that model-checking is only applicable for finite state systems. But that is not true; one needs finite representations for sets of configurations and one needs a guarantee that the fixed point computation (in order to get the set of reachable states) terminates.

The first point is addressed in [KMM$^+$97]. There its authors propose to use *rich assertional languages* to represent sets of configurations. In fact, they use regular languages. The finite alphabet represents the finitely many states of a process. A word in the language characterizes one configuration of the system, where each position in the word represents one process. Hence, they can apply the star operator to handle a parameterized number of processes. For example, $a^\star$ is the set of configurations where all processes are in state a, $a^\star bc^\star$ expresses that a number of processes are in state a, one process in state b, and some more in state c. Unfortunately, [KMM$^+$97] gives no solution to the second point; hence, they are only able to represent and verify a simple

token passing algorithm. Consider, for example, an unconditional transition from state a to b. Then a forward analysis for starting configuration $a^\star$ would generate:

$$a^\star \rightarrow a^\star b a^\star \rightarrow a^\star b a^\star b a^\star \rightarrow \cdots \quad .$$

The idea of using rich assertional languages is also applied in [ABJN99]. Its authors generalize the considered class of systems by introducing contexts to represent global guards for transitions. Primarily, they solve the second problem by constructing a so-called accelerated transducer that produces the transitive closure of transitions. In the example above they directly compute $(a + b)^\star$. [JN00] characterizes exactly the class of systems they can handle and for which the accelerated transducer exists. [BJNT00] coins for this kind of verification the term *regular model-checking*. [DLS01] gives an alternative approach to compute a transducer representing the transitive closure of applying a transducer iteratively. The approach is generalized to tree topologies in [AJMd02] which emphasizes the relation to our work since their generalization corresponds to ours presented in Chapter 12 that uses WS$\boldsymbol{n}$S to model tree topologies.

The work presented in [PS00, PRZ01, PZ01, PXZ02] also uses WS1S transition systems to model parameterized systems. [PS00] expresses iterative application of transitions as an accelerated transition described in WS1S which allows for simple examples to compute the set of reachable states exactly. [PRZ01, PZ01] give heuristics to calculate auxiliary invariants for WS1S systems in order to find an inductive invariant that allows to prove safety properties. [PXZ02] also applies an abstraction-based verification approach close to ours and presents a technique to strengthen fairness conditions at the abstract level.

[FO97b, DFN01] also use regular sets as assertional language to represent configurations. In contrast to the model-checking approach these authors use rewriting techniques to come up with a regular expression characterizing the reachable states. [FO97b] formally has to justify that their "guess" is indeed correct, while [DFN01] uses Caucal's algorithm to construct an unavoidable set of configurations.

Another use of rich assertional languages is presented in [FO97a] where Presburger arithmetic is used to describe configurations of Petri nets. Using the decidability of Presburger arithmetic the authors are able to compute reachability sets of Petri nets.

1.5 Structure of the Thesis

After introducing the problem of the verification of parameterized networks, Chapter 2 defines several classes of parameterized systems which will be subject to verification efforts in the sequel. The chapter concentrates on characterizing networks with finite state processes. We consider both synchronous and asynchronous semantics.

Chapter 3 motivates the introduction of WS1S systems and defines a translation from parameterized systems to WS1S systems. These systems are ordinary transition systems except that they have second-order variables and that the parameterization gets hidden in the initial condition. This makes it straightforward to define abstraction relations in order to get a finite abstract system.

The second part of the thesis presents an abstraction-based verification approach to verify parameterized networks. Chapter 4 shows how to prove properties of a system by verifying an abstraction. The discussion of abstraction introduces model-checking techniques to verify finite state systems. Moreover, Chapter 4 discusses what kind of properties are of interest for parameterized networks and defines a logic to express them. In Chapter 5 two types of abstractions are presented to define a finite abstract system. Heuristics are given in order to get a meaningful abstract system in the sense that interesting properties can be proven for it. Model-checking the automatically constructed abstract systems is explained in Chapter 6.

Chapter 6 unveils some problems in the verification of liveness properties. Hence, the next part of the thesis concentrates on solving these problems. In Chapter 7 we construct fairness conditions which are guaranteed to hold in the concrete system. Those fairness requirements allow to exclude certain unfair cycles of the abstract system which would disprove the desired liveness property. To handle those fairness conditions we have to augment our transition systems. We do that for both the concrete and the abstract systems. Therefore, we also discuss how to lift given fairness constraints from the concrete to the abstract level.

Chapter 8 presents several examples to illustrate our verification approach. It contains as classical benchmark for the verification of parameterized systems Szymanski's mutual exclusion algorithm. To show the complex interaction of several fairness conditions, we verify a variant of Dijkstra's mutual exclusion algorithm. Another classical example for ring topologies is the token ring which is also verified in that chapter.

In Chapter 9 we justify the way we construct fairness conditions by simulating decidability results for restricted classes of parameterized networks in our framework. The completeness results illustrate how powerful the gen-

erated fairness conditions are and show their application.

The last part of the thesis offers an outlook how the restrictions chosen for the systems can be weakened. In our framework, we focus on parameterized systems with finite-state processes. Nevertheless, Chapter 10 shows the verification of a group membership protocol where each process maintains its own set variable containing the actual members of the network. We give some rules and heuristics in order to simplify such systems to make them fit into our framework. Moreover, we present the verification of a multi-dimensional parameterized network.

Chapter 11 exploits the fact that we can express the concept of neighborhood in WS1S. In linear or ring topologies some interesting properties base on propagating information from neighbor to neighbor. We give a proof rule formalizing under which assumptions this information eventually reaches every process.

Chapter 12 shows how to apply the techniques presented in another setting. Namely, we can adapt the same techniques with WS2S or WS$\boldsymbol{n}$S to handle tree topologies. We give an example and show how to apply the induction principle in trees.

Chapter 13 summarizes the results of this thesis, emphasizes the contribution of this work, and gives an outlook upon future work.

1.6 Publications

Parts of thesis have been published in [BBLS00, BLS00, BLS01, BBLS01]. The idea of using WS1S transition systems to represent parameterized system and to attack them with an abstraction-based approach is published in [BBLS00]. The basic idea how to generate fairness conditions to prove liveness properties and many examples are presented in [BLS00, BLS01]. The example of a distributed shared-memory protocol representing a multi-dimensional parameterized network which is verified in detail in Chapter 10 can be found in [BBLS01].

Part I

Parameterized Networks

Chapter 2

Systems of Interest

Systems with a parameterized number of homogeneous processes occur in many applications, e.g., mutual exclusion algorithms, cache coherence or communication protocols. The algorithms differ on their underlying execution model as well as on their communication mechanisms. In textbooks on concurrent and distributed algorithms, like [Lyn96, AW98], most of the algorithms are presented in a parameterized manner. A generic process template is given which is then instantiated with the total number of participating processes and a unique process identifier. For some algorithms the knowledge of the total size of the network and its own identity is important, for others not.

In this chapter we introduce transition systems to model such process templates. A transition system is a triple consisting of a set of variables, an initial condition constraining the initial evaluations of the variables, and transitions characterizing the change in the process state. A process template $\mathcal{S}(i, n)$ is parameterized by its own identity i and the size of the network n. Our transition systems contain a special variable *pc* (program counter) to model the control flow. This allows to represent protocols, written in common programming languages, in our framework easily. The transitions are guarded with predicates, controlling enabledness of transitions, and assignments, changing the process' state space.

To represent parameterized networks we define parallel execution of transition systems $\mathcal{S}_1 \| \mathcal{S}_2$ and what we mean with $\mathcal{S}^n$. As already mentioned, verification of parameterized systems is undecidable even for finite state processes. To develop a promising approach to the verification of networks, we have to restrict the process templates. Beneath requiring the processes to have a finite state space, we introduce a simple first order logic and characterize the systems of interest therein.

2.1 Transition Systems

To formalize properties of a given system we have to choose a formal specification language and a computational representation. For both we need an underlying assertion language $\mathcal{A}$ including interpreted symbols for expressing the standard operations and relations over some concrete domains. These concrete domains should at least contain the set of integers. Assertions (we also say predicates) in $\mathcal{A}$ are interpreted in states that assign values to the variables of $\mathcal{A}$. With Σ we denote the set of states. We write $s \models \varphi$ for a state s and a predicate φ if s satisfies φ. If all states satisfy φ this is denoted by $\models \varphi$. Let $[\varphi]$ denote the set of states satisfying φ, $[\varphi] = \{s \in \Sigma \mid s \models \varphi\}$. We use $\mathit{free}(\varphi)$ as a notation for the free variables occurring in the predicate φ. We also have a set of expressions underlying $\mathcal{A}$. If e is such an expression, we denote the evaluation of e in a state s with $e(s)$.

Now, before we can define how a parameterized number of processes interact we need a computational representation for a single process. Hence, we first define sequential transition systems closely related to those in [MP95b, dRdBH+01].

Definition 2.1 (Transition system)
A transition system is a structure $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \Theta)$, where

- $\mathcal{V}$ is a finite set of typed variables, $\mathcal{V} = \{x_1 : D_1, \ldots, x_n : D_n\}$. Each variable x_i ranges over the finite or infinite data domain D_i. x_1 is assumed to be a control variable ranging over the finite domain D_1. Therefore, we denote x_1 with *pc* and D_1 with $\mathcal{L}$, the set of *control locations*.

- $\mathcal{T}$ is a finite set of transitions. Each transition τ is characterized by a quadruple $(\mathit{source}(\tau), \mathit{guard}(\tau), \mathit{effect}(\tau), \mathit{target}(\tau))$, where

  - $\mathit{source}(\tau)$ and $\mathit{target}(\tau)$ are in $\mathcal{L}$. $\mathit{source}(\tau)$ is the source location and $\mathit{target}(\tau)$ is the target location of transition τ.
  - $\mathit{guard}(\tau)$ is the guard of t. $\mathit{guard}(\tau)$ is an assertion in $\mathcal{A}$ with its free variables in $\mathcal{V}$.
  - $\mathit{effect}(\tau)$ is a set of equations like $x = e$ with free variables in $\mathcal{V}$. For each $x \in \mathcal{V}$ there is at most one equation with x on its left hand side.

- Θ is a predicate in $\mathcal{A}$ with free variables in $\mathcal{V}$, the initial predicate, it specifies the initial condition. □

Instead of listing up all transitions of a system we present the transition systems graphically. This makes the flow of control explicit. Then, a transition τ can be represented as illustrated in Figure 2.1. ℓ_i represents $source(\tau)$

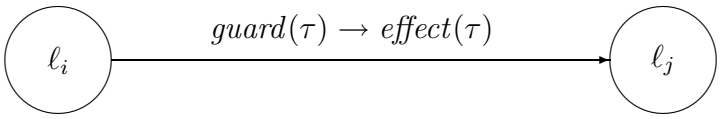


Figure 2.1: Graphical representation of τ

whereas ℓ_j represents $target(\tau)$.

Each transition induces a relation $\rightarrow_\tau$ on the state space Σ. Given a transition τ and states s, s' we have $s \rightarrow_\tau s'$ if the following conditions are satisfied:

- $s \models pc = source(\tau) \wedge guard(\tau)$, i.e., s satisfies the enabledness condition of τ.

- for each $x = e$ in $effect(\tau)$, s' satisfies $s'(x) = e(s)$; for each $x \in \mathcal{V} \setminus \{pc\}$ with no assignment in τ, s' assigns the same value to x as s; and $s'(pc) = target(\tau)$.

If $s \rightarrow_\tau s'$ holds s' is called a τ-successor of s. Now we are able to characterize the transition relation by a predicate ρ_τ. Given a transition τ with $effect(\tau) = \{y_1 = e_1, \ldots, y_n = e_n\}$ and $\{z_1, \ldots, z_m\}$ the set of unchanged variables we have:

$$\begin{aligned} \rho_\tau \equiv \quad & guard(\tau) \wedge pc = source(\tau) \wedge \\ & \textstyle\bigwedge_{i=1}^{n} y_i' = e_i \wedge \bigwedge_{i=1}^{m} z_i' = z_i \wedge pc' = target(\tau) \ , \end{aligned}$$

where the primed variables are interpreted by s'.

Since we use transition systems as representation for reactive systems, we are interested in computations of those transition systems. A computation of a system $\mathcal{S}$ is a sequence of states $\sigma : s_0, s_1, \ldots$, such that s_0 is an initial state satisfying Θ and for every $i \geq 0$ there is a transition $\tau \in \mathcal{T}$ satisfying $s_i \rightarrow_{tau} s_{i+1}$. With $[\![\mathcal{S}]\!]$ we denote the set of all computations of $\mathcal{S}$.

In order to represent parameterized systems by transition systems, we model each process of such a system as a transition system. So we have to consider the parallel composition of transition systems.

2.2 Asynchronous Systems

First, we assume an interleaving semantics. So we have an asynchronous model with communication by shared variables. To define the semantics of the parallel construct, we define the product of two transition systems.

Definition 2.2 (Asynchronous product)
Let $\mathcal{S}_i = (\mathcal{V} \cup \{pc_i\}, \mathcal{T}_i, \Theta_i)$, for $i = 1, 2$, be transition systems. The *asynchronous product* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is a transition system $\mathcal{S}_1 \otimes \mathcal{S}_2 = (\mathcal{V} \cup \{pc\}, \mathcal{T}, \Theta)$, where

- pc ranges over $\mathcal{L} = \mathcal{L}_1 \times \mathcal{L}_2$.
- a transition $\tau = ((\ell_1, \ell_2), \mathit{guard}(\tau), \mathit{effect}(t), (\ell_1', \ell_2'))$ is in $\mathcal{T}$ iff
  - there is a transition $\tau_1 = (\ell_1, \mathit{guard}(\tau), \mathit{effect}(\tau), \ell_1')$ in $\mathcal{T}_1$ and $\ell_2 = \ell_2'$, or
  - there is a transition $\tau_2 = (\ell_2, \mathit{guard}(\tau), \mathit{effect}(\tau), \ell_2')$ in $\mathcal{T}_2$ and $\ell_1 = \ell_1'$.
- $\Theta \equiv \Theta_1 \wedge \Theta_2$, where pc_i is replaced by the ith projection of pc, i.e., $\pi_i(pc)$. □

Now we can define the set of computations of $\mathcal{S}_1 \| \mathcal{S}_2$ to be that of $\mathcal{S}_1 \otimes \mathcal{S}_2$.

2.2.1 Fair Transition Systems

So, we model the interleaving semantics of asynchronous, parallel systems by nondeterminism in the product of transition systems representing the processes.

This yields some problems in the verification of parallel algorithms, since the interleaving semantics allows one process to take its transitions exclusively. In reality, this problem is assumed to be solved by an intelligent scheduling algorithm of the operating systems that guarantees a fair amount of computation time for every process. Or, considering distributed algorithms, on makes assumptions on the computation speed of the underlying hardware.

In theory, the common solution is to add fairness conditions to the formal model in order to exclude some *unfair* computations. Usually, these fairness conditions subdivide into two classes. The first is called *weak fairness* and expects a process to eventually take a weak fair transition if its guard is enabled continuously. The second one is called *strong fairness*. A transition marked as being strong fair requires the transition to be taken eventually if

the guard is enabled infinitely often. Indeed, strong fairness restricts the set of computations stronger than weak fairness requirements.

Formally, we define *fair transition systems* as a quintuple $\mathcal{S}^{\mathcal{F}} = (\mathcal{V}, \mathcal{T}, \Theta, \mathcal{J}, \mathcal{C})$ where $\mathcal{J} \subseteq \mathcal{T}$ denotes the set of *just*, i.e., weak fair, transitions, and $\mathcal{C} \subseteq \mathcal{T}$ denotes the set of *compassionate*, i.e., strong fair, transitions. For the asynchronous product $\mathcal{S}_1 \otimes \mathcal{S}_2$ a transition is in $\mathcal{J}$ ($\mathcal{C}$) if the underlying transition $\tau_1 \in \mathcal{J}_1$ ($\tau_1 \in \mathcal{C}_2$), respectively, $\tau_2 \in \mathcal{J}_2$ ($\tau_2 \in \mathcal{C}_2$).

The computations of $[\![\mathcal{S}^{\mathcal{F}}]\!]$ are those computations in $[\![\mathcal{S}]\!]$ satisfying all fairness conditions required by $\mathcal{J}$ and $\mathcal{C}$. Though in theory it is reasonable to distinguish between these two types of fairness, in practice both are often expressed as a single fairness condition $\mathcal{F}$. Then the definition of a fair transition system reduces to a quadrupel $\mathcal{S}^{\mathcal{F}} = (\mathcal{V}, \mathcal{T}, \Theta, \mathcal{F})$.

2.2.2 Pseudo Code

Instead of representing transition systems graphically, for asynchronous (but also for synchronous) systems, it is sometimes convenient to give a pseudo code representation. Each line of code starts with a unique label and corresponds to the control flow of a transition system, i.e., the label of the current line of code to be executed corresponds to the value of *pc*. The guards are represented by an **await** statement followed by the formula characterizing the guard. Then, a list of assignments may follow. A **goto** statement determines the next line of code to execute, if not present the following line is executed. As compound statement we have the **if_then_else_fi** statement checking the guard following the **if** keyword and continuing according to the result. Another compound statement is the **loop_do_od** construct which computes the body as long as the guard following the loop keyword is evaluated to *true*. In this context we use the keyword **forever** synonymously with *true*.

As an example we present Szymanski's mutual exclusion algorithm. It has to guarantee exclusive access to the critical section which is marked as **critical section**. The critical section describes restricted resources which are not available for more than one process at once, like printers, other hardware, or global data structures in operating systems. The algorithm presents a protocol to handle access to the critical section. It is executed by the processes that leave their non critical sections and try to enter their critical sections.

Indeed, Szymanski presented several algorithms in this period reducing the number of bits needed to solve the mutual exclusion problem. Except for the variable *pc* needed to represent the flow of control, the algorithm only uses three bits modified locally to synchronize access to the critical section.

That makes Szymanski's algorithm a perfect candidate for our verification approach. Thus, we use it as a running example throughout this thesis.

Example 2.3 (Szymanski's mutual exclusion algorithm)
An instance of n processes running Szymanski's algorithm in parallel starts initially with all boolean variables set to *false*, i.e.,

$$\textbf{mutex} \equiv a, s, w := \mathit{false}, \mathit{false}, \mathit{false}; [\mathcal{S}(0,n) \| \mathcal{S}(0,n) \| \ldots \| \mathcal{S}(n-1,n)]$$

Each process executes the following pseudo code:

$\mathcal{S}(i,n) \equiv$ **loop forever do**
ℓ_0: **noncritical section**; $a[i] := \mathit{true}$
ℓ_1: **await** $\neg\, \exists_n\, j : s[j]$
ℓ_2: $w[i], s[i], a[i] := \mathit{true}, \mathit{true}, \mathit{false}$
ℓ_3: **if** $\exists_n\, j : a[j]$
then $s[i] := \mathit{false}$; **goto** ℓ_4
else $w[i] := \mathit{false}$; **goto** ℓ_5
ℓ_4: **await** $\neg\, \forall_n\, j : \neg s[j] \vee w[j]$; $s[i] := \mathit{true}$
ℓ_5: **await** $\neg\, \exists_n\, j : w[j]$
ℓ_6: **await** $\neg\, \exists_n\, j : j < i \wedge s[j]$
ℓ_7: **critical section**; $s[i] := \mathit{false}$;
od □

In the sequel we prove several safety and liveness properties of Szymanski's mutual exclusion algorithm to illustrate applicability of our method. The obvious demand of a mutual exclusion algorithm is that the algorithm guarantees that there are never two processes in their critical section at the same time, i.e., in our example no two processes may reside at ℓ_7 at the same time. To exclude trivial solutions that block all processes another natural requirement is that processes eventually reach their critical section.

We hope that at the end of the thesis the confidence of the reader in the correctness of the algorithm is higher than after reading the informal justification due to Szymanski [Szy90]:

> "The idea behind the algorithm is simple. The prologue section simulates a waiting room with a door. All processes requesting entry to the critical section at roughly the same time gather first in the waiting room. Then, when there are no more processes requesting entry, processes inside the waiting room shut the door and move to the exit from the waiting room. From there, one by one, they enter their critical sections in the order of their numbering. Any process requesting access to its critical section at

that time has to wait in the initial part of the prologue section (at the entry to the waiting room).

The door to the waiting room is initially opened. The door is closed when a process inside the waiting room does not see any new processes requesting entry. The door is opened again when the last process inside the waiting room leaves the exit section of the algorithm."

It is intuitively clear that Szymanski's algorithm is already an example of a parameterized system. The algorithm is expected to be correct for any n. In the next section we get the theoretical basis to handle such systems formally.

2.3 Parameterized Networks

Having the notion of two systems running in parallel we are now able to define parameterized networks.

Definition 2.4 (Parameterized network)
Let $\mathcal{S}(i,n) = (\mathcal{V}, \mathcal{T}_i, \Theta_i)$ (resp. $\mathcal{S}(i,n) = (\mathcal{V}, \mathcal{T}_i, \Theta_i, \mathcal{J}, \mathcal{C})$) be a (fair) transition system. $\mathcal{S}$ may contain i and n as free variables that fix the process identifier (PID) i for $\mathcal{S}$ and the actual size of the network upon instantiation.

Then, we call the family of systems $\mathcal{P} = \{\mathcal{S}(0,n) \| \cdots \| \mathcal{S}(n-1,n) \mid n \in \omega\}$ the parameterized (asynchronous) network based on $\mathcal{S}$. With $\mathcal{P}(m) \in \mathcal{P}$ we denote an instance of the parameterized network consisting of m systems. Sometimes we also write $\mathcal{S}^m$.

Parameterized networks can be classified according to their variable set and the expressiveness of their guards. During this work we are mainly interested in the verification of parameterized systems consisting of finite state processes. Technically, we define a restricted class of parameterized systems by introducing boolean transition systems.

2.3.1 Monadic Parameterized Systems

Apart from requiring the processes to be finite state we assume the transitions to be expressible in a restricted logic. Intuitively, the logic we define in this section is the first-order fragment of the weak second-order logic of one successor (WS1S). That allows us to represent such restricted classes of parameterized systems as WS1S systems.

Let n be a variable. To define parameterized systems, we introduce the set $\mathrm{AF}(n)$ of formulae f defined by:

$$f \ ::= \ b[x] \mid x = x \mid x + 1 = x \mid x < x \mid \neg f \mid f \wedge f \mid \forall_n x : f \mid \exists_n x : f \ ,$$

where x is a *position variable*, b is a *boolean array variable*. The quantifiers $\forall_n x : f$, respectively, $\exists_n x : f$ are abbreviations for $\forall x : x < n \rightarrow f$, respectively, $\exists x : x < n \wedge f$.

Let $m \in \omega$. We denote by Σ_m the set of evaluations s such that $s(n) = m$, $s(x) \in \{0, \cdots, m-1\}$ and $s(b) : \{0, \cdots, m-1\} \rightarrow \{true, false\}$. Then, formulae in $\mathrm{AF}(n)$ are interpreted over evaluations $s \in \bigcup_{m \in \omega} \Sigma_m$ in the usual way. In the sequel, we also assume the usual notion of free variables, closed formulae, etc., as known.

Definition 2.5 (Boolean transition system)
A *boolean transition system* $\mathcal{S}(i, n)$, parameterized by n and i, where n and i are variables ranging over natural numbers, is described by the triple $(\mathcal{V}, \Theta, \mathcal{T})$, where

- $\mathcal{V} = \{b_1, \ldots, b_k\}$ and each b_j, $1 \leq j \leq k$, is a boolean array of length n.
- Θ is a formula in $\mathrm{AF}(n)$ with $\mathit{free}(\Theta) \subseteq \mathcal{V} \cup \{i\}$ and which describes the set of initial states.
- $\mathcal{T}$ is a finite set of transitions where each $\tau \in \mathcal{T}$ is given by a formula $\rho_\tau \in \mathrm{AF}(n)$ such that $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}' \cup \{i\}$ and $\mathcal{V}'$ contains a primed copy for each variable in $\mathcal{V}$. □

Notice, that if we identify the boolean array variables b_j by the m boolean variables $b_j[1], \cdots, b_j[m]$, the formulae describing the initial states as well as the transitions of $\|_{l=0}^{m-1} \mathcal{S}(l, m)$ are first-order WS1S formulae whose free variables are in $\mathcal{V}$, respectively, $\mathcal{V} \cup \mathcal{V}'$. Thus, $\mathcal{P}(m) = \|_{l=0}^{m-1} \mathcal{S}(l, m)$ is a transition system in the usual sense, i.e., it does not contain the parameters n and i. Hence, we assume the definition of a computation of $\mathcal{P}(m)$ as known and we denote by $[\![\mathcal{P}(m)]\!]$ the set of its computations.

Definition 2.6 (Monadic parameterized system)
A network $\mathcal{P} = \{\mathcal{P}(m) \mid m \geq 1\}$ is called monadic parameterized if it is built from boolean transition system $\mathcal{S}(i, n)$. The class of such systems is called *monadic parameterized systems* (MPS for short). □

To illustrate the above definitions we consider Szymanski's mutual exclusion algorithm [Szy88] as a boolean transition system.

Example 2.7 (Szymanski's algorithm as MPS)
In our formal model of boolean transition systems the control locations are also modeled by boolean array variables $\text{at\_}\ell_k$ for each location ℓ_k, $0 \leq k \leq 7$. According to Definition 2.5 the transition from ℓ_1 to ℓ_2 is given by the AF(n) formula:

$$\begin{aligned}(\neg\,\exists_n\, j : s[j]) \quad \wedge \quad \forall_n\, j : j \neq i \rightarrow & \bigwedge_{l=0}^{7}((\text{at\_}\ell_l[j] \leftrightarrow \text{at\_}\ell_l'[j]) \wedge (a[j]' \leftrightarrow a[j]) \\ & \wedge (s[j]' \leftrightarrow s[j]) \wedge (w[j]' \leftrightarrow w[j])) \\ & \wedge \text{at\_}\ell_1[i] \wedge \neg\text{at\_}\ell_1'[i] \wedge \text{at\_}\ell_2'[i] \\ & \wedge (a[i]' \leftrightarrow a[i]) \wedge (s[i]' \leftrightarrow s[i]) \wedge (w[i]' \leftrightarrow w[i]) \\ & \wedge \neg\text{at\_}\ell_0'[i] \wedge \bigwedge_{l=3}^{7} \neg\text{at\_}\ell_l'[i] \ .\end{aligned}$$

The initial condition states that each process starts in ℓ_1 with boolean variables a, s, w set to *false*.

Our aim is to prove that this algorithm satisfies the mutual exclusion property, i.e., it is never the case that there are two process at ℓ_7 at the same time. □

The algorithm above is a so-called *single-writer* algorithm where each $\mathcal{S}(i, n)$ alters its own state exclusively. But, since without *multi-reader* we would have no communication at all, each $\mathcal{S}(i, n)$ is allowed to read the state of the other processes part of the network, i.e., those with PID $i < n$.

2.3.2 MPS with Global Variables

Szymanski's mutual exclusion algorithm communicates by guarding the transitions of one process with predicates evaluating the states of the other processes. In practice it is often the case that processes communicate over global variables, i.e., a finite set of *multi-writer* variables is used to pass information between the processes.

In our framework we use variables ranging over the natural numbers to model such multi-writer variables. Therefore we extend the set of variables in boolean transition systems.

Definition 2.8 (MPS with global variables)
We first define AF^+ as:

$$\begin{aligned} f \ ::= \ & b[x] \mid x = x \mid x + 1 = x \mid x < x \mid \neg f \mid f \wedge f \mid \forall_n\, x{:}f \mid \exists_n\, x{:}f \\ & a = a \mid a < a \mid a + 1 = a \mid a = x \ , \end{aligned}$$

where x is a *position variable*, a is a *global variable*, and b is a *boolean array variable*. Note that global and position variables can be mixed.

Then, we define boolean transition system with global variables $\mathcal{S}(i,n) = (\mathcal{V}, \mathcal{T}, \Theta)$ as follows:

- $\mathcal{V} = \{b_1, \ldots, b_k\} \cup \{a_1, \ldots, a_l\}$ where each b_j, $1 \leq j \leq k$, is a boolean array of length n and each a_j, $1 \leq j \leq l$, is a global variable ranging over the natural numbers.

- Θ is a formula in $\mathrm{AF}^+(n)$ with $\mathit{free}(\Theta) \subseteq \mathcal{V} \cup \{i\}$ and which describes the set of initial states.

- $\mathcal{T}$ is a finite set of transitions where each $\tau \in \mathcal{T}$ is given by a formula $\rho_\tau \in \mathrm{AF}^+(n)$ such that $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}' \cup \{i\}$ and $\mathcal{V}'$ contains a primed copy for each variable in $\mathcal{V}$.

  Then, a *monadic parameterized system with global variables* $\mathcal{P}$ built from $\mathcal{S}(i,n)$ is the set $\{\|_{l=0}^{m-1} \mathcal{S}(l,m) \mid m \geq 1\}$. □

As a typical example for the usage of such global variables we show a simple mutual exclusion algorithm. Note that not the full range of natural numbers is used but only the range $[0 \ldots n-1]$, i.e., the range is also parameterized by the number of participating processes.

Example 2.9 (Simple ME algorithm)
The parameterized network consists of processes described by the following pseudo code:

$\mathcal{S}(i,n) \equiv$ **loop forever do**
ℓ_0: **await** *true*;
ℓ_1: **if** $\mathit{turn} = i$ **do goto** ℓ_2
else if at_$\ell_0[\mathit{turn}]$ **do** $\mathit{turn} := i$ **od od**;
goto ℓ_1;
ℓ_2: **critical section**;
od

Initially, all processes are at ℓ_0. Location ℓ_2 represents the critical section.

The variable *turn* is meant to indicate which process has the right to enter ℓ_2. If the process having the *turn* is at ℓ_0, *turn* is free and another process at ℓ_1 may take it.

The interesting questions are whether the algorithm satisfies the mutual exclusion property as well as the individual property that each process p reaches its critical section infinitely often.

To come to a positive answer we need fairness conditions as mentioned in Section 2.2.1. The transition τ_{01} from ℓ_0 to ℓ_1, τ_{12} from ℓ_1 to ℓ_2, and τ_{20} from ℓ_2 to ℓ_0 are weak fair whereas the loop τ_{11} from ℓ_1 to ℓ_1 is strong fair, i.e., $\mathcal{J} = \{\tau_{01}, \tau_{12}, \tau_{20}\}$ and $\mathcal{C} = \{\tau_{11}\}$.

It is easy to see how each process $\mathcal{S}(i,n)$ can be described using a boolean variable $\mathrm{at}\_\ell_m[i]$ for each control point ℓ_m. As global variable we have *turn*. Transition τ_2 getting the *turn* is given by the $\mathrm{AF}^+(n)$ formula:

$$\begin{aligned}&\mathrm{at}\_\ell_0[turn] \wedge \forall_n\, j : j \neq i \rightarrow \textstyle\bigwedge_{l=0}^{2} \mathrm{at}\_\ell_l[j] \leftrightarrow \mathrm{at}\_\ell'_l[j]\\ &\quad \wedge \mathrm{at}\_\ell_1[i] \wedge \mathrm{at}\_\ell'_1[i] \wedge turn' = i \wedge \textstyle\bigwedge_{l=0,2} \neg \mathrm{at}\_\ell_l[i]\ .\end{aligned}$$ □

The example above just uses boolean arrays $\mathrm{at}\_\ell_k[]$ to model the flow of control. Except for the global variable *turn* no other variables are used to model the local state of the processes. Also Szymanski's algorithm can be reformulated in such a way and we do so for the sake of brevity. It is clear that a real implementation would not read the program counter of another process, in theory, however, there is no difference to an ordinary variable. The soundness of our simplification can be formally justified with methods presented in [BLO98b, Bau98].

Example 2.10 (Szymanski's algorithm (cont'd))
Szymanski's algorithm modeled exclusively with variables representing the program counter (cf. [ABJN99]):

$\mathcal{S}(i,n) \equiv$ **loop forever do**
- ℓ_0: **noncritical section**
- ℓ_1: **await** $\forall_n\, j : \mathrm{at}\_\ell_0[j] \vee \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_4[j]$
- ℓ_2: **skip**
- ℓ_3: **if** $\exists_n\, j : \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j]$
  - **then goto** ℓ_4
  - **else goto** ℓ_5
- ℓ_4: **await** $\exists_n\, j : \mathrm{at}\_\ell_5[j] \vee \mathrm{at}\_\ell_6[j] \vee \mathrm{at}\_\ell_7[j]$
- ℓ_5: **await** $\forall_n\, j : \neg(\mathrm{at}\_\ell_3[j] \vee \mathrm{at}\_\ell_4[j])$
- ℓ_6: **await** $\forall_n\, j : j < i : \mathrm{at}\_\ell_0[j] \vee \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j]$
- ℓ_7: **critical section**;

od

Then, modeling the example as an MPS gets simpler and more compact.

According to Definition 2.5 the transition from ℓ_1 to ℓ_2 simplifies to:

$$\begin{aligned}&(\forall_n\, j : \mathrm{at}\_\ell_0[j] \vee \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_4[j])\\ &\quad \wedge \forall_n\, j : j \neq i \rightarrow \textstyle\bigwedge_{l=0}^{7} \mathrm{at}\_\ell_l[j] \leftrightarrow \mathrm{at}\_\ell'_l[j]\\ &\quad \wedge \mathrm{at}\_\ell_1[i] \wedge \neg \mathrm{at}\_\ell'_1[i] \wedge \mathrm{at}\_\ell'_2[i] \wedge \neg \mathrm{at}\_\ell'_0[i] \wedge \textstyle\bigwedge_{l=3}^{7} \neg \mathrm{at}\_\ell'_l[i]\ .\end{aligned}$$ □

In such cases we sometimes define the system $\mathcal{S}(i,n)$ as *labeled transition graph* $(\mathcal{L},\mathcal{T},\mathcal{I})$. In the example above we have $\mathcal{L} = \{\ell_0,\ldots,\ell_7\}$, $\mathcal{I} = \{\ell_0\}$, and $\mathcal{T}$ contains transitions from ℓ_i to ℓ_j labeled with guards, e.g. $\forall_n\, j : \text{at\_}\ell_0[j] \vee \text{at\_}\ell_1[j] \vee \text{at\_}\ell_2[j] \vee \text{at\_}\ell_4[j]$. The graph is presented in Figure 2.2. Note, that the transitions are only labeled by guards and do not involve other assignments except for the change in the control location. Since there are no variables except for the control flow we have $\Sigma = \mathcal{L}$. Hence, we also write $(\Sigma,\mathcal{T},\mathcal{I})$ to denote a labeled transition graph.

As an example that does not fall into the classes of asynchronous MPS defined above we present the ticket algorithm [FLBB79, FLBB89].

Example 2.11 (Ticket algorithm)
The ticket algorithm represents also a mutual exclusion algorithm. Processes of the Ticket algorithm

$$\textbf{ticket} \equiv \mathit{num} := 1; \mathit{next} := 1; \mathit{turn} := 0; [\mathcal{S}(0,n) \| \mathcal{S}(1,n) \| \ldots \| \mathcal{S}(n-1,n)]$$

execute the following pseudo code.

$\mathcal{S}(i,n) \equiv$ **loop forever do**
ℓ_1 : **noncritical section**;
ℓ_2 : $\mathit{turn}[i] := \mathit{num}$; $\mathit{num} := \mathit{num} + 1$;
ℓ_3 : **wait** $\mathit{turn}[i] := \mathit{next}$;
ℓ_4 : **critical section**;
ℓ_5 : $\mathit{next} := \mathit{next} + 1$;
od

The algorithm is expected to satisfy the mutual exclusion property, i.e., there exists no computation of **ticket** leading to a state with more than one process being at control location ℓ_4. Figure 2.3 shows a graphical representation.

For each process the algorithm needs a variable *turn* ranging over the natural numbers to control access to the critical section. Hence, we need a parameterized number of unbounded variables. That cannot be expressed as an MPS nor as an MPS with global variables. An MPS with global variables can model the variables *num* and *next* but not the array $\mathit{turn}[0..n-1]$.

A correctness proof of this algorithm can be found in [Lyn96, AW98]. □

2.4 Undecidability

The previous section introduces parameterized networks and defines some restricted classes of them. As it is the purpose of this work to verify those

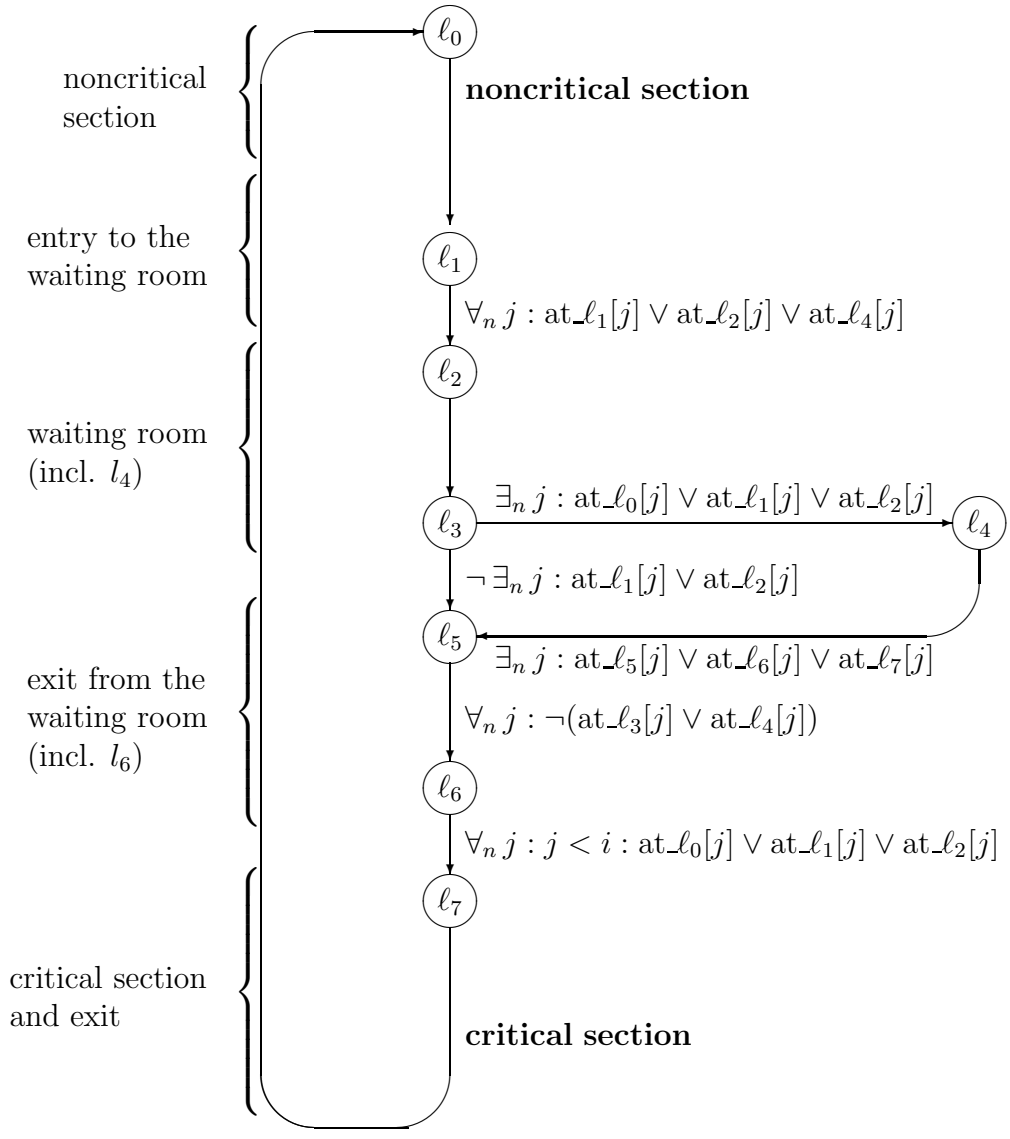


Figure 2.2: Szymanski's algorithm.

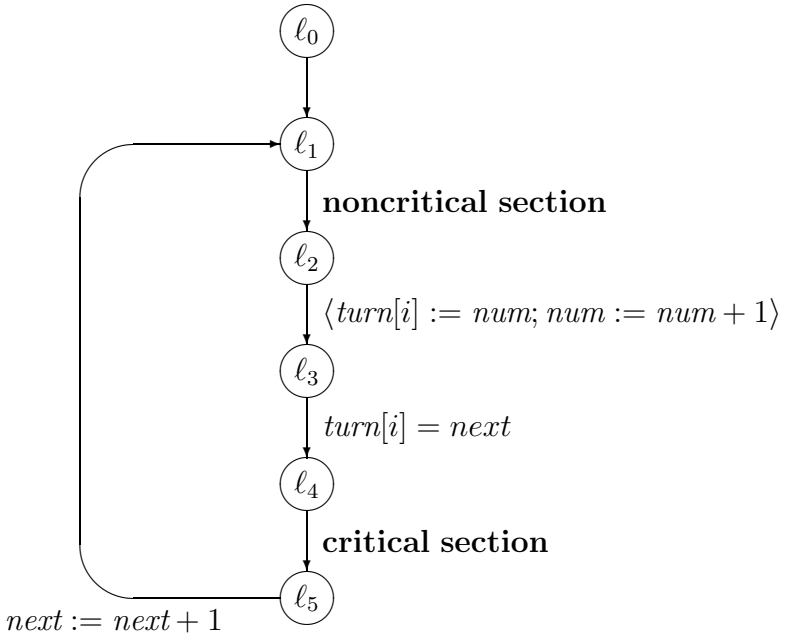


Figure 2.3: Ticket algorithm

networks the first question is whether this problem is decidable or not. Apt and Kozen show in [AK86] that the verification of parameterized networks is undecidable.

Theorem 2.12 (Apt & Kozen)
The verification of MPS is undecidable. Let $\mathcal{P}$ be an MPS. Let

$$\mathcal{Q} := \{(\mathcal{P}, \varphi) \mid \forall n : \mathcal{P}(n) \models \varphi(n)\} \ .$$

Then, $\mathcal{Q}$ is Π_1^0-complete (co-RE).

Proof (Sketch)

$\mathcal{Q} \in co - RE$**:** For each n $\mathcal{P}(n)$ is a finite state system. Hence, $\mathcal{P}(n) \models \varphi(n)$ is in fact decidable. For a parameterized system $\mathcal{P}$ not satisfying φ there exists an m such that $\mathcal{P}(m) \not\models \varphi(m)$. Hence, those systems are recursively enumerable. Thus, systems satisfying φ with all instances are in $co - RE$, i.e., $\mathcal{Q} \in co - RE$.

$\mathcal{Q}$ is Π_1^0-complete:

- Let M be a Turing-machine.

- Define $\mathcal{P}$ such that $\mathcal{P}(n)$ simulates the n first steps of M.
- Moreover, introduce a boolean variable b set to *true* when M terminates.

Then, M does not halt iff $\forall n \geq 0 : \mathcal{P}(n) \models \Box\neg b$. Hence, we have reduced $\mathcal{Q}$ to the halting problem. ■

Despite this negative result automated and semi-automated methods for the verification of restricted classes of parameterized networks have been developed. Our approach presented in this work is based on verification by abstraction [CGL94, DGG94].

2.5 Synchronous Systems

All definitions and conventions made for asynchronous parameterized networks carry over for synchronous systems. The only thing we need for synchronous parameterized networks is a synchronous semantics. To define a synchronous semantics we introduce the synchronous product.

Definition 2.13 (Synchronous product)
Let $\mathcal{S}_i = (\mathcal{V} \cup \{pc_i\}, \mathcal{T}_i, \Theta_i)$, for $i = 1, 2$, be transition systems. The *synchronous product* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is a transition system $\mathcal{S}_1 \oplus \mathcal{S}_2 := \langle \mathcal{V} \cup \{pc\}, \mathcal{T}, \Theta\rangle$, where

- pc ranges over $\mathcal{L} = \mathcal{L}_1 \times \mathcal{L}_2$.
- a transition

  $$\tau = ((\ell_1, \ell_2), \mathit{guard}(\tau_1) \wedge \mathit{guard}(\tau_2), \mathit{effect}(\tau_1) \cup \mathit{effect}(\tau_2), (\ell_1', \ell_2'))$$

  is in $\mathcal{T}$ iff
  - there is a transition $\tau_1 = (\ell_1, \mathit{guard}(\tau_1), \mathit{effect}(\tau_1), \ell_1')$ in $\mathcal{T}_1$, and
  - there is a transition $\tau_2 = (\ell_2, \mathit{guard}(\tau_2), \mathit{effect}(\tau_2), \ell_2')$ in $\mathcal{T}_2$.
- $\Theta \equiv \Theta_1 \wedge \Theta_2$, where pc_i is replaced by $\pi_i(pc)$. □

Similarly, we define now the set of computations of $\mathcal{S}_1 \|_s \mathcal{S}_2$ to be that of $\mathcal{S}_1 \oplus \mathcal{S}_2$. The subscript is omitted in the sequel of this work since the actual kind of computation will be obvious from the context. Hence, for a template process $\mathcal{S}(i, n)$ the synchronous parameterized network $\mathcal{P}$ is defined as $\{\|_{l=0}^{m-1} \mathcal{S}(l, m) \mid m > 1\}$ where $\|$ is the synchronous parallel operator.

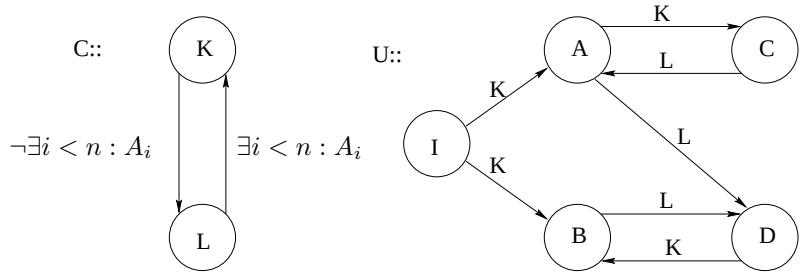


Figure 2.4: Transition for control process C and user process U

Also, the definitions of restricted parameterized networks, i.e., MPS and MPS with global variables, carry over for the synchronous networks. In the synchronous setting an MPS with global variables is often represented as an MPS with an additional control process $C(n)$. Then, we denote the parameterized number of user processes with $U(i, n)$. Hence, the parameterized network is defined as $\mathcal{P} = \{C \| U^m \mid m > 1\}$. When representing the control process and the user process template as labeled transition systems, the parameterized network is characterized by the pair $\langle C, U \rangle$, respectively, $\langle (\mathcal{L}_C, \mathcal{T}_C, \mathcal{I}_C), (\mathcal{L}_U, \mathcal{T}_U, \mathcal{I}_U) \rangle$.

As an example we take a synchronous system from [EN96].

Example 2.14 (Synchronous transition system)
The system is given by the processes $C = (\mathcal{L}_C, \mathcal{T}_C, \mathcal{I}_C), U = (\mathcal{L}_U, \mathcal{T}_U, \mathcal{I}_U)$ where $\mathcal{L}_C = \{K, L\}$, $\mathcal{I}_C = \{K\}$ and $\mathcal{L}_U = \{I, A, B, C, D\}$, $\mathcal{I}_U = \{I\}$. The transition relation is given in Figure 2.4. In the example the guards of the control process check the existence of user processes at certain states whereas the user processes perform their transitions according to the actual control state. □

In this example neither the control process nor the user processes depend on the total number of participating processes. Moreover, the user processes have no notion of PID, hence, the processes act totally interchangeable. For this class of networks there exists a decidability result [EN96]. We show how to achieve this result in our framework in Chapter 9.

A more powerful and interesting class of systems is the one of time-triggered systems.

Definition 2.15 (Time-triggered network)
Define the set $\mathrm{AF}^\star$ of formulae f defined by:

$$f \ ::= \ b[x] \mid c = i \mid \neg f \mid f \wedge f \mid \forall_n x : f \mid \exists_n x : f \ ,$$

where x is a *position variable*, b is a *boolean array variable*, c is the single global variable, and i is a free variable.

Let $U(i, n)$ be a boolean transition system with global variable c where all predicates defining the system are in $\mathrm{AF}^\star$.

Let $C(n)$ be the control process with only instruction $c' = c \oplus_n 1$, where $\oplus_n$ is defined as addition modulo n.

Then, $\mathcal{P} = \langle C(n), U(i, n) \rangle$ is a *time-triggered network*. □

Since the control and the user processes run synchronously, the global variable c is increased modulo n in every step. Hence, each user process i eventually gets $c = i$ and can perform special instructions, see Figure 2.5. An example of such a system is shown in Chapter 10.

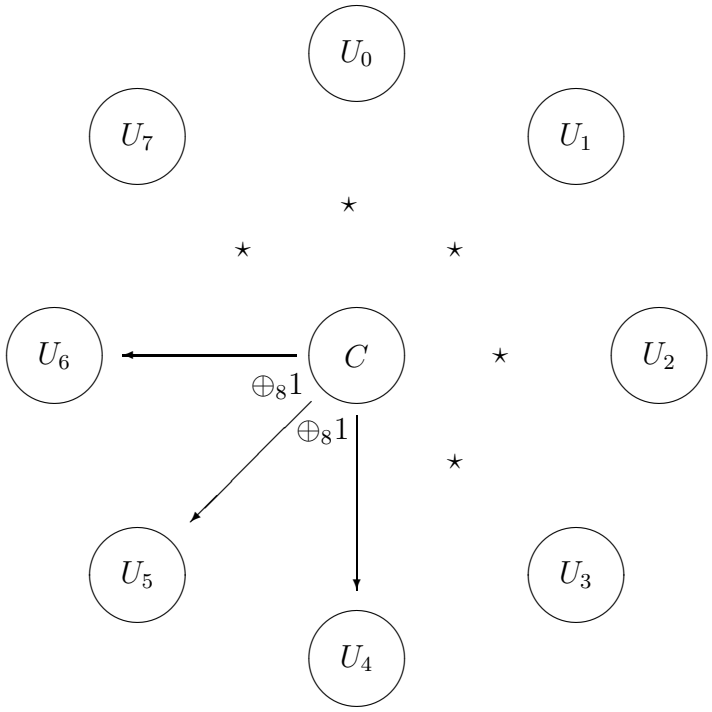


Figure 2.5: Time-triggered systems

2.6 Beyond Finite State Processes

The main scope of this work is the verification of parameterized systems based on finite state processes, maybe with additional global variables. Neverthe-

less, the definition of parameterized networks allows for arbitrary systems to be set in parallel. Example 2.11 shows the ticket algorithm with two global variables $num, next$ where each process has a local unbounded variable $turn$ besides its program counter pc. Between finite state processes and those with infinite state space there are some intermediate steps. Three sources for unboundedness are sketched in the next sections.

2.6.1 Parameterized Systems with Non-atomic Tests

In AF(n) we allow explicitly existential and universal quantification to check for the existence of some process in a certain state or to check whether all processes are in certain states. As already mentioned our semantics assumes transitions labeled by those guards to be atomic, i.e., they represent one step although such a guard has to check a parameterized number of processes instantaneously. On the one hand this can be viewed as an abstraction of a concrete implementation where such a guard can be programmed as a loop checking all processes successively. On the other hand this is a simplification which, especially in the asynchronous setting, ignores a lot of interleavings. Formally, to model non-atomic tests the program counter pc has to depend on the number of participating processes, i.e., we have a variable $pc(n)$ which refines the flow of control for each additional process by introducing intermediate control locations to represent partially completed tests. Hence, the state space of a process depends on n and is unbounded.

2.6.2 Processes with Parameterized State Space

As an example for parameterized networks we mentioned group membership protocols which are sometimes used to figure out which subset of processes is prepared to perform some common computation. The aim of such protocols is that each process that wants to participate in some further computation tries to get a set of all participating processes. I.e., each process keeps a variable ranging over subsets of processes in the network. Since the network is parameterized this set variable is unbounded a priori. In Chapter 10 we show such a group membership protocol. The algorithm presented there is synchronous and time-triggered. We show there how to reduce those local set variables to one global variable. Then, the reduced algorithm falls in the class of time-triggered synchronous networks from Definition 2.15. We then show how to verify those systems.

2.6.3 Multi-dimensional Parameters

As another application of distributed algorithms we mentioned cache coherence protocols. Those algorithms try to guarantee some coherence properties on distributed memory. Naturally, such algorithms are parameterized in two dimensions. First, we allow for an arbitrary number of processes to have their own cache. Moreover, also the address space of available memory is parameterized. Normally, those algorithms handle access to a certain memory address or memory page totally independent from requests for other addresses or pages. In Chapter 10 we present such an algorithm. We generalize formally the independence of both parameters and show how to reduce the algorithm to one which handles just a single address page.

2.7 Bibliographic Notes

To be able to compare other results in the field of parameterized verification to those presented here, we first have to analyze which kind of systems are handled by others and how they can be represented in the framework of MPS. Therefore, we take some representative works in the different areas — asynchronous, synchronous, time-triggered — we have introduced in this chapter.

The work handling parameterized systems closest to our standard MPS is [ABJN99]. They use regular expressions to represent configurations of the network. A transition is modeled by a transducer altering a single position in the configuration string. The guard is characterized by two automata that restrict the left, respectively the right context of the current process. Hence, this is a model for asynchronous process execution. Because of the close connection between WS1S and regular languages [Tho90] the context restrictions established by the automata can be expressed in WS1S. Although we handle almost the same kinds of networks, in contrast to them we use an abstraction-based approach while they use acceleration techniques to compute the exact state space.

Concerning asynchronous systems [EK00] presents decidability results based on restricting the guards controlling the transitions. The classes of systems defined their can be expressed easily as MPS. Moreover, in Chapter 9 we show that our heuristics result in abstractions for those systems that allow their extensive verification. In fact, we derive their decidability result in our framework.

The same holds for a class of synchronous systems proved to be decidable in [EN96]. That class can be represented as restricted, synchronous MPS

and we give abstractions and fairness conditions sufficient to verify them automatically.

[EN95] gives a decidability result for token rings. The model defined there only allows to pass a token without a value. We can model such token rings as MPS and use the successor function in AF(n) to model communication, respectively, token passing to a neighbor process. We show an example in Chapter 8.

Chapter 3

Parameterized Networks as WS1S Systems

In this chapter we show how to model monadic parameterized systems, asynchronous as well as synchronous, as WS1S systems. WS1S is the decidable weak second order logic of one successor. Hence, WS1S systems have second order variables and the transitions are described by WS1S formulae. The idea is that we model a parameterized network, i.e., an infinite family of concurrent systems, as a single WS1S system by identifying each process with its PID and its state by the membership of its PID in certain sets. Since we concentrate on the verification of parameterized systems consisting of finite processes, we can transform such networks to WS1S systems handling a finite set of second-order variables. Also a finite set of global variables (boolean or integer) can be modeled for communication issues. Intuitively, the parameterization gets hidden in the initialization of the WS1S system. The initial condition states the existence of some n such that n processes start in their initial states. The transitions keep this n constant. Hence, when the WS1S system satisfies certain properties, each instance of the parameterized network does so.

WS1S systems itself are normal transition systems. Hence, general verification methods for such transition systems can be applied. Moreover, when dealing with abstraction techniques we can make use of the decidability of WS1S; when the abstraction relation can be characterized by means of WS1S predicates, abstract systems with a finite set of boolean variables can be computed automatically. How to define such abstraction relations is presented in Chapter 5.

3.1 The Decidable Logic WS1S

Let us first briefly recall the definition of weak second order theory of one successor (WS1S for short) [Büc60, Tho90].

Terms of WS1S are built up from the constant 0 and 1st-order variables by applying the successor function $succ(t)$ ("$t+1$"). *Atomic formulae* are of the form b, $t = t'$, $t < t'$, $t \in X$, where b is a boolean variable, t and t' are terms, and X is a set variable (second-order variable). WS1S formulae are built up from atomic formulae by applying the boolean connectives as well as quantification over both 1st-order and 2nd-order variables.

WS1S formulae are interpreted in models that assign finite sub-sets of ω to 2nd-order variables and elements of ω to 1st-order variables. The interpretation is defined in the usual way.

Given a WS1S formula f, we denote by $[\![f]\!]$ the set of models of f. The set of free variables in f is denoted by $free(f)$.

Finally, we recall that by Büchi [Büc60] and Elgot [Elg61] the satisfiability problem for WS1S is decidable. Indeed, the set of all models of a WS1S formula is representable by a finite automaton (see, e.g., [Tho90]).

3.2 WS1S Transition Systems

First, we introduce WS1S transition systems which are transition systems with variables ranging over finite sub-sets of ω. Then, we show how they can be used to represent parameterized networks as introduced in Chapter 2.

Definition 3.1 (WS1S transition system)
A *WS1S transition system* $\mathcal{W} = (\mathcal{V}, \mathcal{T}, \Theta)$ is given by the following components:

- $\mathcal{V} = \{X_1, \ldots, X_k\}$: A finite set of second order variables where each variable is interpreted as a finite set of natural numbers.
- Θ: A WS1S formula with $free(\Theta) \subseteq \mathcal{V}$ describing the initial condition of the system.
- $\mathcal{T}$: A finite set of transitions where each $\tau \in \mathcal{T}$ is represented as a WS1S formula $\rho_\tau(\mathcal{V}, \mathcal{V}')$, i.e., $free(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}'$. Again, we use the primed version of the variables to denote the post-state.

The computations of $\mathcal{W}$ are defined as usual. Moreover, let $[\![\mathcal{W}]\!]$ denote the set of computations of $\mathcal{W}$. □

Except for the special finite variable *pc* which we have introduced to model control flow explicitly, the definition above matches Definition 2.1. Only the transitions are restricted to those representable as WS1S formulae. For parameterized networks where each process is finite state and the arithmetic used on PIDs and possible global integer variables is simple (no multiplication, addition only with constants) this is often the case.

In the sequel we show how to model asynchronous or synchronous parameterized systems as WS1S systems. For those networks not restricted in the way described above, first of all not consisting of finite state processes, we show verification approaches in Chapter 10.

3.3 Modeling Asynchronous Systems

In Chapter 2 we introduced two kinds of asynchronous parameterized networks based on finite state processes, without or with additional global variables. Here we show how to model those networks as a single WS1S system. We prove the WS1S translation with a fixed value for n to be bisimilar with the network instance of size n.

As mentioned in Chapter 2 the methods generalize to networks of the form $U_1^{n_1} \| \ldots \| U_m^{n_m}$. Therefore, the translation defined next has to be adapted in a straightforward manner.

We define a translation that maps an asynchronous MPS (without global variables) to a bisimilar WS1S system.

We fix a second-order variable P that is used to model the set of processes that are part of the network. Hence, P contains the natural numbers from 0 up to $n-1$. The translation from MPS to WS1S systems uses a function tr from AF(n) into WS1S. This function replaces all occurrences of atomic subformulae of the form $b[i]$ in an AF(n)-formula by $i \in B$, all n by $\max(P)+1$[1], and λ_n by λ_P where λ is one of the quantifiers $\forall$ or $\exists$. Again, $\forall_P : f$ abbreviates $\forall x : x \in P \rightarrow f$ and $\exists_P x : f$ abbreviates $\exists x : x \in P \wedge f$.

Definition 3.2 (Translation of boolean to WS1S systems)
Consider an MPS $\mathcal{P}$ built from $\mathcal{S}(i,n)$ where $\mathcal{S}(i,n) = (\mathcal{V}, \Theta, \mathcal{T})$. Define a WS1S system $(\widetilde{\mathcal{V}}, \widetilde{\mathcal{T}}, \widetilde{\Theta})$ by constructing the variable set, the initial condition, and the transitions as follows:

- For each boolean array b_k in $\mathcal{V}$, $\widetilde{\mathcal{V}}$ contains the variable B_k. Additionally, $\widetilde{\mathcal{V}}$ contains the set variable P.
- Let $\widetilde{\Theta}$ be $\exists n : P = \{0, \ldots, n-1\} \wedge \bigcup_{l=1}^{k} B_l \subseteq P \wedge (\forall_P i : tr(\Theta))$.

[1]It is not difficult to check that $\max(P)$ can be expressed in WS1S.

- Let $\widetilde{\mathcal{T}}$ be the set $\{\exists_P i : tr(\rho_\tau) \wedge P = P' \wedge \bigcup_{l=1}^{k} B'_l \subseteq P' \mid \tau \in \mathcal{T}\}$. □

We denote the above transformation of an MPS $\mathcal{P}$ by $Tr(\mathcal{P})$.

Example 3.3 (Szymanski's algorithm as WS1S system)
The translation of Example 2.10 into a WS1S system introduces a set variable P and set variables $\mathrm{At\_}\ell_0, \ldots, \mathrm{At\_}\ell_7$. According to the definition of Tr we have as initial condition $\widetilde{\Theta}$

$$\exists n : P = \{0, \ldots, n-1\} \wedge \bigcup_{l=0}^{7} \mathrm{At\_}\ell_l \subseteq P \wedge \forall_P i : (i \in \mathrm{At\_}\ell_0 \wedge \bigwedge_{l=1}^{7} i \notin \mathrm{At\_}\ell_l) \ .$$

The translation of the AF(n) formula characterizing the transition from ℓ_1 to ℓ_2 presented in Example 2.10 yields (up to some simplifications)

$$\begin{array}{l}(\forall_P j : j \in \mathrm{At\_}\ell_0 \cup \mathrm{At\_}\ell_1 \cup \mathrm{At\_}\ell_2 \cup \mathrm{At\_}\ell_4) \\ \quad \wedge \forall_P j : j \neq i \rightarrow \bigwedge_{l=0}^{7} j \in \mathrm{At\_}\ell_l \leftrightarrow j \in \mathrm{At\_}\ell'_l \\ \quad \wedge i \in \mathrm{At\_}\ell_1 \wedge i \notin \mathrm{At\_}\ell'_1 \wedge i \in \mathrm{At\_}\ell'_2 \wedge i \notin \mathrm{At\_}\ell'_0 \wedge \bigwedge_{l=3}^{7} i \notin \mathrm{At\_}\ell'_l\end{array}$$

which, using the invariant $\bigcup_{l=1}^{k} B_l \subseteq P$, can be simplified to

$$\begin{array}{l}(\forall_P j : j \in \mathrm{At\_}\ell_0 \cup \mathrm{At\_}\ell_1 \cup \mathrm{At\_}\ell_2 \cup \mathrm{At\_}\ell_4) \\ \quad \wedge \mathrm{At\_}\ell'_0 = \mathrm{At\_}\ell_0 \wedge \bigwedge_{l=3}^{7} \mathrm{At\_}\ell_l = \mathrm{At\_}\ell'_l \\ \quad \wedge \mathrm{At\_}\ell'_1 = \mathrm{At\_}\ell_1 \setminus \{i\} \wedge \mathrm{At\_}\ell'_2 = \mathrm{At\_}\ell_2 \cup \{i\} \ .\end{array}$$ □

In order to state the relationship between an MPS $\mathcal{P}$ and its translation, we introduce a function h relating the states of both systems. Let $\mathcal{P}$ be an MPS built from $\mathcal{S}(i,n) = (\mathcal{V}, \Theta, \mathcal{T})$ with $\mathcal{V} = \{b_1, \ldots, b_k\}$. Let m be a natural number.

Let Σ_m denote the state space of $\mathcal{P}(m)$ and let $\widetilde{\Sigma}$ denote the set of interpretations $\tilde{s}$ of $\widetilde{\mathcal{V}}$ such that $\tilde{s}(X) \subseteq \tilde{s}(P)$, for each $X \in \widetilde{\mathcal{V}}$. Then, define $h_m : \Sigma_m \rightarrow \widetilde{\Sigma}, s \mapsto \tilde{s}$ by $\tilde{s}(P) = \{0, \ldots, m-1\}$ and $\tilde{s}(B_j) = \{l < m \mid s(b_j[l]) = \mathit{true}\}$, for every $1 \leq j \leq k$. Then, we define $h = \bigcup_{m \in \omega} h_m$. Notice that h is a bijection.

Our aim is to establish a connection between an MPS and its translation. To show properties for an MPS via proving those properties for the translated WS1S system we have to relate the computations of each system. On the level of states $h(s)$ gives us an interpretation for sets $P, B_1, \ldots, B_k$, where s is a state interpreting boolean arrays $b_1, \ldots, b_k$.

The following lemma shows that h is consistent with the translation tr from AF(n) into WS1S.

Lemma 3.4 (Correctness of translation)
Let f be a formula in AF(n) *with free$(f) \subseteq \mathcal{V} \cup \mathcal{V}' \cup \{i\}$ and let h be the function introduced above. Then, for all $m \in \omega$, for all $s, s' \in \Sigma_m$, we have:*

$$(s, s'), i \models f \quad \textit{iff} \quad (h(s), h(s')), i \models tr(f) \ . \qquad \square$$

Using this lemma we can prove the following theorem that justifies our verification method given in Section 5. The theorem states that $\mathcal{P}(m)$ is bisimilar to $Tr(\mathcal{P})$ when we initialize P to $\{0, \ldots, m-1\}$.

Theorem 3.5 (Relating Boolean and WS1S systems)
Let $\mathcal{P}$ be an MPS built from $\mathcal{S}(i, n)$ and $m \in \omega$. Then, h is a bisimulation between $\mathcal{P}(m)$ and $Tr^*(\mathcal{P})$, where $Tr^*(\mathcal{P})$ is obtained from $Tr(\mathcal{P})$ by taking as initial condition $\widetilde{\Theta} \equiv P = \{0, \ldots, m-1\} \wedge \bigcup_{l=1}^{k} B_l \subseteq P \wedge \forall_P i : tr(\Theta)$.

Proof Consider s_0 to be an initial state of $\mathcal{P}(m)$, i.e., $s_0 \models \Theta$. With Lemma 3.4 it follows that

$$h(s_0) \models P = \{0, \ldots, m-1\} \wedge \bigcup_{l=1}^{k} B_l \subseteq P \wedge (\forall_P i : tr(\Theta)) \ .$$

Vice versa, for any initial state $\tilde{s}_0$ of some computation in $[\![Tr^*(\mathcal{P})]\!]$ the state $h^{-1}(\tilde{s}_0)$ is an initial state of $\mathcal{P}$.

With the same argumentation we can show for $s, s' \in \Sigma_m$ and $\tilde{s}, \tilde{s}' \in \widetilde{\Sigma}$ with $\tilde{s} = h(s)$ that for any $\tau \in \mathcal{T}$,

- if s' is a τ-successor of s then $h(s')$ is a $tr(\rho_\tau)$-successor of $\tilde{s}$, and
- if $\tilde{s}'$ is an $tr(\rho_\tau)$-successor of $\tilde{s}$ then $h^{-1}(\tilde{s}')$ is a τ-successor of s.

Hence, both systems are bisimilar. ■

Using Theorem 3.5, we can prove the following:

Corollary 3.6 (Equivalence of $\mathcal{P}$ and its translation)
Let $\mathcal{P}$ be an MPS built from $\mathcal{S}(i, n)$. Then, lifting h to computations, we have a bijection between $\bigcup_{m \in \omega} [\![\mathcal{P}(m)]\!]$ and $[\![Tr(\mathcal{P})]\!]$. $\square$

This result allows us to verify the WS1S translation instead of the original MPS. Correctness results obtained for the WS1S systems can be carried over to the MPS.

3.3.1 MPS with Global Variables

Strictly spoken, variables in WS1S range over subsets of ω. In practice boolean and variables ranging over the natural numbers can be used anyhow, tools that decide WS1S support such variables, internally they are represented as sets. Natural numbers are represented as the corresponding singleton and booleans are modeled as sets being empty or nonempty. Hence, the translation Tr of MPS into WS1S systems can be easily adapted to a setting with global variables. Indeed, we can use the same translation tr from AF(n) to WS1S. The translation of global variables is the identity, we just have to extend the set of variables to include them also.

Definition 3.7 (Translation of MPS with global variables)
Consider an MPS with global variables $\mathcal{P}$ built from $\mathcal{S}(i,n) = (\mathcal{V}, \Theta, \mathcal{T})$ where $\mathcal{V}$ contains the global variables $a_1, \ldots, a_m$ ranging over the natural numbers. Define a WS1S system $(\widetilde{\mathcal{V}}, \widetilde{\mathcal{T}}, \widetilde{\Theta})$ as follows:

- For each boolean array b_k in $\mathcal{V}$, $\widetilde{\mathcal{V}}$ contains the variable B_k. All global variables are in $\widetilde{\mathcal{V}}$, i.e., $\{a_1, \ldots, a_m\} \subseteq \mathcal{V}$. Additionally, $\widetilde{\mathcal{V}}$ contains the set variable P.
- Let $\widetilde{\mathcal{T}}$ be the set $\{\exists_P i : tr(\rho_\tau) \wedge P = P' \wedge \bigcup_{l=1}^{k} B'_l \subseteq P' \mid \tau \in \mathcal{T}\}$.
- Let $\widetilde{\Theta}$ be $\exists n : P = \{0, \ldots, n-1\} \wedge \bigcup_{l=1}^{k} B_l \subseteq P \wedge (\forall_P i : tr(\Theta))$.

Since the definition resemble Definition 3.2 except for the variable set $\mathcal{V}$ and since there is no danger of confusion, we denote the above transformation of an MPS system $\mathcal{P}$ as well by $Tr(\mathcal{P})$. □

Because we have not changed the definition of tr it can be checked easily that Lemma 3.4 can be stated similarly for AF$(n)^+$.

Lemma 3.8 (Translation of AF$^+(n)$)
Let f be a formula in AF$^+(n)$ *with free$(f) \subseteq \mathcal{V} \cup \mathcal{V}' \cup \{i\}$ and define $h^+ = \bigcup_{m\in\omega} h^+_m$ where $h^+_m : \Sigma_m \to \widetilde{\Sigma}, s \mapsto \tilde{s}$ by $\tilde{s}(P) = \{0, \ldots, m-1\}$, $\tilde{s}(B_j) = \{l < m \mid s(b_j[l]) = true\}$, for every $1 \le j \le k$, and $\tilde{s}(a_j) = s(a_j)$, for every $1 \le j \le m$. Then, we define $h^+ = \bigcup_{m\in\omega} h^+_m$.*

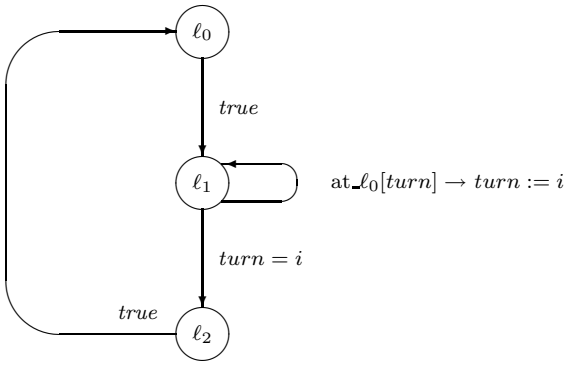
Then, for all $m \in \omega$, for all $s, s' \in \Sigma_m$, we have:

$$(s, s'), i \models f \quad \textit{iff} \quad (h^+(s), h^+(s')), i \models tr(f) \ .$$ □

Then, we can state and prove analogously Theorem 3.5 and Corollary 3.6.

As an example we show the translation of the simple mutual exclusion algorithm introduced in Chapter 2.

Example 3.9 (Translation of Simple ME algorithm)
The algorithm presented as pseudo code in Example 2.9 can also be characterized graphically as follows:



To represent this network as a WS1S system we introduce three set variables $\text{At\_}\ell_0, \text{At\_}\ell_1, \text{At\_}\ell_2$ corresponding to the control locations, the first-order variable $turn$, and a set variable P representing the set of processes being part of the network.

For the sake of illustration, we show the characterization $\rho_{\tau_1}(\mathcal{V}, \mathcal{V}', i)$ of the loop transition τ_1:

$$\begin{aligned}\rho_{\tau_1}(\mathcal{V}, \mathcal{V}', i) \;\equiv\; & i \in \text{At\_}\ell_1 \wedge turn \in \text{At\_}\ell_0 \\ & \wedge i = turn' \wedge i \in \text{At\_}\ell_1' \wedge \bigwedge_{k=0,2} i \notin \text{At\_}\ell_k \\ & \wedge (\forall_P j : j \neq i \rightarrow \bigwedge_{k=0,1,2} (j \in \text{At\_}\ell_k \leftrightarrow j \in \text{At\_}\ell_k')) \\ & \wedge P = P' \wedge \bigcup_{k=0,1,2} \text{At\_}\ell_k' \subseteq P' \;\; . \end{aligned}$$

□

3.4 Modeling Synchronous Systems

For parameterized synchronous systems we have also defined the notion of MPS. Having the translation of asynchronous systems in mind, it is straightforward to define a translation operator Tr for synchronous networks. To model a synchronous semantics one has to guarantee a simultaneous execution of all processes.

We give a WS1S translation for synchronous networks that are represented as a unique control process C which may be omitted and an arbitrary number of user processes U, i.e., $\mathcal{P}(m) = C \| U_0 \| \ldots \| U_m$. In a synchronous setting it is often useful to have an extra process for administration purposes

or to model some common resources for the other processes. We assume the control process C and the process template U to be given as a labeled transition graph. To obtain U_i we subscribe the states of U with i. Transitions in C and U are labeled with guards where the states of the processes serve as atomic propositions.

The modeling of the infinite family of system instances $\mathcal{P}(m)$ as WS1S systems is straightforward. Again, the infinite family is represented as one transition system. The parameterization disappears in the initial condition which states that there exists some n which gives the number of involved user processes.

Definition 3.10 (Translation of synchronous systems)
Let $\mathcal{P} = \langle(\Sigma_C, \mathcal{T}_C, \mathcal{I}_C), (\Sigma_U, \mathcal{T}_U, \mathcal{I}_U)\rangle$ be a parameterized system given by its control and user processes. Define a WS1S system $\mathcal{W}$ as follows:

Set of variables: For each $c \in \Sigma_C$, $\mathcal{V}$ contains a boolean variable c and for each $u \in \Sigma_U$, $\mathcal{V}$ contains a set variable X_u intended to hold those indices i of user processes currently in state u_i. Moreover, a set variable U is added to hold the whole set of constituent user processes.

Initial condition: Initially, we have to constrain the WS1S system as follows to model an initial state of the original system:

$$\begin{aligned}&\exists n : U = \{0, \ldots, n-1\} \wedge \textstyle\bigvee_{c \in \mathcal{I}_C}(c \wedge \bigwedge_{c' \in \Sigma_C \setminus \{c\}} \neg c')\\ &\wedge U = \textstyle\bigcup\{X_u \mid u \in \mathcal{I}_U\} \wedge \bigwedge_{u,v \in \mathcal{I}_U} X_u \cap X_v = \emptyset\\ &\wedge \emptyset = \textstyle\bigcup\{X_u \mid u \in \Sigma_U \setminus \mathcal{I}_U\}\end{aligned}$$

Transition relation: The transition relation is given as one predicate connecting the pre- and post-state of the system. The post-state is represented by primed versions of the variables in $\mathcal{V}$:

$$\begin{aligned}U = U' \quad \wedge \quad & \bigvee_{c_i \xrightarrow{g_C} c_j \in \mathcal{T}_C} c_i \wedge tr(g_C) \wedge c_j' \wedge \bigwedge_{c_j \neq c_l \in \Sigma_C} \neg c_l'\\ \wedge \quad & \forall i \in U : \bigvee_{u \xrightarrow{g_U} v \in \mathcal{T}_U} (i \in X_u \wedge tr(g_U) \wedge i \in X_v'\\ & \qquad \wedge i \notin \textstyle\bigcup\{X_w' \mid v \neq w \in \Sigma_U\})\end{aligned}$$

Here, tr denotes the translation of u_i for $u \in \Sigma_U$ into $i \in X_u$.

The definition of $\mathcal{W}$ is well-defined since the predicates defining the initial condition and the transition relation are indeed expressed as WS1S formulae. The set of computations $[\![\mathcal{W}]\!]$ is defined as usual and $\Sigma_{\mathcal{W}}$ is the state space of $\mathcal{W}$. □

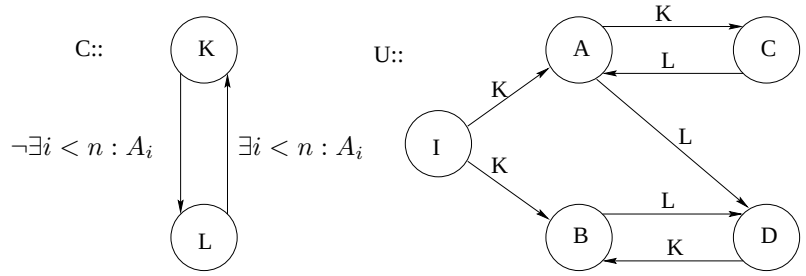


Figure 3.1: Graphical representation of C and U.

As an example we take again the synchronous system from [EN96] presented earlier as Example 2.14.

Example 3.11 (Synchronous WS1S system)
The transition relation is given in Figure 3.1. To model the control process we need two boolean variables c_K, c_L. To model the user processes, $\mathcal{V}$ contains 5 set variables X_I, X_A, X_B, X_C, X_D. For clarity we name them I, A, B, C, D. As Θ we get by translation

$$\begin{aligned} & \exists n : U = \{0, \ldots, n-1\} \wedge (c_K \wedge \neg c_L) \\ \wedge \quad & U = I \wedge \emptyset = \bigcup \{A, B, C, D\} \ . \end{aligned}$$

The transition relation is given by

$$\begin{aligned} U = U' \quad \wedge \quad & (c_K \wedge \neg\exists i < n : i \in A \wedge c'_L \wedge \neg c'_K) \vee \\ & (c_L \wedge \exists i < n : i \in A \wedge c'_K \wedge \neg c'_L) \\ \wedge \quad & \forall i \in U : (i \in I \wedge c_K \wedge i \in A' \wedge i \notin I' \cup B' \cup C' \cup D') \vee \\ & \qquad (i \in I \wedge c_K \wedge i \in B' \wedge i \notin I' \cup A' \cup C' \cup D') \vee \\ & \qquad (i \in A \wedge c_K \wedge i \in C' \wedge i \notin I' \cup A' \cup B' \cup D') \vee \\ & \qquad (i \in D \wedge c_K \wedge i \in B' \wedge i \notin I' \cup A' \cup C' \cup D') \vee \\ & \qquad (i \in B \wedge c_L \wedge i \in D' \wedge i \notin I' \cup A' \cup B' \cup C') \vee \\ & \qquad (i \in A \wedge c_L \wedge i \in D' \wedge i \notin I' \cup A' \cup B' \cup C') \vee \\ & \qquad (i \in C \wedge c_L \wedge i \in A' \wedge i \notin I' \cup B' \cup C' \cup D') \ . \end{aligned}$$

□

As the translation of such synchronous systems to WS1S systems is quite intuitive we can easily prove the following lemma.

Lemma 3.12 ($\langle C, U \rangle$ and $\mathcal{W}$ are bisimilar)
Let $\mathcal{P} = \langle (\Sigma_C, \mathcal{T}_C, \mathcal{I}_C), (\Sigma_U, \mathcal{T}_U, \mathcal{I}_U) \rangle$ be a parameterized synchronous system and let $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ be the translated WS1S system. For each initial valuation $U = \{0, \ldots, m-1\}$, $\mathcal{W}$ behaves bisimilar to $\mathcal{P}(m)$.

PROOF Note, once the computation of the WS1S system is started, U is invariant. Hence, let $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ start initially with $U = \{0, \ldots, n-1\}$. We construct a bisimulation $h : \Sigma_C \times \Sigma_U^n \longrightarrow \Sigma_{\mathcal{W}}, s = (c_j, u_{j_0}, \ldots, u_{j_{n-1}}) \mapsto h(s)$ between $\mathcal{P}(n)$ and $\mathcal{W}$.

Define h such that $h(s)(U) = \{0, \ldots, n-1\}$, $h(s)(c) \leftrightarrow \pi_1(s) = c$, and $j \in h(s)(X_u) \leftrightarrow \pi_j(s) = u$ with π_i being the i-th projection.

Let $s_0 \in \mathcal{I}_C \times \mathcal{I}_u^n$ be an initial state, i.e., $s_o = (c, u_{j_0}, \ldots, u_{j_{n-1}})$ for $c \in \mathcal{I}_C$ and $u_{j_0}, \ldots, u_{j_{n-1}} \in \mathcal{I}_U$. Hence, $h(s_0) \models c$ and $h(s_0) \models \bigwedge_{c' \in \Sigma_C \setminus \{c\}} \neg c'$ according to the definition of h. Moreover, every X_u with $u \notin \mathcal{I}_U$ is empty and $\bigcup_{u \in \mathcal{I}_U} X_u$ is a partition of U. So, we have $h(s_0) \models \Theta$.

Now, let $s, s' \in \Sigma_C \times \Sigma_U^n$ such that s' is a successor state of s in $\mathcal{P}_n$. I.e., there exists a transition in $\mathcal{T}_C$ with $\pi_1(s) \xrightarrow{g_C} \pi_1(s')$ and there exists for each $i \in \{0, \ldots, n-1\}$ a transition in $\mathcal{T}_U$ with $\pi_i(s) \xrightarrow{g_i} \pi_i(s')$. Moreover, we have that $s \models g_C \wedge \bigwedge_{i \in \{0,\ldots,n-1\}} g_i$. According to Definition 3.10 we have that indeed $h(s), h(s')$ models the transition predicate of $\mathcal{W}$.

Since h is a bijection between $\mathcal{P}(n)$ and $\mathcal{W}$ where $U = \{0, \ldots, n-1\}$ is fixed, the same argumentation holds in the other direction. I.e., each initial state in $\mathcal{W}$ with $U = \{0, \ldots, n-1\}$ is an initial state in $\mathcal{P}(n)$ and if $\tilde{s}, \tilde{s}'$ models the transition predicate of $\mathcal{W}$ then we know that $\tilde{s}, \tilde{s}'$ are in the image of h for some s, s' and s' is indeed a successor of s in $\mathcal{P}(n)$. Hence, we have established bisimulation. ∎

Part II

Verification of WS1S Systems

Chapter 4

Abstraction-Based Verification

In the previous part we have defined several classes of parameterized networks and shown their translation to WS1S systems. We have shown undecidability even for the restricted class of MPS. For the undecidability proof of Theorem 2.12 it suffices to chose the simplest kind of property, reachability of a certain state, to get the undecidability result. Hence, we cannot find a sound and complete method for the verification of MPS. An automatic verification approach has to give up one of the two requirements, soundness or completeness. Of course we want to be sure on the results of our verification method, a positive outcome means that the MPS indeed satisfies the given property. Therefore, we cannot guarantee that every MPS that satisfies a given property is proven correct automatically by our method. In such cases the method delivers so-called *false negatives*. Those approaches are sometimes called *semi-automatic*. They have to be complemented by deductive verification methods in order to verify the general class of MPS.

Our method is abstraction-based. For the abstraction-based verification approach we give the basic definitions and results which allow to transfer verification results from the abstract to the concrete system. Intuitively, an abstraction over-approximates the concrete system, if the computations of the abstract system then lie within a desired set of traces, the more constraint concrete computations do so as well. If the abstract system fails to satisfy a desired property, we cannot state anything about the concrete system. Clearly, for automatically constructed and verified abstract systems the method is semi-automatic. We discuss the main problems for that approach; finding the abstraction relation, refining the abstraction to exclude false negatives, and model-checking the abstract system.

First, we introduce classes of properties that are of interest for algorithms, sequential or distributed, in order to characterize their correctness. We motivate the classes of safety and liveness properties by illustrating where they

arise in the verification of traditional sequential algorithms. Then, we define a temporal logic to express the properties of interest for parameterized networks.

4.1 Properties of Interest

The properties that can be and have to be verified in order to establish correctness of an algorithm, sequential or running in a distributed network, are manifold. Moreover, the properties vary according to the purposes an algorithm is designed for. In standard algorithm courses correctness means that the algorithm produces the intended outcome for the given inputs. E.g., sorting algorithms are expected to return sorted lists or a program approximating the number π should return π up to a given deviation ϵ. Classical algorithms to solve such problems are sequential programs. Their correct behavior is called partial correctness and their verification is well studied [Flo67, Hoa69, Apt81, HJ89]. Partial correctness means that whenever the algorithm terminates it provides the correct result. The method presented in [Flo67] is called Floyd's inductive assertion method. It is based on finding an assertion network assigning to each location a predicate. At the initial location the predicate has to be an implication of the precondition. For each transition one has to show that whenever the assertion at the source location holds, the predicate at the target location can be established after firing the transition. Therefore, the method is called inductive assertion method. Having fulfilled the proof obligations the assertions are invariants at their corresponding locations.

Naturally, another question arises: is the algorithm going to terminate. Also for this problem [Flo67] gives here a solution known as Floyd's Well-foundedness method proving convergence via ranking functions. Informally, the ranking functions count the steps needed to terminate. It has to be shown that such functions decrease during the execution in order to establish termination.

In our work we focus on concurrent and distributed algorithms parameterized in the number of participating processes. Those algorithms or protocols organize communication and coordination between the processes, or they are used to gain information about the processes composing the network and its topology [Lyn96, AW98]. They are not expected to terminate. Nevertheless, the verification also splits into two parts. First, is the accumulated information correct and second does every process, intended to have the information, eventually gets it. The first class belongs to the so-called class of *safety properties* while the second one represents *liveness properties*. Informally, a safety

property states that "never something bad is going to happen". A liveness property states that "eventually something good is going to happen".

To formalize these notions we first have to express *something good* and *something bad*. To do so, we need an assertional language $\mathcal{A}$ to characterize certain configurations of the network. To be able to distinguish every two different configurations we need the state of each process as proposition in $\mathcal{A}$, hence we have $\mathrm{AF}(n) \subseteq \mathcal{A}$ for every n. We define a property P as a set of finite or infinite sequences of propositions.

4.1.1 Safety Properties

We adopt the definition of [Lam76, AS85, Lyn96] to formalize what we mean by "nothing bad happens". We make three assumptions about "bad" things. First, nothing "bad" can happen before anything else has happened, i.e., the empty sequence is always in the set of sequences defining a safety property. Second, if nothing "bad" has happened in a sequence, it is also absent in any prefix. Finally, if something "bad" happens, this is caused by a particular event (occurrence of a particular proposition) in the sequence. Hence we presume limit-closure for safety properties.

Definition 4.1 (Safety property)
Let P be a property given as a set of traces over Σ with

1. P is not empty.
2. P is *prefix-closed*, i.e., for every $\sigma \in P$ and every prefix σ' of σ, $\sigma' \in P$.
3. P is *limit-closed*, i.e., for every sequence $\sigma_1, \sigma_2, \ldots \in P^\omega$ where σ_i is a prefix of σ_{i+1} for every i, the unique limit σ is also in P.

Then, we call P a safety property. □

To prove that a concrete system $\mathcal{S}$ satisfies a property P, one has to show $[\![\mathcal{S}]\!] \subseteq P$. For safety properties the second condition of the definition above allows us to prove this entailment by showing that all finite prefixes of computations are indeed in P. Deductively, this is done by induction over the length of a computation. Since fairness conditions only exclude infinite traces from computations of $[\![\mathcal{S}]\!]$, for fair systems $\mathcal{S}^{\mathcal{F}}$ it is equivalent to prove $[\![\mathcal{S}]\!] \subseteq P$ instead of $[\![\mathcal{S}^{\mathcal{F}}]\!] \subseteq P$.

4.1.2 Liveness Properties

The informal definition of a liveness property P implies that at any point in a sequence of P it cannot be excluded that the "good" thing is still going to happen. This results in the formal definition that P is a liveness property if every finite sequence σ has an extension in P.

To establish liveness properties one has to take fairness conditions into account, i.e., one has to show $[\![\mathcal{S}^{\mathcal{F}}]\!] \subseteq P$. To prove such claims methods based on *temporal logic* have shown to work well in practice.

4.1.3 Linear-Time Temporal Logic

All the trace properties of parameterized systems we are interested in can be specified in linear-time temporal logic [Pnu77]. In fact, we restrict to future formulae. Linear-time temporal logic allows us to specify properties over the set of all computations of a system. We use the same assertional language $\mathcal{A}$ as introduced above. We refer to a formula in the assertion language as a state formula, or simply as an assertion.

A temporal formula is constructed out of state formulae by applying boolean operators $\neg$ and $\wedge$ (the other boolean connectives can be defined from these), and the basic temporal operators $\bigcirc$ and $\mathcal{U}$.

A model for a temporal formula φ is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$ such that the s_i give rigid interpretations to the variables n, respectively, P [MP92, MP95b, MP81]. Given a model σ we define the notion of φ holding in σ at position j, $j \geq 0$, denoted by $(\sigma, j) \models \varphi$ inductively on the formula:

$$\text{If } \varphi \text{ is a state formula, } (\sigma, j) \models \varphi \text{ if and only if } s_j \models \varphi$$

$$\begin{array}{lll}
(\sigma, j) \models \neg\varphi & \text{iff} & (\sigma, j) \not\models \varphi \\
(\sigma, j) \models \varphi \wedge \psi & \text{iff} & (\sigma, j) \models \varphi \text{ and } (\sigma, j) \models \psi \\
(\sigma, j) \models \bigcirc\varphi & \text{iff} & (\sigma, j+1) \models \varphi \\
(\sigma, j) \models \varphi\,\mathcal{U}\,\psi & \text{iff} & (\sigma, k) \models \psi, \text{ for some } k, \text{ with } j \leq k, \\
 & & \quad \text{and for all } i \text{ with } j \leq i < k, (\sigma, i) \models \varphi
\end{array}$$

I.e., a state formula φ can be evaluated locally using the interpretation given by s_j to the free variables appearing in φ.

Other temporal operators can be defined as abbreviations, e.g., $\Diamond\varphi \equiv \mathit{true}\ \mathcal{U}\ \varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

Now we are able to define whether a system $\mathcal{S}$ satisfies a property φ or not. We say a formula φ is $\mathcal{S}$-*valid* if for each computation σ of $\mathcal{S}$, $(\sigma, 0) \models \varphi$, and denote it with $\mathcal{S} \models \varphi$.

How to specify properties that one expects to hold for a system and how to be sure that one catches all the properties that are necessary to establish the overall correctness of a system is no trivial task [Wol86]. [MP92] gives an introduction to specification of reactive systems.

For the examples presented so far, Szymanski's algorithm and the Simple ME algorithm, we already stated the correctness properties informally. They basically require mutual exclusion to be guaranteed as well as accessibility of the critical section. We are now able to define those properties formally as LTL formulae.

Example 4.2 (Basic LTL properties of ME algorithms)
In the pseudo-code of Szymanski's algorithm 2.10, the critical section is represented by the location ℓ_7. Mutual exclusion requires that there are never two processes in the critical section at the same time. The existence of two processes at ℓ_7 is expressible by the state formula $\exists_n\, p, q, p \neq q : \mathrm{at}\_\ell_7[p] \wedge \mathrm{at}\_\ell_7[q]$. Hence, the property of mutual exclusion is characterized by the LTL formula

$$\Box\neg(\exists_n\, p, q, p \neq q : \mathrm{at}\_\ell_7[p] \wedge \mathrm{at}\_\ell_7[q]) \ .$$

Since our verification approach proves properties of the original MPS by verification of its bisimilar WS1S translation we have to translate the state formulae to those corresponding to the WS1S translation. We can use the translation function *tr* and get as safety property for the WS1S translation of Szymanski's algorithm (Example 3.3):

$$\Box\neg(\exists_P\, p, q, p \neq q : p \in \mathrm{At}\_\ell_7 \wedge q \in \mathrm{At}\_\ell_7) \ .$$

As a minimal liveness requirement we expect a process to eventually reach its critical section whenever there exists a process that tries to enter its critical section:

$$\Box(\exists_P\, p : p \in \mathrm{At}\_\ell_1 \rightarrow \Diamond\, \exists_P\, p : p \in \mathrm{At}\_\ell_7) \ .$$

Note that the process being at ℓ_1 and that expected to be at ℓ_7 are not necessarily the same. We call such liveness property *communal accessibility* [MP92, MP91].

For the Simple ME algorithm of Examples 2.9 and 3.9, the safety property can be stated as:

$$\Box\neg(\exists_P\, p, q, p \neq q : p \in \mathrm{At}\_\ell_2 \wedge q \in \mathrm{At}\_\ell_2) \ .$$

We sometimes use equivalently

$$\Box(\exists_P\, p : \mathrm{At}\_\ell_2 \subseteq \{p\}) \ .$$

Communal accessibility is expressed as:

$$\Box(\exists_P\, p : p \in \mathrm{At}\_\ell_1 \rightarrow \Diamond\, \exists_P\, p : p \in \mathrm{At}\_\ell_2) \ . \qquad \Box$$

Especially for the Simple ME algorithm we introduced some fairness conditions in order to prove that each process eventually reaches its critical section whenever it wants to. This property is stronger than communal accessibility, it expects the same process to make progress. This type of liveness property is called *individual accessibility*. To define such a property formally we have to extend the definition of LTL formulae.

Definition 4.3 (Universal properties)
Let ψ be an LTL formulae with $\mathit{free}(\psi) = \{p_1, \ldots, p_k\}$. Then the formula φ defined as

$$\forall_n\, p_1, \ldots, p_k : \psi(p_1, \ldots, p_k) \ ,$$

respectively,

$$\forall_P\, p_1, \ldots, p_k : \psi(p_1, \ldots, p_k) \ ,$$

is called a *universal property*.

A model for a universal property φ is an infinite sequence $\sigma : s_0, s_1, \ldots$ such that the s_i give rigid interpretations to the variables n, respectively, P. Then, $\sigma \models \varphi$ iff for all $x_1, \ldots, x_k \in \{0, \ldots n-1\}$, respectively, $x_1, \ldots, x_k \in P$

$$\sigma[x_1, \ldots, x_k/p_1, \ldots, p_k] \models \psi(p_1, \ldots, p_k)$$

holds, i.e., the p_i are also interpreted as rigid variables. □

Now we are able to express individual accessibility of the mutual exclusion algorithms formally.

Example 4.4 (Individual accessibility)
For Szymanski's algorithm individual accessibility is characterized by

$$\forall_P\, p : \Box(p \in \mathrm{At}\_\ell_1 \rightarrow \Diamond p \in \mathrm{At}\_\ell_7) \ .$$

In case of the Simple ME algorithm where ℓ_2 represents the critical section we have:

$$\forall_P\, p : \Box(p \in \mathrm{At}\_\ell_1 \rightarrow \Diamond p \in \mathrm{At}\_\ell_2) \ .$$ □

4.2 Verification by Abstraction

The previous section shows how to specify properties of transition systems. Proving an algorithm given as a transition system correct means to establish all those properties one has identified to be important to hold. For sequential algorithms we have seen that Floyd's methods can establish basic safety and liveness properties. For concurrent programs the methods are generalized by

the methods of Owicki & Gries [OG76a, OG76b, Owi75, Owi76, NDOG86, Gri77] or Apt, Francez & de Roever [AFdR80, dR85]. In the parameterized setting these methods are applicable as well, if it is possible to find assertion networks for each process i and for any n.

In our abstraction-based verification approach we prove properties of the original system by establishing them for an abstract system. The result of an abstraction is not a correctness proof but another transition system behaving similar to the original one in certain ways. Via an abstraction relation the computations of the concrete system can be simulated by the abstract one. The abstract system over-approximates the computations of the original system. Hence, the benefit of verifying the abstract system instead of the original, concrete one, is that it is in general simpler. Choosing an adequate abstraction allows to focus on certain properties one is interested in. The simpler abstract system then allows for simpler proofs or even automatic verification.

The concept of abstraction does not only appear in the formal context we define in the next section, but abstraction is often also applied informally when modeling an algorithm one wants to verify. Instead of handling a concrete implementation one verifies a model that is reduced to the core functionality one is interested in. It would be a research task of its own to prove formally the relationship between a real world implemented algorithm and its model verified in scientific papers.

4.2.1 Abstraction-based Verification

To prove by abstraction is first of all an intuitive task. Dealing with everyday problems one abstracts from unimportant details to focus on the main problem. In a mathematical setting, the basis to express formally the relationship between the concrete and the abstract level is [CC77]. There it is done in terms of domain theory and Galois connections.

We give the basic definitions for abstraction of transition systems and show how to prove properties of systems in that framework. A general survey on the different kinds of abstraction can be found in [dRE98].

Definition 4.5 (Abstraction)
Given a deadlock-free[1] transition system $\mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T})$ and a total abstraction relation $\alpha \subseteq \Sigma \times \Sigma_A$, we say that $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ is an *abstraction* of $\mathcal{S}$ w.r.t. α, denoted by $\mathcal{S} \sqsubseteq_\alpha \mathcal{S}_A$, if the following conditions hold:

1. $\Theta \subseteq \alpha^{-1}(\Theta_A)$ and

[1] Throughout this work we only consider deadlock free transition systems which can be achieved by adding an idle transition.

2. $\tau \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \tau_A$

for adequate pairs (τ, τ_A), $\tau \in \mathcal{T}$, $\tau_A \in \mathcal{T}_A$. Moreover, we write $[\![\mathcal{S}]\!]$ and $[\![\mathcal{S}_A]\!]$ to denote the computations of the systems defined in the usual way. For finite Σ_A, we call α a finite abstraction relation. □

As mentioned above we would like to use the abstract system to establish properties of the concrete one. The following preservation result allows us for LTL to do so.

Theorem 4.6 (Preservation)
Let φ, φ_A be LTL formulae and let $[\![\varphi]\!]$ (resp. $[\![\varphi_A]\!]$) denote the set of models of φ (resp. of φ_A). Then, $\mathcal{S} \sqsubseteq_\alpha \mathcal{S}_A$, $\alpha^{-1}([\![\varphi_A]\!]) \subseteq [\![\varphi]\!]$, and $\mathcal{S}_A \models \varphi_A$ implies $\mathcal{S} \models \varphi$. □

This theorem shows the interest of verification by abstraction. In this thesis we present techniques to find finite abstraction relations. Moreover, we construct the abstract system $\mathcal{S}_A$ fully automatically. Since $\mathcal{S}_A$ is finite, it can automatically be checked whether $\mathcal{S}_A \models \varphi_A$ holds.

The theorem above does not depend in an essential way on LTL formulae. In fact, a similar preservation result holds for any temporal logic without existential quantification over paths, e.g., $\forall CTL^\star$, LTL, or $\mu_\Box$ [CGL94, DGG94, LGS+95].

In case $\mathcal{S}$ is a fair transition system with $\mathcal{F}$ as fairness formula, i.e., $\mathcal{S}^\mathcal{F} = (\mathcal{V}, \Theta, \mathcal{T}, \mathcal{F})$, and if $\mathcal{F}_A$ is the fairness formula of $\mathcal{S}_A^\mathcal{F} = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A, \mathcal{F}_A)$, then by requiring $\alpha^{-1}([\![\neg\mathcal{F}_A]\!]) \subseteq \overline{[\![\mathcal{S}^\mathcal{F}]\!]}$, we have the same preservation result as above. We indicate this type of abstraction by $\mathcal{S}^\mathcal{F} \sqsubseteq_\alpha^\mathcal{F} \mathcal{S}_A^\mathcal{F}$.

The premise for fair abstraction is trivially fulfilled if one has $\alpha^{-1}([\![\neg\mathcal{F}_A]\!]) \subseteq [\![\neg\mathcal{F}]\!]$, i.e., the fairness requirements of the abstract system do not exclude as much computations as the concrete fairness requirements. Furthermore, the definition allows to add fairness conditions to the abstract system that are guaranteed to hold in the concrete system without any fairness assumptions, i.e., $\alpha^{-1}([\![\neg\mathcal{F}_A]\!]) \subseteq \overline{[\![\mathcal{S}]\!]}$. We use this fact in Chapter 7 to safely add fairness requirements.

4.2.2 Finding the Abstraction Relation

The main problem for the verification by abstraction approach is to find the abstraction relation, i.e., how to define an abstraction relation that focuses on the properties one likes to establish. Numerous papers applying the abstraction-based approach to verification center around this problem [BCG89, Lon93, LGS+95, Dam96, Kel95, DGG97, AAB+99, Uri99]. Of

course the abstraction depends on what details one wants to hide; data abstraction reduces the data handled, control abstraction merges different control locations, and in the parameterized setting one tries to find a representation for an arbitrary number of processes. We discuss the topic of finding an abstraction relation in Chapter 5 and give there some heuristics for parameterized systems.

Except for decidable classes of systems where adequate abstraction relations can be found, all approaches have to deal with the failure case, i.e., the abstract system is to coarse to establish a desired property. For finite abstractions, model-checking of the abstract system yields counterexamples in the failure case. [CGJ+00, CCK+02, CGKS02] give solutions of how to refine the abstraction relation using the found counterexamples.

4.2.3 Generating the Abstract System

Given an abstraction relation it is not clear how to get to the abstract system. In [GS97, BLO98a] the theorem prover PVS is used to check whether a transition belongs to the abstract system or not. One of the main contributions of this work is that, by using the decidable logic WS1S to model the systems, we construct the abstract system fully automatically. That is the reason for calling our method semi-automatic: once the abstraction relation is chosen, constructing the finite abstract system and model-checking it is fully automatic. Hence, a positive outcome of the model-checker establishes the property for the original system. In the failure case refinements of the abstraction have to be made.

4.3 Model-Checking

We use the abstraction-based verification approach to get an abstract system with finite state space for that the verification problem is decidable. The technique that allows to check whether an arbitrary finite system satisfies a property or not is called *model-checking*.

The term model-checking has been used in the literature to refer to an algorithmic approach for showing that a system satisfies a formal specification given as a temporal logic formula or as an automaton. The technique was invented independently and at about the same time by [QS81] and [CE81]. Since then, many different model-checking algorithms have been developed. The papers [LP89, SC85, VW86] are of special interest in this context as they provide algorithms to check for LTL properties.

LTL model-checking whether a finite state system $\mathcal{S}$ satisfies a property φ or not works roughly as follows.

- Construct an automaton $\mathcal{A}_{\neg\varphi}$ that corresponds to the specification $\neg\varphi$.
- Model the system as an automaton $\mathcal{P}$. If the program is given as a state graph, roughly speaking, this state graph can be considered as the automaton.
- Check that there is no string that is accepted both by $\mathcal{P}$ and $\mathcal{A}_{\neg\varphi}$. To do so, built another automaton $\mathcal{B}$ that is the synchronous product of $\mathcal{P}$ and $\mathcal{A}_{\neg\varphi}$. If $\mathcal{L}(\mathcal{B}) = \emptyset$, then the system satisfies the property, i.e., $\mathcal{S} \models \varphi$.

In the sequel we use the fact that we can check $\mathcal{S} \models \varphi$ for a finite state system $\mathcal{S}$ and an LTL formula φ whenever needed. In Chapter 6 we discuss which model-checkers we use in our framework. An excellent overview on model-checking can be found in [CGP99].

Chapter 5

Abstraction of WS1S Systems

Next, we want to analyze the WS1S systems defined in Chapter 3 by abstraction as explained in Chapter 4. Let $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ be a given WS1S system. We show how to define the abstract system $\mathcal{S}_A$ for a given abstraction function α. Then we show how to construct such an α by using *predicate abstraction*, that means, we identify state formulae characterizing interesting system configurations and use them to define the abstraction relation. Each predicate introduces a boolean variable. Hence, α is a boolean abstraction function and the abstract system we construct is finite. Moreover, if all chosen predicates are WS1S formulae the abstraction relation can be expressed as a WS1S formula $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$ itself. Together with the fact that all the transitions in $\mathcal{T}$ are expressed in WS1S this allows us to give an effective construction of the abstract system. The constructed finite system can then be subject to model-checking techniques.

5.1 The Abstract System

Assume that we already have an abstraction function on the state space. Hence, the state space of the abstract system is determined by the abstraction function $\alpha : \Sigma_{\mathcal{W}} \to \Sigma_A$. The variable set contains boolean variables for every predicate that is used to define our abstraction. It remains to find an initial condition and transitions for the abstract system. The kind of abstraction that is defined in Definition 4.5 represents a conservative approximation, i.e., every computation at the concrete level can be simulated at the abstract level. In other words, if there exists a transition connecting two concrete states, the abstractions of those states need to be connected by an abstract transition. Hence, the initial condition as well as the transitions can be expressed by existential quantification over the concrete states. The initial states of the

abstract system we construct can be described by the formula Θ_A:

$$\exists \mathcal{V} : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \ .$$

As transitions the abstract system contains for each concrete transition τ an abstract transition τ_A, which is characterized by the formula

$$\exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A)$$

with free variables $\mathcal{V}_A$ and $\mathcal{V}'_A$.

Sometimes, we are interested in so-called *universal* progress or response properties, which are properties that guarantee that each single process p eventually makes some progress, or each request by p to q eventually gets a response by q. To prove those properties by abstraction the abstraction function has to focus on single processes, i.e., the abstraction function contains p or p, q as free variables ($\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p)$ or $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q)$).

Then, the abstract system contains as abstract transitions

$$\exists \mathcal{V}, \mathcal{V}' : \exists p, q : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A, p, q)$$

and starts in initial state:

$$\exists \mathcal{V} : \exists p, q : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q) \ .$$

The next definition subsumes both kinds of abstraction.

Definition 5.1 (Abstract system $\mathcal{S}_A$)
Let $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ be a WS1S system and let α be a boolean abstraction function expressed as a WS1S formula $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$.

Then call $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ with

$$\begin{aligned} \Theta_A &\equiv \exists \mathcal{V} : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \\ \mathcal{T}_A &\equiv \{\exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \mid \tau \in \mathcal{T}\} \end{aligned}$$

global abstraction or simply abstraction of $\mathcal{S}$ with respect to α.

If, moreover, α has free first order variables p, q (we will use the definition analogously for numbers of processes other than 2), i.e., it is represented as WS1S formula $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q)$, then we call the abstract system $\mathcal{S}_A^{p,q} = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ with

$$\begin{aligned} \Theta_A &\equiv \exists \mathcal{V} : \exists p, q : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q) \\ \mathcal{T}_A &\equiv \{\exists \mathcal{V}, \mathcal{V}' : \exists p, q : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A, p, q) \mid \tau \in \mathcal{T}\} \end{aligned}$$

local abstraction of $\mathcal{W}$ w.r.t. α. Sometimes we also denote the local abstraction simply with $\mathcal{S}_A$. □

It remains to prove that $\mathcal{S}_A$ is indeed an abstraction of $\mathcal{W}$ w.r.t. α.

Theorem 5.2 ($\mathcal{W} \sqsubseteq_\alpha \mathcal{S}_A$)
For a given WS1S system $\mathcal{W}$ and an abstraction relation α, let $\mathcal{S}_A$ be the abstract system constructed in Definition 5.1. Then, we have $\mathcal{W} \sqsubseteq_\alpha \mathcal{S}_A$.

PROOF To prove abstraction according to Definition 4.5 we have to establish

1. $\Theta \subseteq \alpha^{-1}(\Theta_A)$ and
2. $\tau \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \tau_A$

for corresponding $\tau \in \mathcal{T}$, $\tau_A \in \mathcal{T}_A$.

ad (1): Consider an initial state $s \in [\Theta(\mathcal{V})]$. Hence, we have an evaluation of $\mathcal{V}$ such that $s(\mathcal{V}) \models \Theta$. Θ_A is defined as $\exists \mathcal{V} : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$. Assuming α to be total, we have an abstract state s_A such that $s_A(\mathcal{V}_A) \models \exists \mathcal{V} : \Theta(\mathcal{V}) \wedge \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$. Applying the inverse of α to s_A includes s. Hence, we have proven (1).

ad (2): Let $\tau \in \mathcal{T}$ be a concrete transition. Consider any abstract state $s_A \in \Sigma_A$. If there exists any concrete counterpart s we have to prove that its τ-successor s' is a concrete counterpart of the τ_A-successor of s_A. Assume we have s and s'. Assuming α to be total, we get an s'_A such that $(s'(\mathcal{V}), s'_A(\mathcal{V}_A)) \models \alpha(\mathcal{V}, \mathcal{V}_A)$. By definition, we have

$$(s(\mathcal{V}_A), s'_A(\mathcal{V}_A)) \models \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \ .$$

Therefore, $s' \in \alpha^{-1} \circ \tau_A(s_A)$. ∎

5.2 Choosing the Abstract Variables

Since the verification of parameterized systems is undecidable in general (Theorem 2.12) there exists no strategy which abstract variables to choose in order to establish some properties of the concrete system by analyzing the abstract system. Nevertheless, we give some guidelines which abstract variables to choose. These heuristics are based on our experiences in verifying different algorithms. Moreover, they are justified in Chapter 9 when showing that for restricted classes of parameterized networks the abstractions found according to these guidelines are complete, i.e., they are sufficient to analyze the original systems.

In the section above we introduced local and global abstractions. For local abstractions, the basic set of abstract variables is chosen such that

the abstract system keeps an exact copy of certain processes (p and q, for example). We define the abstraction relation as predicate abstraction, i.e., we choose a predicate φ characterizing important states of the concrete system and introduce an abstract, boolean variable b_φ to have that information also present in the abstract system. We denote such definitions with $b_\varphi \equiv \varphi$. Hence, to focus on two processes p and q we introduce the variables

$$\begin{aligned} p_X &\equiv p \in X \\ q_X &\equiv q \in X \end{aligned}$$

for every $X \in \mathcal{V}$ and perhaps

$$order \equiv p < q$$

whenever the order of process identifiers is important. Sometimes, these variables have to be amended by those specifying the context of the concrete processes. Hence, we have to characterize certain configurations of processes important for the algorithm. Considering global abstractions we are dealing exclusively with those important configurations. Hence, our heuristics for choosing abstract variables for global abstractions presented next carry over directly to local abstractions.

5.2.1 Guards, Locations, and Contexts

Our heuristic to construct an abstraction function for WS1S systems assumes that the transitions have the form

$$\exists_P i : G \wedge L(i) \wedge C(i) \wedge \mathcal{V}' = exp(\mathcal{V}, \mathcal{V}') \ ,$$

for asynchronous systems, resp.

$$\forall_P i : \bigwedge_{\tau \in \mathcal{T}} G_\tau \wedge L_\tau(i) \wedge C_\tau(i) \wedge \mathcal{V}' = exp_\tau(\mathcal{V}, \mathcal{V}') \ ,$$

for the one transition of a synchronous system, where $G, L(i), C(i)$, resp. $G_\tau, L_\tau(i), C_\tau(i)$ are WS1S formulae whose free variables are in $\mathcal{V}$ and such that:

- G is a first-order closed WS1S formula. Intuitively, G describes a global condition. E.g., in the Szymanski example (see Example 3.3), the presented transition contains the global condition $\forall_P j : j \in \text{At\_}\ell_1 \cup \text{At\_}\ell_2 \cup \text{At\_}\ell_4$.

- $L(i)$ is a quantifier-free formula with i as the unique free first-order variable. Intuitively, if i models a process index then $L(i)$ is a condition on the local state of this process.

- $C(i)$ is a condition that as in the case of $L(i)$ has i as the unique free first-order variable but which contains first-order quantifiers. Intuitively, it imposes conditions on the context of process i.

Though, the above requirements restrict the set of considered WS1S systems, it includes most translations of systems considered in this work.

Let $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ be a WS1S system whose transitions satisfy the restriction above.

5.2.2 Heuristics

We are now prepared to present our heuristic for constructing abstraction functions. The set $\mathcal{V}_A$ of abstract variables contains a boolean variable b_X for each variable $X \in \mathcal{V}$. Moreover, for each global guard G, resp. local guard L occurring in a transition, it contains a boolean variable b_G, resp. b_L. Since the context guards $C(i)$ describe a dependence between process i and the remaining processes, it turns out to be useful to combine them with the local guards. Indeed, this allows to check, for instance, whether some dependence is propagated over some transitions. Therefore, we introduce boolean variables b_{L_k,C_l} for some boolean combinations of local guards and context guards. Additionally, $\mathcal{V}_A$ contains a boolean variable for each state formula appearing in the property to be verified.

It remains now to present how we relate the concrete and abstract states, i.e., to describe an abstraction relation α. The abstraction relation α can be expressed on the syntactic level by a predicate $\widehat{\alpha}$ over $\mathcal{V}, \mathcal{V}_A$ which is defined as the conjunction of the following equivalences:

$$\begin{aligned} b_X &\equiv X \neq \emptyset \\ b_G &\equiv G \\ b_L &\equiv \exists_P i : L(i) \\ b_{L_k,C_l} &\equiv \exists_P i : L_k(i) \wedge C_l(i) \\ b_\xi &\equiv \xi \end{aligned}$$

Henceforth, we use $\widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A)$ to denote the predicate obtained from $\widehat{\alpha}$ by substituting the unprimed variables with their primed versions.

5.3 Global vs. Local Abstraction

Which kind of abstraction, global or local, to choose depends on the properties one wishes to verify. A property like mutual exclusion, never two distinct processes are in the critical section at the same time, can be proven using global abstraction. Properties defining the behavior of an individual process, in the section above defined as universal properties, need local abstractions. E.g., for mutual exclusion algorithms properties to be named are *individual accessibility* stating, whenever a process wants to enter the critical section it finally does. In contrast, the property of *communal accessibility* only states that whenever a process wants to enter the critical section *one process* finally does.

The section above gave some guidelines how to choose abstract variables in the global and in the local context. There and most notably in practice it gets obvious that the borderline between global and local abstraction is fuzzy.

Nevertheless, to illustrate both kinds of abstraction we show how to define a global and a local abstraction for Szymanski's mutual exclusion algorithm (Examples 2.10 and 3.3). Both are strong enough to establish the safety property of mutual exclusion which we prove in the next chapter. But, they differ in their ability to prove liveness properties.

Example 5.3 (Abstraction of Szymanski's algorithm)
The modeling as a WS1S system introduces a set variable P and set variables $\mathrm{At\_}\ell_1, \ldots, \mathrm{At\_}\ell_7$.

Applying the heuristics of Section 5.2 for global abstraction we get seven boolean variables:

$$\psi_i \equiv \mathrm{At\_}\ell_i \neq \emptyset, \text{ for each } 1 \leq i \leq 7 \ .$$

The global guards not referring to i can be derived by the above variables and do not lead to a finer partitioning of the state space. All the local guards $i \in \mathrm{At\_}\ell_l$ would introduce a boolean variable with meaning $\exists i : i \in \mathrm{At\_}\ell_l$ which is equivalent to stating that $\mathrm{At\_}\ell_l$ is not empty. In the transition leading from ℓ_6 to ℓ_7 we have a context $\forall j < i : j \in \mathrm{At\_}\ell_1 \cup \mathrm{At\_}\ell_2 \cup \mathrm{At\_}\ell_4$ which we have to combine with the local guards $i \in \mathrm{At\_}\ell_l$. For this example it turns out to be enough to take only one combination, namely

$$\varphi \equiv \exists i : i \in \mathrm{At\_}\ell_7 \wedge \forall j < i : j \in \mathrm{At\_}\ell_1 \cup \mathrm{At\_}\ell_2 \cup \mathrm{At\_}\ell_4 \ .$$

Moreover, for the property of interest we introduce

$$\xi \equiv \neg\exists l, j : l \neq j \wedge l \in \mathrm{At\_}\ell_7 \wedge j \in \mathrm{At\_}\ell_7 \ .$$

In contrast, when choosing a local abstraction we copy two processes into our abstract system, i.e., we introduce

$$\begin{aligned} p_i &\equiv p \in \text{At\_}\ell_i \\ q_i &\equiv q \in \text{At\_}\ell_i, \text{ for each } 1 \leq i \leq 7 \end{aligned}$$

as abstract variables to define the abstraction relation $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, p, q)$. Since, the order of process identifiers is important in this algorithm we add another variable $order \equiv p < q$. The property of mutual exclusion is expressed as $\Box\neg(p_7 \wedge q_7)$ which can be checked on the abstract system by a model-checker as shown in the next chapter.

The difference of both approaches came in when verifying liveness properties of Szymanski's algorithm. Whereas communal accessibility can be expressed in the global abstraction as $\Box(\psi_2 \rightarrow \Diamond\psi_7)$, individual accessibility is not expressible in this context. Individual accessibility on the other hand can be expressed as $\Box(p_2 \rightarrow \Diamond p_7)$ is the local setting which is not possible in global one. □

The verification results of this example are presented in the next chapter and in Chapter 8.

5.4 Constructing the Abstract System

To compute the abstract system, one has to find all states satisfying the formulae defining the initial state and the transitions, which is possible since they are WS1S formulae. This means, choosing properties $\varphi_i(\mathcal{V})$ of the concrete system as abstract variables $a_i \equiv \varphi_i(\mathcal{V})$ allows us to compute automatically the abstract system according to the boolean abstraction function $\bigwedge_{i=1}^{n} a_i \leftrightarrow \varphi_i(\mathcal{V})$.

We use MONA [KM98, HJJ+96] to decide the predicates mentioned above. In fact, MONA is able to construct all models of a WS1S predicate.

Our system PAX constructs the abstract system as follows. It uses MONA to compute the abstract initial states and the abstract transitions. To do so, it creates from its input files input for MONA. I.e., files containing the predicates describing the concrete initial state, the concrete transitions, and the definition of the abstraction function are used to generate WS1S predicates with the abstract variables being free in these formulae. MONA constructs an automaton which represents all satisfying interpretations of the predicates. Since the free variables are boolean the interpretations are vectors over the abstract variables and have length 1. Hence, PAX can interpret the MONA results directly as abstract initial state, resp. abstract transitions. Once an

abstract system is constructed, a translation to SMV, alternatively to SPIN, can be used to model-check the obtained abstract system.

Chapter 6

Model-Checking the Abstract System

The last chapter presents some heuristics on how to abstract a WS1S transition system. The heuristics assume that the WS1S system is a translation of an original MPS. In the previous chapter it is also shown how to construct the abstract system automatically when the abstraction relation is given. Since we define a finite set of boolean abstract variables to characterize the concrete system, the resulting abstract system is finite state by construction.

As described in Section 4.3, for finite state systems a variety of model-checking techniques has been derived to analyze those systems exhaustively. Our tool PAX allows to represent the constructed abstract system suitably as input for different model-checkers. We take those different model-checkers as a given back-end for our verification approach. Indeed, it is possible to adapt the output for the abstract system in order to use one's preferred model-checker.

We use two different kinds of model-checkers. Spin [Hol91, Hol99] is an enumerative model-checker with a user-friendly input language Promela. Spin allows to model the abstract system as well as instances of the concrete MPS easily. This capability can be applied to simulate found counterexamples at the concrete level.

SMV [McM92, McM93] is a symbolic model-checker using symbolic representations of the state space via BDDs [Bry85, Bry86, BM01]. Originally, SMV is a CTL model-checker, i.e., it can check finite state models versus Computation Tree Logic formulae. There are now extensions available which also handle LTL formulae. There is one from CMU called NuSMV [CCGR99] and one which is a redesigned SMV including LTL by Kenneth McMillan called CadenceSMV [McM99a, McM99b]. The NuSMV model-checker is well-suited for our purposes; it performs well for our examples and is ca-

pable of handling several fairness conditions. To verify liveness properties we generate several fairness constraints automatically and pass them as premises of the desired LTL formula to the model-checker. We will see their necessity in Section 6.2 and show how to generate these needed fairness requirements in the next part of the thesis.

6.1 Safety Properties

The safety properties considered in this work are of the form that a certain condition is always valid (we restrict ourselves to future LTL). To check for such invariance properties one has to compute the set of reachable states. Of course, all the model-checkers mentioned above are capable of doing so, but for a first analysis it is preferable to save the translation to the model-checker input language and use a tool working directly on the representation of the abstract system. For illustration we present parts of the abstract system for Szymanski's algorithm constructed with the global abstraction as presented in Example 5.3. There, we introduced 10 abstract variables, $\psi_0, \ldots, \psi_7$, to characterize positions of the processes, φ as an important context controlling access to the critical section, and ξ expressing the safety property. The output of MONA formatted by PAX is shown in Figure 6.1.

The abstract transitions are represented as pairs of pre- and post-states. An X in the pre-state indicates that the transition does not depend on that value, an X in the post-state abbreviates transitions to a state where the corresponding value is *true* and one where the value is *false*.

As indicated by the dots the listing of abstract transitions that correspond to a given concrete transition is not complete. The second transition chosen for 't67', the transition leading from location 6 to location 7 in the concrete system, makes ξ *false*. That means that something bad has happened. This does not immediately contradict the desired property, we first have to check whether the pre-state allowing to take the step to the bad state is reachable. Therefore, PAX applies a fixed point computation [Kna28, Tar55, Kle52] starting from the initial state and searching for matching pre-states in the transitions. For every match the corresponding post-state is added to the list of reachable states. Then, the search for matching pre-states is extended to the list of reachable states computed so far. Termination of the fixed point computation is obvious in this case, since we have a finite state system.

Transition	ψ_0	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	ψ_6	ψ_7	φ	ξ	ψ'_0	ψ'_1	ψ'_2	ψ'_3	ψ'_4	ψ'_5	ψ'_6	ψ'_7	φ'	ξ'
init	1	0	0	0	0	0	0	0	0	1										
t01	1	0	0	0	0	0	0	0	0	1	X	1	0	0	0	0	0	0	0	1
t12	X	1	X	0	0	0	0	0	0	1	X	X	1	0	0	0	0	0	0	1
	X	1	X	0	1	0	0	0	0	1	X	X	1	0	1	0	0	0	0	1
t23	X	0	1	X	0	0	0	0	0	1	X	0	X	1	0	0	0	0	0	1
				.	.	.								.	.	.				
t34	X	0	1	1	X	0	0	0	0	1	X	0	X	1	1	0	0	0	0	1
				.	.	.								.	.	.				
t35	X	0	0	1	0	0	0	0	0	1	X	0	0	X	0	1	0	0	0	1
				.	.	.								.	.	.				
t45	X	0	0	0	1	0	0	1	0	1	X	0	0	0	X	1	0	1	0	1
				.	.	.								.	.	.				
t56	X	0	0	0	0	1	X	0	0	1	X	0	0	0	0	X	1	0	0	1
				.	.	.								.	.	.				
t67	X	0	0	0	0	0	1	0	0	1	X	0	0	0	0	0	X	1	1	1
t67	X	0	0	0	0	0	1	1	X	1	X	0	0	0	0	0	X	1	1	0
				.	.	.								.	.	.				
t70	X	X	0	0	0	1	0	1	1	X	0	X	0	0	0	1	0	0	0	1
				.	.	.								.	.	.				

Figure 6.1: Parts of the abstract system for Szymanski

6.1.1 Translation to SMV

Considering other safety properties than only invariance properties or to benefit from the facilities of a high-performance model-checker, the PAX tool set allows to translate an abstract system characterized as shown in Figure 6.1 to the SMV input language.

An SMV input file has the general form:

```
MODULE main
 VAR
   ...

 INIT
   ...

 TRANS
   ...
```

The translation is straightforward. For Szymanski's algorithm the translation of the abstract system into this format looks as follows with the obvious replacements for Greek letters.

Variable declaration:

```
VAR
        psi0 : BOOLEAN;
        psi1 : BOOLEAN;
        psi2 : BOOLEAN;
        psi3 : BOOLEAN;
        psi4 : BOOLEAN;
        psi5 : BOOLEAN;
        psi6 : BOOLEAN;
        psi7 : BOOLEAN;
        phi  : BOOLEAN;
        xi   : BOOLEAN;
```

Initial condition:

```
INIT
    TRUE
    & psi0 = 1
    & psi1 = 0
    & psi2 = 0
    & psi3 = 0
    & psi4 = 0
    & psi5 = 0
```

```
& psi6 = 0
& psi7 = 0
& phi  = 0
& xi   = 1
```

Transition relation: SMV is originally tailored for hardware verification, hence, the semantics of defining several transitions is that they are executed simultaneously. To use SMV for our purposes we define one large transition including all abstract transitions by combining them disjunctively:

```
TRANS
  FALSE
  -- T01
  |  psi0 &  !psi1 &  !psi2 & !psi3 & !psi4
     & !psi5 & !psi6 & !psi7 & !phi  & xi
     & NEXT(psi1) = 1 & NEXT(psi2) = 0 & NEXT(psi3) = 0
     & NEXT(psi4) = 0 & NEXT(psi5) = 0 & NEXT(psi6) = 0
     & NEXT(psi7) = 0 & NEXT(phi)  = 0 & NEXT(xi)   = 1
```

NuSMV checks the safety property $\Box\xi$ within a second.

6.2 Liveness Properties

In order to compute the set of reachable states, PAX checks for each reachable state which transition is enabled in that state. This information can be used to construct an abstract state graph which illustrates the dynamic behavior of the algorithm. The theory behind these abstract state graph can be found in [BBLS00]. The abstract graph for Szymanski's algorithm is shown in Figure 6.2. For notational brevity, we leave out the label $0 \to 1$ for most of the loops.

A desired property of Szymanski's mutual exclusion algorithm beyond safety is that the protocol is always successful in electing a process to enter the critical section, i.e., whenever a process wants to enter the critical section eventually one process succeeds. This property is often called *communal accessibility* [MP92, MP91]. In our model of the algorithm a process requests entry to the critical section by proceeding from ℓ_0 to ℓ_1. Assume that all processes are in their initial state, then the only possible transition is that of entering the protocol and asking for access. Hence, from each node in the state graph there should be paths to a shadowed node (those nodes with a

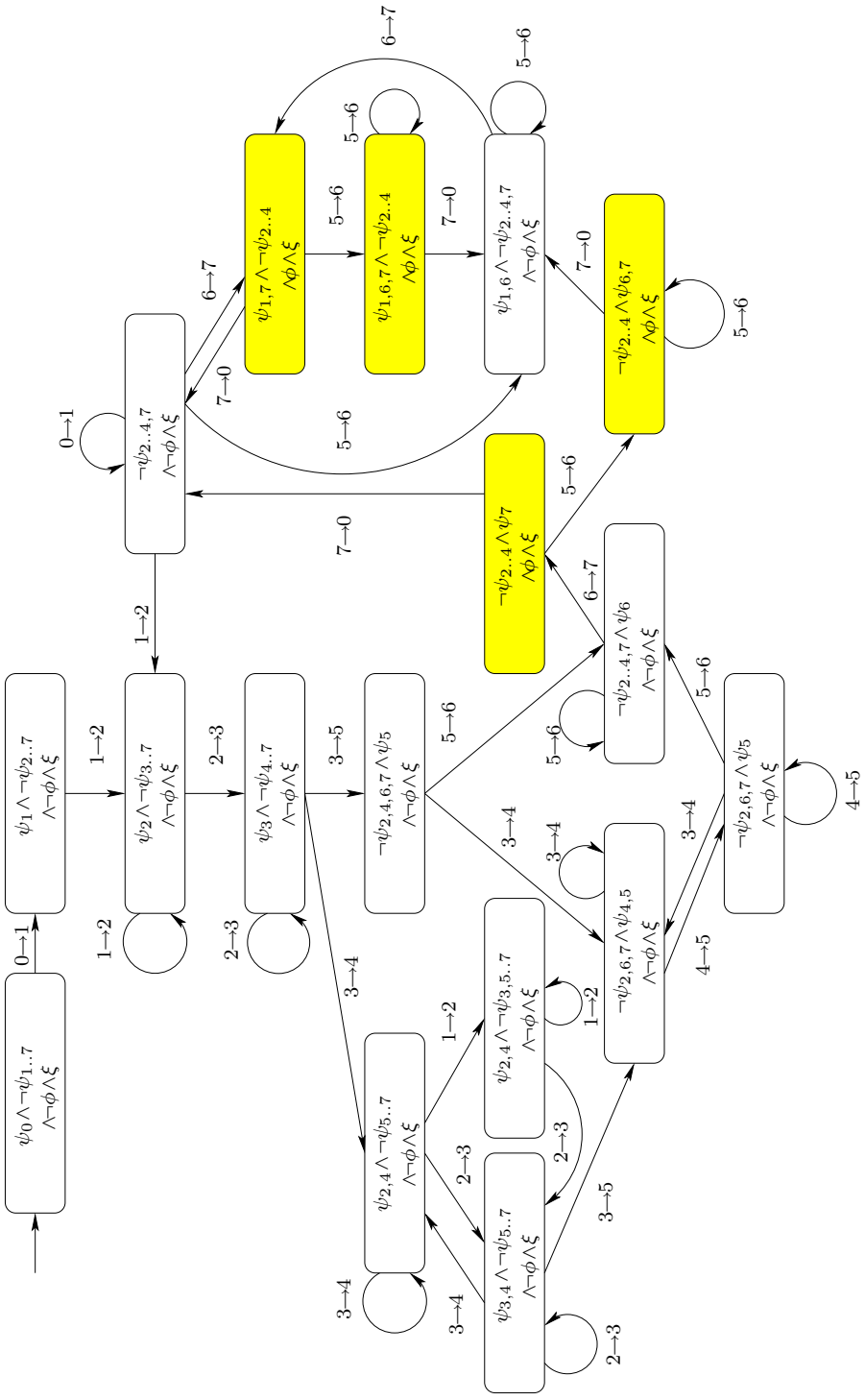


Figure 6.2: Reachability graph for Szymanski's algorithm

process in the critical section are shadowed) and each of those paths is of finite length.

Examining the graph, this is obviously not the case. This is due to the cycles which generate infinite traces in $\tilde{\mathcal{G}}$ without ever reaching a shadowed node. However, these traces have no corresponding computations in the concrete WS1S system. E.g., the loops labeled $1 \to 2$ are the abstraction of the transition taking an element out of $\text{At\_}\ell_1$ and adding it to $\text{At\_}\ell_2$. Since WS1S is interpreted over finite sets, it is clearly impossible to execute transition $1 \to 2$ infinitely often without taking a transition that adds elements to $\text{At\_}\ell_1$.

In the next part of the thesis we focus on this problem and show how to generate adequate fairness conditions to exclude such counterexamples.

To give an idea of how a solution could look like, we observe that only finitely many processes can leave a certain location. In other words, when infinitely many processes leave a location then an infinite amount of them has to enter the location. Fortunately, the transition relation is given by 9 separate transitions, each of them describes explicitly which location is left and which is joined. If we use this fact and augment the abstract system with flags indicating which transition was taken, we can formalize the needed fairness constraint *fair*.

$$
\begin{array}{lll}
(\Box\Diamond t_{01} \to \Box\Diamond t_{70}) & \wedge & \\
(\Box\Diamond t_{12} \to \Box\Diamond t_{01}) & \wedge & (\Box\Diamond t_{23} \to \Box\Diamond t_{12}) \wedge \\
(\Box\Diamond t_{34} \to \Box\Diamond t_{23}) & \wedge & (\Box\Diamond t_{35} \to \Box\Diamond t_{23}) \wedge \\
(\Box\Diamond t_{45} \to \Box\Diamond t_{34}) & \wedge & (\Box\Diamond t_{56} \to \Box\Diamond (t_{45} \vee t_{35})) \wedge \\
(\Box\Diamond t_{67} \to \Box\Diamond t_{56}) & \wedge & (\Box\Diamond t_{70} \to \Box\Diamond t_{67})
\end{array}
$$

The fairness condition can be added safely to the abstract system, since it trivially holds in the concrete system and therefore it cannot exclude computations with real counterparts.

Obviously, the fairness condition excludes all trivial cycles. For a more complex cycle see Figure 6.3. It shows a strongly connected part of the graph with the only in-going edge *in* and only outgoing edge *out*. The labels are changed to emphasize which sets increase and which decrease. To prove communal accessibility it is necessary to show that the system cannot cycle forever in this component.

We could convince ourselves that communal accessibility indeed holds for Szymanski's algorithm. To increase our confidence in that result, we can check $\mathit{fair} \to \Box\Diamond \text{At\_}\ell_7$ with NuSMV and get a positive answer.

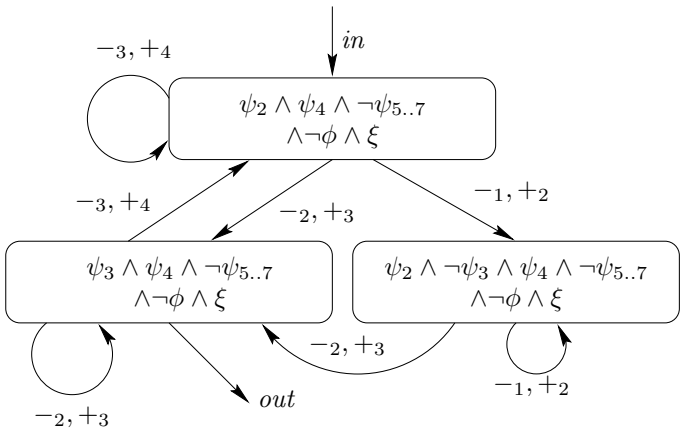


Figure 6.3: Part of the labeled state graph

6.3 False Negatives

The abstract cycles in the previous section that do not have any concrete counterparts are *false negatives*. Due to our conservative abstractions defined in Chapter 5 we may get counterexamples violating the checked property that can not occur in the concrete system. The false negatives found above show a basic problem of liveness verification by finite abstractions. The abstraction relation introduces cycles in the abstract system that are not present at the concrete level. This makes verification of liveness properties complicated and calls for the introduction of fairness arguments. We treat this problem in the next part. A general essay on this topic can be found in [KP00].

Even when not dealing with liveness properties, the problem of false negatives appears. When the abstraction is chosen too coarse, it cannot simulate the concrete behavior exact enough to establish a desired property. This results in an abstract path that violates the property and withdraws from concretization.

Example 6.1 (Coarse abstraction)
Consider again Szymanski's mutual exclusion algorithm. We try to verify the safety property $\Box\neg\xi$ without the abstract variable φ. Remember that φ is essential to control access to the critical section. Using NuSMV we get the counterexample shown in Figure 6.4. Analyzing the trace we find that the last but one step allowed the process with the minimal PID to enter the critical section. Hence it resides in the critical section in the next state. Thus, all processes waiting at location 6 have larger PIDs and have to wait. Unfortunately, the abstraction does not store this information. So, in the

Step	ψ_0	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	ψ_6	ψ_7	ξ
1	1	0	0	0	0	0	0	0	1
2	1	1	0	0	0	0	0	0	1
3	1	0	1	0	0	0	0	0	1
4	1	0	0	1	0	0	0	0	1
5	1	0	0	0	0	1	0	0	1
6	1	0	0	0	0	0	1	0	1
7	1	0	0	0	0	0	1	1	1
8	1	0	0	0	0	0	0	1	0

Figure 6.4: Counterexample for coarse abstraction

abstract system another process may enter the critical section violating the safety property. □

To reconstruct the abstract counterexample and to figure out where the concretization fails is one way to handle false negatives. Another one is called *bounded model-checking* and analyzes a system up to a certain search depth.

6.3.1 Bounded Model-Checking

The term bounded model-checking (BMC) was coined by E. Clarke [CBRZ01] and describes the bounded search for counterexamples. The systems to be verified get more and more complex. Hence, even symbolic model-checkers are not able to compute the whole behavior. A solution is to restrict the number of transitions to be taken. Beside this restriction, BMC is interesting because new techniques can be applied. Especially, SAT-solvers are used to find variable interpretations that validate a predicate describing a path of determined length to a state violating a certain property.

For simplicity, assume we have one predicate ρ describing the transition relation. We want to check whether the property ξ is violated in 3 steps. The initial condition is described by a predicate Θ. Then, we have to find an evaluation for the predicate

$$\Theta(\mathcal{V}) \wedge \rho(\mathcal{V}, \mathcal{V}') \wedge \rho(\mathcal{V}', \mathcal{V}'') \wedge \rho(\mathcal{V}'', \mathcal{V}''') \wedge \neg\xi(\mathcal{V}''') \ .$$

An evaluation validating this predicate also gives us the concrete counterexample. If we are satisfied with a yes/no-answer we can check

$$\exists \mathcal{V}, \mathcal{V}', \mathcal{V}'', \mathcal{V}''' : \Theta(\mathcal{V}) \wedge \rho(\mathcal{V}, \mathcal{V}') \wedge \rho(\mathcal{V}', \mathcal{V}'') \wedge \rho(\mathcal{V}'', \mathcal{V}''') \wedge \neg\xi(\mathcal{V}''') \ .$$

In our case, in WS1S setting both predicates are in WS1S themselves and, hence, can be decided. Moreover, for the first one we get all evaluations satisfying the formula.

6.3.2 Refining the Abstraction Relation

When it is not obvious whether the counterexample has a real counterpart or not, we can use the technique of BMC to get certainty. In case of a false negative BMC can also be used to find the point of divergence where the abstract and the concrete system behave different.

With the same augmentation as in Section 6.2 indicating which transition was taken, we can assume to have an abstract counterexample

$$a_0 \xrightarrow{t_1} a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} a_n \ .$$

We assume the abstraction relation to be given as a WS1S formula $\widehat{\alpha}(\mathcal{V}, VA)$. Now, we can concretize that counterexample and see where it fails:

$$\Theta(\mathcal{V}_0) \wedge \widehat{\alpha}(\mathcal{V}_0, a_0) \wedge \bigwedge_{i=1}^{j} (\widehat{\alpha}(\mathcal{V}_i, a_i) \wedge \rho_{t_i}(\mathcal{V}_{i-1}, \mathcal{V}_i)) \ ,$$

where $1 \leq j \leq n$ indicates how many steps should be concretized.

The same procedure can be applied backwards:

$$\widehat{\alpha}(\mathcal{V}_{j-1}, a_{j-1}) \wedge \bigwedge_{i=j}^{n} (\widehat{\alpha}(\mathcal{V}_i, a_i) \wedge \rho_{t_i}(\mathcal{V}_{i-1}, \mathcal{V}_i)) \ ,$$

for $1 \leq j \leq n$. For $j = 0$ we should check also for the initial condition, but then backward and forward analysis collapse, anyway.

Example 6.2 (Abstraction refinement)
Inspecting Figure 6.4 we see that the trace is very well possible up to the last but one step. Hence, we choose the backward analysis and check for the abstract states

$$\begin{array}{rcl@{\quad}c@{\quad}rcl} a_{n-1} & \equiv \langle & \psi_0 : \mathit{false}, & \text{and} & a_n & \equiv \langle & \psi_0 : \mathit{false}, \\ & & \psi_1 : \mathit{false}, & & & & \psi_1 : \mathit{false}, \\ & & \psi_2 : \mathit{false}, & & & & \psi_2 : \mathit{false}, \\ & & \psi_3 : \mathit{false}, & & & & \psi_3 : \mathit{false}, \\ & & \psi_4 : \mathit{false}, & & & & \psi_4 : \mathit{false}, \\ & & \psi_5 : \mathit{false}, & & & & \psi_5 : \mathit{false}, \\ & & \psi_6 : \mathit{true}, & & & & \psi_6 : \mathit{false}, \\ & & \psi_7 : \mathit{true}, & & & & \psi_7 : \mathit{true}, \\ & & \xi : \mathit{true} \ \rangle & & & & \xi : \mathit{false} \ \rangle \end{array}$$

the formula

$$\widehat{\alpha}(\mathcal{V}, a_{n-1}) \wedge (\widehat{\alpha}(\mathcal{V}', a_n) \wedge \rho_{t_{67}}(\mathcal{V}, \mathcal{V}')) \ .$$

The formula is satisfiable. The satisfying interpretations show that the process with the smallest ID enters its critical section.

Going another step backwards and taking

$$a_{n-2} \equiv \langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true} \rangle$$

into account, the formula

$$\widehat{\alpha}(\mathcal{V}, a_{n-2}) \wedge (\widehat{\alpha}(\mathcal{V}', a_{n-1}) \wedge \rho_{t_{67}}(\mathcal{V}, \mathcal{V}')) \wedge (\widehat{\alpha}(\mathcal{V}'', a_n) \wedge \rho_{t_{67}}(\mathcal{V}', \mathcal{V}''))$$

is found to be unsatisfiable. Hence, it is necessary to add another abstract variable storing the information that the process occupying location 7 is the one with the smallest PID. □

How to use the information gained during that backtracking procedure to refine the abstraction relation is an ongoing research topic [CGJ+00, CCK+02, CGKS02, LBOB01].

Part III

Checking Liveness Properties

Chapter 7

Fairness Conditions

An obstacle to the verification of liveness properties using abstraction is that the abstract system often contains cycles which do not correspond to fair computations of the concrete system. A way to overcome this difficulty is to enrich the abstract system with fairness conditions or, more generally, to use ranking functions over well-founded sets to eliminate undesirable computations. We present a method to enrich the abstract system with strong fairness conditions while preserving the property that each concrete computation corresponds to an abstract *fair* one. Then, the enriched abstract system can be used to prove liveness properties of the concrete WS1S system and, consequently, of the original parameterized network.

First, we discuss under which circumstances fairness conditions given for the original system can be lifted to the finite abstract one. In particular, we show that by introducing additional requirements on the abstraction relation, it is sound to lift strong fairness. Weak fairness can only be lifted for a distinguished process on which the abstraction focuses.

7.1 Lifting Fairness Conditions

In order to simulate the behavior of parameterized networks with fairness conditions we need the notion of fairness for WS1S transition systems.

Definition 7.1 (Fair WS1S transition system)
A *fair WS1S transition system* $\mathcal{W}^{\mathcal{F}} = (\mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$ is defined by the following components:

- $\mathcal{V} = \{X_1, \ldots, X_k\}$: A finite set of second order variables X_i ranging over finite sets of natural numbers.

- Θ: A WS1S formula with $\mathit{free}(\Theta) \subseteq \mathcal{V}$ describing the initial condition of the system.
- $\mathcal{T}$: A finite set of transitions where each $\tau \in \mathcal{T}$ is represented as a WS1S formula $\rho_\tau(\mathcal{V}, \mathcal{V}')$, i.e., $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}'$.
- $\mathcal{J}$: A set of pairs of second order variables expressing a weak fairness condition. For each pair $(X_m, X_{m'})$ it is required that, for each $p \in \omega$, the weak fairness condition that p cannot be continuously in X_m without being eventually in $X_{m'}$, that is, $\forall p \in \omega : (\Diamond\Box(p \in X_m) \rightarrow \Box\Diamond(p \in X_{m'}))$.
- $\mathcal{C}$: A set of pairs $(X_m, X_{m'})$ of second order variables expressing the strong fairness condition $\forall p \in \omega : (\Box\Diamond(p \in X_m) \rightarrow \Box\Diamond(p \in X_{m'}))$. □

A state s of the WS1S system $\mathcal{W}^{\mathcal{F}}$, in particular, is a mapping from the variables in $\mathcal{V}$ into finite sub-sets of ω. Hence, a *computation* σ of $\mathcal{W}^{\mathcal{F}}$ is a sequence $(s_i)_{i\in\omega}$ of states such that $\Theta[s_0(\mathcal{V})/\mathcal{V}]$ and $\bigvee_{\tau\in\mathcal{T}} \tau[s_i(\mathcal{V}), s_{i+1}(\mathcal{V})/\mathcal{V}, \mathcal{V}']$ are valid formulae.

A computation $(s_i)_{i\in\omega}$ satisfies a weak fairness condition $(X_m, X_{m'}) \in \mathcal{J}$ iff the following condition holds for every $p \in \omega$:

> *if* $\exists i \in \omega.\forall j \geq i : p \in s_j(X_m)$*, then there exist infinitely many* i *such that* $p \in s_i(X_{m'})$*.*

The computation satisfies the strong fairness condition $(X_m, X_{m'}) \in \mathcal{C}$ iff the following condition holds for every $x \in \omega$:

> *if there exist infinitely many* i *such that* $p \in s_i(X_m)$*, then there exist infinitely many* i *such that* $p \in s_i(X_{m'})$*.*

Then a *fair computation* of $\mathcal{W}^{\mathcal{F}}$ is a computation that satisfies all fairness conditions in $\mathcal{J}$ and $\mathcal{C}$. Henceforth, we denote the set of fair computations of $\mathcal{W}^{\mathcal{F}}$ by $[\![\mathcal{W}^{\mathcal{F}}]\!]$.

In order to translate fair MPS into fair WS1S systems we have to enhance the translation function *Tr* in Definitions 3.2 and 3.7 to include the fairness conditions.

Definition 7.2 (Translation of MPS into fair WS1S systems) Consider an asynchronous MPS system (with global variables) $\mathcal{P}$ built from $S(i, n)$ where $\mathcal{S}(i, n) = (V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$. Define a WS1S system $(\widetilde{V}, \widetilde{\mathcal{T}}, \widetilde{\Theta}, \widetilde{\mathcal{J}}, \widetilde{\mathcal{C}})$ by constructing the variable set, the initial condition, the transitions, and the fairness requirements as follows:

- For each boolean array b_k in V, $\widetilde{V}$ contains the variable B_k. Additionally, $\widetilde{V}$ contains the set variable P. For any global variable a, if present, a is also in $\widetilde{V}$. For every $\tau \in \mathcal{T}$ we add the two set variables E_τ and T_τ to keep track of enabled and taken transitions.

- Let $\widetilde{\Theta}$ be

$$\exists n : P = \{0, \ldots, n-1\} \wedge \bigcup_{l=1}^{k} B_l \subseteq P \wedge (\forall_P i : tr(\Theta)) \wedge \\ \bigwedge_{\tau \in \mathcal{T}} (E_\tau = \{i \in P \mid \exists \mathcal{V}'' : \rho_\tau(\mathcal{V}', \mathcal{V}'', i)\} \wedge T_\tau = \emptyset) \ .$$

- Let $\widetilde{\mathcal{T}}$ be the set

$$\{ \quad \exists_P i : tr(\rho_\tau) \wedge P = P' \wedge \bigcup_{l=1}^{k} B'_l \subseteq P' \wedge \\ T'_\tau = \{i\} \wedge \bigwedge_{\tau \neq \tau' \in \mathcal{T}} T'_{\tau'} = \emptyset \\ \bigwedge_{\tau' \in \mathcal{T}} E'_{\tau'} = \{i \in P \mid \exists \mathcal{V}'' : \rho_{\tau'}(\mathcal{V}', \mathcal{V}'', i)\} \quad \mid \tau \in \mathcal{T}\} \ .$$

- Let $\widetilde{\mathcal{J}} = \{(E_\tau, T_\tau) \mid \tau \in \mathcal{J}\}$.

- Let $\widetilde{\mathcal{C}} = \{(E_\tau, T_\tau) \mid \tau \in \mathcal{C}\}$. □

Since the variables E_τ and T_τ do not appear in any guards, it is clear that Theorem 3.5 establishing h as a bisimulation still holds. We only have to extend h such that it provides the adequate valuations for E_τ and T_τ. However, to propagate Corollary 3.6 we have to check that the fairness conditions exclude the same traces.

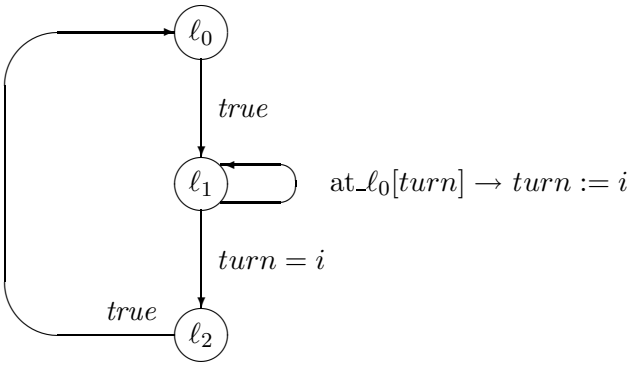
Lemma 7.3 (Equivalence of $\mathcal{P}^{\mathcal{F}}$ and $\mathcal{W}^{\mathcal{F}}$)
Let $\mathcal{P}^{\mathcal{F}}$ be an MPS built from $\mathcal{S}^{\mathcal{F}}(i, n)$. Then, lifting h to computations, we have a bijection between $\bigcup_{m \in \omega} [\![\mathcal{P}^{\mathcal{F}}(m)]\!]$ and $[\![Tr(\mathcal{P}^{\mathcal{F}})]\!]$.

Proof In the definition of fair WS1S systems we have deliberately chosen the semantics for the fairness requirements to exactly match those of fair transition systems. The translation in Definition 7.2 alters the sets E_τ and T_τ for a transition τ such that E_τ contains the processes for which τ is enabled and T_τ contains the process that has taken the transition if such a process exists. Hence, the fairness requirements correspond exactly to those of the MPS. ■

behavior of a certain process except for the control process. The guards only

As an illustrating example we use the simple mutual exclusion algorithm from Examples 2.9 and 3.9 to illustrate how to analyze fair transition systems.

Example 7.4 (Simple ME algorithm)
The parameterized network consists of processes which can be characterized graphically as follows:



The loop τ_{11} from ℓ_1 to ℓ_1 is strong fair, i.e., $\mathcal{C} = \{\tau_{11}\}$, whereas all other transitions are weak fair, i.e., $\mathcal{J} = \{\tau_{01}, \tau_{12}, \tau_{20}\}$

We want to verify that the algorithm satisfies the mutual exclusion property as well as the universal property that each process p reaches its critical section infinitely often, i.e., $\forall_n p : \Box\Diamond \text{at\_}\ell_2[p]$.

As in Example 3.9, we represent this network as a WS1S system by introducing three set variables $\text{At\_}\ell_0, \text{At\_}\ell_1, \text{At\_}\ell_2$ corresponding to the control locations, the first-order variable $turn$, and a set variable P representing the set of processes being part of the network. As required by Definition 7.2, we need two additional set variables E_τ and T_τ for each transition to express the fairness conditions. Let $\widetilde{\mathcal{V}}$ denote this set of variables.

For the self-loop τ_{11} at ℓ_1, we then have $\mathcal{C} = \{(E_{\tau_{11}}, T_{\tau_{11}})\}$. For the sake of illustration, we show the representation of τ_{11}:

$$\exists_P i : \rho_{\tau_{11}}(\mathcal{V}, \mathcal{V}', i) \wedge \bigwedge_{\tau \in \mathcal{T}} E'_\tau = \{i \in P \mid \exists \mathcal{V}'' : \rho_\tau(\mathcal{V}', \mathcal{V}'', i)\} \ ,$$

where $\rho_\tau(\mathcal{V}', \mathcal{V}'', i)$ characterizes those $i \in P$ which can take a τ-step and $\rho_{\tau_{11}}(\mathcal{V}, \mathcal{V}', i)$ is defined as:

$$\begin{aligned} \rho_{\tau_{11}}(\mathcal{V}, \mathcal{V}', i) \ \equiv \ & i \in \text{At\_}\ell_1 \wedge turn \in \text{At\_}\ell_0 \wedge i \in \text{At\_}\ell'_1 \wedge i = turn' \\ & \wedge (\forall_P j : j \neq i \rightarrow \bigwedge_{k=0,1,2} (j \in \text{At\_}\ell_k \leftrightarrow j \in \text{At\_}\ell'_k)) \\ & \wedge P = P' \wedge \bigcup_{B \in \widetilde{V}} B' \subseteq P' \wedge \bigcup_{\tau \neq \tau_0} T'_\tau = \emptyset \wedge T'_{\tau_0} = \{i\} \ . \end{aligned}$$

The liveness property we want to check can be expressed by $\forall_P p : \Box\Diamond(p \in \text{At\_}\ell_2)$. □

Note that Definition 7.2 only concerns asynchronous systems. The modeling of fair, synchronous MPS is straightforward: one has to change the set variables E_τ and T_τ accordingly. As a consequence the sets T_τ may contain more than one process identifier. Since we have no example of a fair, synchronous MPS we omit a proper definition of a translation.

7.2 Abstraction with Fairness

We have extended the definition of WS1S systems to include the notion of fairness. In our abstraction-based framework we now have to express fairness also at the abstract level. Of course, the method to lift fairness conditions to the abstract level depends on the abstraction relation chosen.

Consider a local abstraction focusing on two processes p and q. Then, we have for each set X in the WS1S system $\mathcal{W}$ the abstract boolean variables $p_X, q_X \in \mathcal{V}_A$ corresponding to $p, q \in X$. Thus, in particular we have the variables p_{E_τ}, q_{E_τ} and p_{T_τ}, q_{T_τ}, for each $\tau \in \mathcal{T}$. For notational brevity we denote them with e^p_τ, e^q_τ and t^p_τ, t^q_τ.

Additionally, for each strong fairness condition $(E_\tau, T_\tau) \in \mathcal{C}$ we introduce boolean variables e_τ, t_τ such that α implies:

$$\begin{aligned} e_\tau &\equiv \exists_P\, j : j \in E_\tau \\ t_\tau &\equiv \exists_P\, j : j \in T_\tau \ . \end{aligned}$$

7.2.1 The Fair Abstract System

For the rest of the section, we fix a WS1S system $\mathcal{W}^{\mathcal{F}} = (\mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$ modeling a parameterized network and an abstraction relation α that contains at least those constructs as explained in the previous paragraph. Then, let $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ be the finite abstract system (without fairness) obtained by the method introduced in Chapter 5.

We show how to add fairness conditions to $\mathcal{S}_A$ leading to a fair abstract system $\mathcal{S}^{\mathcal{F}}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A, \mathcal{J}_A, \mathcal{C}_A)$ such that $\mathcal{W}^{\mathcal{F}} \sqsubseteq^{\mathcal{F}}_\alpha \mathcal{S}^{\mathcal{F}}_A$. First we have to define how fairness requirements have to be interpreted for abstract transition systems. In our framework the abstract systems exclusively contain boolean variables. To correspond with the definition of fairness for WS1S systems we allow pairs (b_1, b_2) as weak or strong fairness conditions.

A computation $(s_i)_{i\in\omega}$ satisfies a weak fairness condition $(b_1, b_2) \in \mathcal{J}_A$ iff the following condition holds:

> *if* $\exists i \in \omega . \forall j \geq i : s_j(b_1)$*, then there exist infinitely many* i *such that* $s_i(b_2)$*.*

The computation satisfies the strong fairness condition $(b_1, b_2) \in \mathcal{C}_A$ iff the following condition holds:

> *if there exist infinitely many i such that $s_i(b_1)$, then there exist infinitely many i such that $s_i(b_2)$.*

Then, a *fair computation* of $\mathcal{S}_A^{\mathcal{F}}$ is a computation that satisfies all fairness conditions in $\mathcal{J}_A$ and $\mathcal{C}_A$.

Now, we can consider to lift fairness conditions to the abstract level. To define α (see Chapter 5), we have now additionally introduced the abstract variables $e_\tau \equiv \exists_P i : i \in E_\tau$ and $t_\tau \equiv \exists_P i : i \in T_\tau$. We now argue that it is safe to augment $\mathcal{S}_A$ with the strong fairness requirements $\mathcal{C}_A = \{(e_\tau, t_\tau) \mid (E_\tau, T_\tau) \in \mathcal{C}\}$, i.e., if e_τ is *true* infinitely often, then also t_τ is *true* infinitely often. Consider a computation where e_τ is *true* infinitely often, that is, $\exists_P i : i \in E_\tau$ is *true* infinitely often.

Now, each instance of the parameterized system only contains a bounded number of processes, hence, by König's lemma, there exists some i such that $i \in E_\tau$ holds infinitely often in this computation. Therefore, by the strong fairness condition of the concrete system, we must have $i \in T_\tau$ infinitely often, and hence, the computation satisfies $\Box\Diamond(\exists_P i : i \in T_\tau)$. Consequently:

Lemma 7.5 (Lifting strong fairness)
Under the assumptions above we have $\mathcal{W}^{\mathcal{F}} \sqsubseteq_\alpha^{\mathcal{F}} \mathcal{S}_A^{\mathcal{F}}$. □

The reasoning above does not hold for weak fairness. Indeed, $\Diamond\Box e_\tau$ may hold for a computation without the existence of an i with $\Diamond\Box(i \in E_\tau)$.

For each transition of a distinctive process p we have the abstract variables e_τ^p and t_τ^p expressing whether the transition is enabled for p, respectively, taken by p. We can show that it is safe to augment the abstract system with strong and weak fairness conditions on the transitions of p.

Lemma 7.6 (Lifting weak fairness)
For the concrete WS1S system $\mathcal{W}^{\mathcal{F}}$ and the abstract system $\mathcal{S}_A^{\mathcal{F}}$ we have:

$$\mathcal{W}^{\mathcal{F}} \sqsubseteq_\alpha^{\mathcal{F}} \mathcal{S}_A^{\mathcal{F}} \ ,$$

where $\mathcal{S}_A^{\mathcal{F}}$ has the strong fairness requirements $\mathcal{C}_A = \{(e_\tau^p, t_\tau^p) \mid (E_\tau, T_\tau) \in \mathcal{C}\}$ and the weak fairness requirements $\mathcal{J}_A = \{(e_\tau^p, t_\tau^p) \mid (E_\tau, T_\tau) \in \mathcal{J}\}$.

Proof Consider a computation σ in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ that is excluded by a fairness requirement in $\mathcal{J}_A$. Hence, there exists a transition τ such that e_τ^p is continuously *true* in σ from a certain point and t_τ^p never becomes *true*. p is meant to keep track of a certain process. Thus, we have two possibilities: either

a concretization $\widetilde{\sigma}$ of σ actually takes the same process in every step, or it switches processes during concretization. In the first case we can safely remove σ from $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$. In the other case, we find another abstract computation σ' for $\widetilde{\sigma}$ which consequently focuses on the same process. Hence, it does no harm to remove σ in any case. The same argumentation also holds for strong fairness. ∎

As for fair MPS, also in the case of fair abstract systems $\mathcal{S}_A^{\mathcal{F}}$ we sometimes express the weak and strong fairness requirements $\mathcal{J}$ and $\mathcal{C}$ as a single LTL formula $\mathcal{F}$ and denote the fair abstract system as a quadruple $\mathcal{S}_A^{\mathcal{F}} = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A, \mathcal{F})$.

7.3 Generating Fairness Conditions

We now have the possibility to express fairness on the abstract level. Moreover, we have shown which fairness requirements can be lifted and how to achieve this. There still remains the problem of cycles occurring in the abstract system that do not have any concrete counterpart. To remove those cycles we want to generate fairness conditions that can be safely required for the abstract system, since these conditions are guaranteed to hold for the concrete system. Hence, for those fairness conditions we have $\mathcal{W} \sqsubseteq^{\mathcal{F}} \mathcal{S}_A^{\mathcal{F}}$.

To concentrate on the topic of generated fairness, we assume $\mathcal{W}$ to be a WS1S system without fairness requirements, i.e, throughout this section, we fix a WS1S system $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$ and an abstraction function α as introduced in Chapter 5. Then, let $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ be the finite abstract system obtained by the method also explained in Chapter 5. We show how to extend $\mathcal{S}_A$ leading to a fair abstract system $\mathcal{S}_A^{\mathcal{F}} = (\mathcal{V}_A^+, \Theta_A^+, \mathcal{T}_A^+, \mathcal{F})$, respectively, $\mathcal{S}_A^{\mathcal{F}} = (\mathcal{V}_A^+, \Theta_A^+, \mathcal{T}_A^+, \mathcal{J}, \mathcal{C})$, such that $\mathcal{S} \sqsubseteq_\alpha^{\mathcal{F}} \mathcal{S}_A^{\mathcal{F}}$.

We use WS1S formulae to express ranking functions. Let $\chi(i, X_1, \cdots, X_k)$ be a predicate with i as free first-order variable and $X_1, \cdots, X_k \in \mathcal{V}$ as free second-order variables. Given a state s of $\mathcal{S}$, i.e., a valuation of the variables in $\mathcal{V}$, the ranking value $\zeta_\chi(s)$ associated to s by χ is the cardinality of $\{i \in \omega \mid \chi(i, s(X_1), \ldots, s(X_k))\}$.

Having defined a ranking predicate χ we extend the abstract system with boolean variables $\{+_\chi, -_\chi\}$. Intuitively, the abstract boolean variable $-_\chi$ is set to *true*, if it is guaranteed that the concrete transition τ associated with an abstract step decreases the ranking value, i.e., $(s, s') \in \tau$ implies $\zeta(s) > \zeta(s')$. Similarly, $+_\chi$ is set to *true* whenever a transition may increase the ranking value, i.e., $\zeta(s) < \zeta(s')$.

Hence, for a set of such ranking functions we get a new set of abstract variables $\mathcal{V}_A^+$ containing all these pairs of new boolean variables. It is not

necessary to restrict these variables initially since they are used to describe the systems' infinite behavior, i.e., we choose $\Theta_A^+ \equiv \Theta_A$.

For each $\chi(i, X_1, \cdots, X_k)$ and each τ in $\mathcal{T}$ we add the following conjuncts to the definition of the abstraction transitions:

$$+_\chi \equiv \{i \mid \chi'(i)\} \not\subseteq \{i \mid \chi(i)\} \wedge$$
$$-_\chi \equiv \{i \mid \chi'(i)\} \subset \{i \mid \chi(i)\} \ .$$

This yields a new set $\mathcal{T}_A^+$ of abstract transitions which update the added variables such that they indicate the decrease or increase of the corresponding ranking values.

Finally, we define

$$\mathcal{F} \equiv \bigwedge_\chi \Box\Diamond -_\chi \rightarrow \Box\Diamond +_\chi \ .$$

Obviously, this fairness condition can also be represented as strong fairness requirement $\mathcal{C} = \{(-_\chi, +_\chi) \mid \chi \text{ is a ranking predicate}\}$.

The next lemma justifies to use the constructed fairness condition $\mathcal{F}$ to restrict our system as it shows that the set of fair computations still over-approximates the set of concrete computations.

Lemma 7.7 ($\mathcal{W} \sqsubseteq_\alpha^\mathcal{F} \mathcal{S}_A^\mathcal{F}$)
The extended fair abstract system $\mathcal{S}_A^\mathcal{F} = (\mathcal{V}_A^+, \Theta_A^+, \mathcal{T}_A^+, \mathcal{F})$ with $\mathcal{F}$ being the fairness constraint, satisfies $\mathcal{W} \sqsubseteq_\alpha^\mathcal{F} \mathcal{S}_A^\mathcal{F}$.

Proof We have to show $\alpha^{-1}([\![\neg\mathcal{F}]\!]) \subseteq \overline{[\![\mathcal{W}]\!]}$. Let $\sigma : \sigma_1, \sigma_2, \ldots$ be a sequence of evaluations of $\mathcal{V}_A^+$ such that $\sigma \in [\![\neg\mathcal{F}]\!]$. I.e., we have $\sigma \models \Box\Diamond -_\chi \wedge \Diamond\Box\neg +_\chi$ for some ranking predicate $\chi(i, X_1, \cdots, X_k)$. The concretization $\alpha^{-1}(\sigma)$ induces a sequence of sets

$$\{i \mid \chi(i, \sigma_1(X_1), \ldots, \sigma_1(X_k))\}, \{i \mid \chi(i, \sigma_2(X_1), \ldots, \sigma_2(X_k))\}, \ldots \ .$$

Since

$$+_\chi \equiv \{i \mid \chi'(i)\} \not\subseteq \{i \mid \chi(i)\} \wedge -_\chi \equiv \{i \mid \chi'(i)\} \subset \{i \mid \chi(i)\}$$

is part of our abstraction function, we know by $\sigma \models \Box\Diamond -_\chi \wedge \Diamond\Box\neg +_\chi$ that this sequence is monotonously decreasing from a certain point on. Moreover, it decreases infinitely often, which, of course, contradicts the well-foundedness of sets. Hence, the lemma holds. ∎

The ranking predicates are very powerful as we will see in Chapter 9, since they decide state-dependently whether the ranking value decreases or not.

Unfortunately, it is also very costly to enrich the abstract system in such a way. As a simplification we could check for a given concrete transition whether it increases or decreases a certain ranking value regardless of the actual state. Of course, this is not as exact as the method presented so far, but it can be checked separately. Then, the fairness condition states that transitions moving processes away from certain locations cannot be taken infinitely often, if no other transitions that lead processes back to that location are taken infinitely often. We only have to enrich the abstract system with flags indicating which transition was taken last. Therefore, we model the WS1S system as a fair system introducing set variables E_τ and T_τ for each transition τ. Then we can add $t_\tau \equiv \exists_P i : i \in T_\tau$ to our abstraction to obtain such flags.

7.3.1 Marking Algorithm

We call the method of checking concrete transitions for increasing or decreasing certain sets *marking algorithm*. The reason is that it labels each abstract transition of the abstract system with one of the symbols $\{+_\chi, -_\chi\}$. Intuitively, an abstract transition τ_A is labeled by $-_\chi$, if it is guaranteed that the concrete transition τ associated with τ_A decreases the ranking value, i.e., $(s, s') \in \tau$ implies $\zeta(s) > \zeta(s')$. The label $+_\chi$ denotes that τ may increase the value for some concrete state.

Input: WS1S system $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$, abstraction $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$, set of predicates $\chi(i, X_1, \cdots, X_k)$

Output: Labeling of $\mathcal{T}_A$

Description: For each $\chi(i, X_1, \cdots, X_k)$, for each edge $\tau_A \in \mathcal{T}_A$, let τ be the concrete transition in $\mathcal{T}$ corresponding to τ_A.

Mark τ_A with $-_\chi$ if the following formula is valid:

$$\forall \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \rightarrow \{i \mid \chi'(i)\} \subset \{i \mid \chi(i)\} \ .$$

Mark τ_A with $+_\chi$ if

$$\exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \rightarrow \{i \mid \chi'(i)\} \not\subseteq \{i \mid \chi(i)\}$$

is valid.

Now, for a ranking predicate χ we denote with $\mathcal{T}_\chi^+$ the set of edges labeled with $+_\chi$. Then, we add for each χ and each transition τ_A labeled with $-_\chi$ the

fairness condition $(\tau_A, \mathcal{T}_\chi^+)$ which states that τ_A can only be taken infinitely often when one of the transitions in $\mathcal{T}_\chi^+$ is taken infinitely often.

In the sequel we assume to have introduced additional variables: for each transition τ the abstract system contains a boolean flag t_τ set to *true* when τ is taken and to *false* for any other transition being taken. Then we can express the fairness condition $(\tau_A, \mathcal{T}_\chi^+)$ generated by the marking algorithm as

$$\mathcal{F} \equiv \Box\Diamond t_{\tau_A} \rightarrow \Box\Diamond \bigwedge_{\tau \in \mathcal{T}_\chi^+} t_\tau \ .$$

7.3.2 Heuristics

Finally, we need to come up with some heuristics for choosing the ranking predicates. The idea is already mentioned and illustrated in Section 6.2. There, the ad hoc fairness requirement results from applying ranking predicates $\chi_{\mathrm{At\_}\ell_k} \equiv i \in \mathrm{At\_}\ell_k$, $0 \leq k \leq 7$. Each of them introduces abstract variables $+_\chi$ and $-_\chi$. Speaking in terms of the WS1S system, each transition of Szymanski's algorithm takes a process identifier out of one set to put it into another. Hence, the transition τ_{01} decreases the set $\mathrm{At\_}\ell_0$ and increases $\mathrm{At\_}\ell_1$; accordingly the auxiliary variables $-_{\mathrm{At\_}\ell_0}$ and $+_{\mathrm{At\_}\ell_1}$ are set to *true*.

When using the marking algorithm we can omit these variables and use the *taken* variables instead. For instance, transition τ_{01} gets labeled with $+_{\mathrm{At\_}\ell_1}$ and $-_{\mathrm{At\_}\ell_0}$. The transition τ_{70} is the only transition that gets labeled with $-_{\mathrm{At\_}\ell_0}$. Hence, we can state $\Box\Diamond t_{01} \rightarrow \Box\Diamond t_{70}$. Running the marking algorithm with the eight ranking predicates $\chi_{\mathrm{At\_}\ell_k} \equiv i \in \mathrm{At\_}\ell_k$, $0 \leq k \leq 7$ we get the fairness stated in Section 6.2.

Hence, as a heuristic we propose to take the set variables $\mathrm{At\_}\ell$ expressing the control flow to define ranking predicates. The cardinality of $\mathrm{At\_}\ell$ characterizes a sufficient ranking value to exclude unbounded application of a single transition (that is not a loop). Further, we suggest to look for cycles in the control flow. E.g., for a cycle ℓ_1, ℓ_2, ℓ_3, we advise to take

$$\chi(i, \mathrm{At\_}\ell_1, \mathrm{At\_}\ell_2, \mathrm{At\_}\ell_3) \equiv i \in \mathrm{At\_}\ell_1 \cup \mathrm{At\_}\ell_2 \cup \mathrm{At\_}\ell_3$$

as a ranking predicate.

Example 7.8 (Individual accessibility for Simple ME algorithm) Recall that we want to verify that our algorithm satisfies the mutual exclusion property as well as the universal property that each process p reaches its critical section infinitely often, i.e., $\forall_P\, p : \Box\Diamond p \in \mathrm{At\_}\ell_2$.

According to the method presented in Chapter 5 we construct the abstract system $\mathcal{S}_A$ from the WS1S translation presented in Example 7.4. We choose a local abstraction focusing on one process p. For the mutual exclusion property we take as abstract variable

$$inv \equiv \mathrm{At\_}\ell_2 \subseteq Turn \wedge \forall_P\, i, j : (i \in Turn \wedge j \in Turn) \rightarrow i = j \ .$$

Moreover, the abstract system contains variables e^p_τ and t^p_τ to express which transitions are enabled for p or were taken by p. Globally, we need the abstract variables t_τ to keep track of which transition was taken.

Our tool PAX constructs the abstract system and provides translations to several input languages for model-checkers, e.g., Spin and SMV. Also, the abstract state space can be explored to prove that inv is indeed an invariant of the abstract system and, hence, mutual exclusion holds for the original system.

Next, we augment $\mathcal{S}_A$ with abstract variables according to the ranking predicates $\chi_{\mathrm{At\_}\ell_0}$, $\chi_{\mathrm{At\_}\ell_1}$, and $\chi_{\mathrm{At\_}\ell_2}$. The strong fairness requirements $\{(-_{\mathrm{At\_}\ell_0}, +_{\mathrm{At\_}\ell_0}), (-_{\mathrm{At\_}\ell_1}, +_{\mathrm{At\_}\ell_1}), (-_{\mathrm{At\_}\ell_2}, +_{\mathrm{At\_}\ell_2})$ can be added safely according to Lemma 7.7.

Alternatively, we can use the marking algorithm and get the equivalent strong fairness requirement $\{(t_{01}, t_{20}), (t_{12}, t_{01}), (t_{20}, t_{12})\}$.

Moreover, with Lemma 7.5 we can lift the strong fairness (e_{11}, t_{11}). For the process we focus on we use Lemma 7.6 to augment $\mathcal{S}^{\mathcal{F}}_A$ with the strong fairness condition (e^p_{11}, t^p_{11}) as well as with the weak fairness requirement (e^p_{01}, t^p_{01}).

All the fairness conditions can be expressed as LTL formulae. We used Spin to prove that $\Box\Diamond p_{\mathrm{At\_}\ell_2}$ holds in $\mathcal{S}^{\mathcal{F}}_A$ which means that, in the original system, each process reaches its critical section infinitely often. □

Chapter 8

Examples

In this chapter we present several examples and our verification results. First, we give a summary of the full verification results of Szymanski's mutual exclusion algorithm. Beyond recapitulating the results obtained so far, we apply the technique of generating fairness conditions of Chapter 7 in order to prove individual accessibility for Szymanski's algorithm. To our knowledge this is the first time that also individual liveness is automatically proven for this algorithm. As a second example, a mutual exclusion algorithm inspired by Dijkstra's algorithm is verified. The verification of this algorithm strongly relies on lifting a lot of fairness conditions to the abstract system to be able to prove individual accessibility on the abstract level. Finally, we show the verification of a token passing algorithm. There, we illustrate how to handle such algorithms generally by expressing the ring topology as a fairness condition that states that every process owns the token infinitely often.

8.1 Szymanski's Mutual Exclusion Algorithm

We first recapitulate the pseudo-code characterization of Szymanski's algorithm [Szy88] as presented in Example 2.10.

The code for one process of Szymanski's mutual exclusion algorithm is given as:

$\mathcal{S}(i,n) \equiv$ **loop forever do**
ℓ_0: **noncritical section**
ℓ_1: **await** $\forall_n\, j : \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j] \vee \mathrm{at}\_\ell_4[j]$
ℓ_2: **skip**
ℓ_3: **if** $\exists_n\, j : \mathrm{at}\_\ell_1[j] \vee \mathrm{at}\_\ell_2[j]$
then goto ℓ_4
else goto ℓ_5

ℓ_4: **await** $\exists_n\, j : \text{at\_}\ell_5[j] \vee \text{at\_}\ell_6[j] \vee \text{at\_}\ell_7[j]$
ℓ_5: **await** $\forall_n\, j : \neg(\text{at\_}\ell_3[j] \vee \text{at\_}\ell_4[j])$
ℓ_6: **await** $\forall_n\, j : j < i : \text{at\_}\ell_0[j] \vee \text{at\_}\ell_1[j] \vee \text{at\_}\ell_2[j]$
ℓ_7: **critical section**;
od

The control flow is modeled by a boolean variable $\text{at\_}\ell_k[p]$ for each location ℓ_k and each process i. The initial condition states that each process starts in ℓ_0.

Note that the **skip** transition results from removing the original boolean variables a, s, w (see Example 2.3) and replacing them by expressions over the control variables $\text{at\_}\ell$ in the guards. Hence, the transition should not be omitted, otherwise we would lose the correspondence to the original algorithm. It represents the entry to the waiting room where all processes gather to enter their critical section.

8.1.1 Properties of Interest

To verify the algorithm we identify the following properties which we would like to establish to claim the algorithm to be correct:

Safety: As a first verification goal we have to establish that there never are two processes in the critical section. This can be specified as

$$\forall_n\, p, q, p \neq q : \Box\neg(\text{at\_}\ell_7[p] \wedge \text{at\_}\ell_7[q]) \ .$$

Communal accessibility: As a first liveness behavior one wants a mutual exclusion algorithm to satisfy communal accessibility, i.e., whenever a process wants to enter the critical section eventually one does so:

$$\Box((\exists_n\, p : \text{at\_}\ell_1[p]) \rightarrow \Diamond(\exists_n\, q : \text{at\_}\ell_7[q])) \ .$$

Individual accessibility: In contrast, the stronger property

$$\forall_n\, p : \Box(\text{at\_}\ell_1[p] \rightarrow \Diamond\text{at\_}\ell_7[p])$$

is called individual accessibility. I.e., each process that wants to enter the critical section eventually does so.

Linear waiting: Another property quantifies the time that a process has to wait for access in the worst case. In Szymanski's algorithm one expects a process to be overtaken by those processes with lower PID in the worst case. Hence, the number of overtaking processes is bounded by

the number n of participating processes. The property of linear waiting can be formalized by using the $\mathcal{U}$-operator:

$$\forall_n\, p, q : \Box\ (\mathrm{at}\_\ell_1[p] \wedge (\mathrm{at}\_\ell_0[q] \vee \mathrm{at}\_\ell_1[q]) \rightarrow \\ ((((\neg \mathrm{at}\_\ell_7[q])\, \mathcal{U}\, \mathrm{at}\_\ell_7[q])\, \mathcal{U}\, (\neg \mathrm{at}\_\ell_7[q]))\, \mathcal{U}\, \mathrm{at}\_\ell_7[p]))$$

The formula states that for a process p that wants to enter its critical section, another process q can overtake p at most once and enter its critical section first.

8.1.2 Global Abstraction

To verify these properties according to our abstraction-based verification approach we have to translate the MPS defined above into a WS1S system. The translation *Tr* introduces set variables At_ℓ_k for each control location ℓ_k, i.e., for each boolean variable at_ℓ_k of the MPS. Except for the fairness conditions the translation is shown in Example 3.3.

A global abstraction is presented in Example 5.3. There we have

$$\psi_i \equiv \mathrm{At}\_\ell_i \neq \emptyset, \text{ for each } 1 \leq i \leq 7$$

to keep track of the locations in which processes reside. It is not important to store the information whether there is a process in the initial location, since that location represents the **noncritical** part.

Moreover we have the abstract variable

$$\varphi \equiv \exists i : i \in \mathrm{At}\_\ell_7 \wedge \forall j < i : j \in \mathrm{At}\_\ell_1 \cup \mathrm{At}\_\ell_2 \cup \mathrm{At}\_\ell_4$$

and one for the property of interest

$$\xi \equiv \neg \exists l, j : l \neq j \wedge l \in \mathrm{At}\_\ell_7 \wedge j \in \mathrm{At}\_\ell_7\ .$$

By decidability of WS1S we are able to construct an abstract system $\mathcal{S}_A$. Its abstract state graph is presented in Figure 6.2. Eye inspection allows to verify that ξ always holds. ξ represents the desired safety property which is established by Theorem 3.5 and Theorem 5.2.

As Section 6.2 shows, verification of communal accessibility fails because of loops that are bounded at the concrete level by the number of participating processes. To handle this problem at the abstract level we introduce additional fairness conditions generated by the marking algorithm (see Section 7.3.1). Applying the marking algorithm to ranking predicates $\chi \equiv i \in \mathrm{At}\_\ell_k$ results in fairness conditions that express that transitions decreasing the number of processes at location ℓ_k can only be taken infinitely

often when taking other transitions increasing that number. Augmenting the abstract system $\mathcal{S}_A$ with abstract variables $t_\tau \equiv \exists i : i \in T_\tau$, the generated fairness condition $\mathcal{F}$ can be expressed as:

$$\begin{array}{lll}
(\Box\Diamond t_{01} \rightarrow \Box\Diamond t_{70}) & \wedge & \\
(\Box\Diamond t_{12} \rightarrow \Box\Diamond t_{01}) & \wedge & (\Box\Diamond t_{23} \rightarrow \Box\Diamond t_{12}) \wedge \\
(\Box\Diamond t_{34} \rightarrow \Box\Diamond t_{23}) & \wedge & (\Box\Diamond t_{35} \rightarrow \Box\Diamond t_{23}) \wedge \\
(\Box\Diamond t_{45} \rightarrow \Box\Diamond t_{34}) & \wedge & (\Box\Diamond t_{56} \rightarrow \Box\Diamond (t_{45} \vee t_{35})) \wedge \\
(\Box\Diamond t_{67} \rightarrow \Box\Diamond t_{56}) & \wedge & (\Box\Diamond t_{70} \rightarrow \Box\Diamond t_{67})
\end{array}$$

Then, we can prove that $\mathcal{S}_A^{\mathcal{F}} \models \Box(\psi_1 \rightarrow \Diamond\psi_7)$. That corresponds to the property of communal accessibility on the concrete level.

8.1.3 Local Abstraction

The remaining properties are universal properties. To prove them we have to focus on individual processes. For the property of individual accessibility we choose an abstraction focusing on one process p:

$$\begin{array}{rcl}
p_0 & \equiv & p \in \mathrm{At_}\ell_0 \\
p_1 & \equiv & p \in \mathrm{At_}\ell_1 \\
e^p_{12} & \equiv & p \in \mathrm{At_}\ell_1 \wedge \forall_P\, p : p \in \mathrm{At_}\ell_0 \cup \mathrm{At_}\ell_1 \cup \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_4 \\
p_2 & \equiv & p \in \mathrm{At_}\ell_2 \\
p_3 & \equiv & p \in \mathrm{At_}\ell_3 \\
e^p_{34} & \equiv & p \in \mathrm{At_}\ell_3 \wedge \neg\, \exists_P\, p : p \in \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
e^p_{35} & \equiv & p \in \mathrm{At_}\ell_3 \wedge \exists_P\, p : p \in \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
p_4 & \equiv & p \in \mathrm{At_}\ell_4 \\
e^p_{45} & \equiv & p \in \mathrm{At_}\ell_4 \wedge \exists_P\, p : p \in \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
p_5 & \equiv & p \in \mathrm{At_}\ell_5 \\
e^p_{56} & \equiv & p \in \mathrm{At_}\ell_5 \wedge \neg\, \exists_P\, p : p \in \mathrm{At_}\ell_3 \cup \mathrm{At_}\ell_4 \\
p_6 & \equiv & p \in \mathrm{At_}\ell_6 \\
e^p_{67} & \equiv & p \in \mathrm{At_}\ell_6 \wedge \forall_P\, p < i : p \in \mathrm{At_}\ell_0 \cup \mathrm{At_}\ell_1 \cup \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_4 \\
p_7 & \equiv & p \in \mathrm{At_}\ell_7
\end{array}$$

Since e^p_{01}, e^p_{23}, and e^p_{71} coincide with p_0, p_2, and p_7, they are omitted. Moreover, the abstract system maintains *taken* variables $t_\tau \equiv \exists_P\, i : i \in T_\tau$ to keep track of which transition was taken.

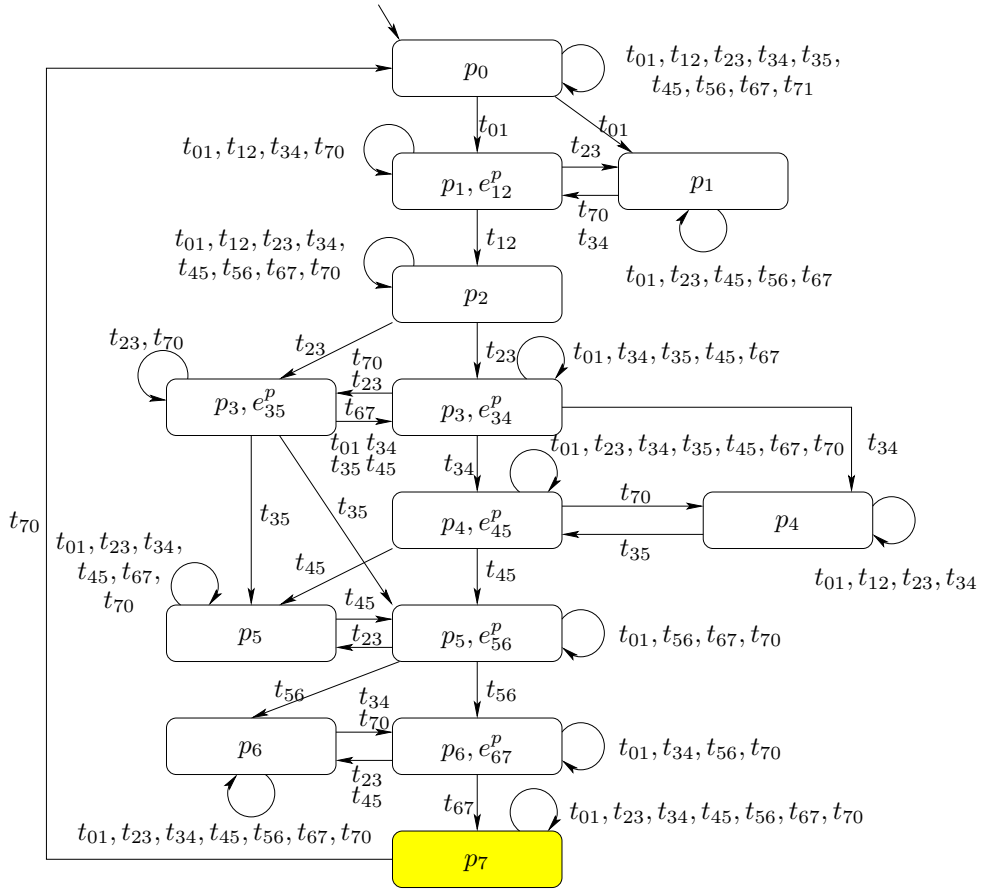


Figure 8.1: Abstract state graph

Figure 8.1 represents an abstract state graph. The states are labeled with those abstract variables which are *true* in that state. The *taken* variables label the edges of the graph and indicate which transition has been taken.

Even with the fairness condition $\mathcal{F}$ from above we cannot prove that

$$\mathcal{F} \rightarrow (\Box(p_1 \rightarrow \Diamond p_7))$$

holds. Model-checking detects two possible counterexamples. The first one lets process p stuck at ℓ_2. The loop at the corresponding node in Figure 8.1 is labeled with almost all *taken* variables such that the loop is not excluded by $\mathcal{F}$. The second is based on a cycle that switches e_{12}^p infinitely often while p is in ℓ_1. While the first problem can be solved by assuming weak fairness on all transitions, which is a rather natural assumption, the second one requires strong fairness. The real cause of the counterexamples is that the abstract system does not keep track of the enabledness of transitions in a global fashion. Otherwise it would be obvious that the guards exclude each other to a certain extend. Hence, refining the abstraction to

$$\begin{aligned}
p_0 &\equiv p \in \mathrm{At_}\ell_0 \\
p_1 &\equiv p \in \mathrm{At_}\ell_1 \\
e_{12}^p &\equiv p \in \mathrm{At_}\ell_1 \wedge \forall_P\, p : p \in \mathrm{At_}\ell_0 \cup \mathrm{At_}\ell_1 \cup \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_4 \\
p_2 &\equiv p \in \mathrm{At_}\ell_2 \\
p_3 &\equiv p \in \mathrm{At_}\ell_3 \\
e_{34} &\equiv \neg\, \exists_P\, p : p \in \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
e_{35} &\equiv \exists_P\, p : p \in \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
p_4 &\equiv p \in \mathrm{At_}\ell_4 \\
e_{45} &\equiv \exists_P\, p : p \in \mathrm{At_}\ell_5 \cup \mathrm{At_}\ell_6 \cup \mathrm{At_}\ell_7 \\
p_5 &\equiv p \in \mathrm{At_}\ell_5 \\
e_{56} &\equiv \neg\, \exists_P\, p : p \in \mathrm{At_}\ell_3 \cup \mathrm{At_}\ell_4 \\
p_6 &\equiv p \in \mathrm{At_}\ell_6 \\
e_{67}^p &\equiv p \in \mathrm{At_}\ell_6 \wedge \forall_P\, p < i : p \in \mathrm{At_}\ell_0 \cup \mathrm{At_}\ell_1 \cup \mathrm{At_}\ell_2 \cup \mathrm{At_}\ell_4 \\
p_7 &\equiv p \in \mathrm{At_}\ell_7
\end{aligned}$$

strengthens the abstraction such that individual accessibility can be proved:

$$\mathcal{S}_A^{\mathcal{F}} \models \Box(p_1 \rightarrow \Diamond p_7) \ .$$

The fact that we do not need any fairness requirements at the concrete level is due to the special functionality of Szymanski's algorithm. Processes that enter the waiting room block the progress of processes already in the waiting

room. If processes start to leave the waiting room, the entrance door to the waiting room is closed. Hence, processes are forced to finish the protocol once they requested to enter their critical section.

Local Abstraction with two processes Last but not least we want to prove linear waiting. Therefore, we need two processes. Thus, our abstraction focuses on two arbitrary processes $p < q$ with abstract variables $p_0, q_0, p_1, q_1, p_2, q_2, \ldots$ expressing the actual location of the processes.

Additional we fix the order between p and q:

$$\mathit{order} \equiv p < q \ .$$

As context predicate to enforce progress for the two processes we observe the locations ℓ_5, ℓ_6, and ℓ_7:

$$\mathit{critical} \equiv \mathrm{At}\_\ell_5 \cup \mathrm{At}\_\ell_\cup \mathrm{At}\_\ell_7 \neq \emptyset \ .$$

Together with the fairness condition $\mathcal{F}$ we can prove three of our four properties to hold for the abstract system:

Safety: $\quad \Box(\neg(p_7 \wedge q_7))$

Individual accessibility: $\quad \mathcal{F} \rightarrow \Box(p_1 \rightarrow \Diamond p_7)$

Competition: If two processes compete, the one with the smaller ID wins:

$$\mathcal{F} \rightarrow \Box(q_1 \wedge p_1 \rightarrow ((\neg q_7)\, \mathcal{U}\, p_7))$$

holds, but

$$\mathcal{F} \rightarrow \Box(q_1 \wedge p_1 \rightarrow ((\neg p_7)\, \mathcal{U}\, q_7))$$

does not hold for the abstract system.

In general, we have linear waiting:

$$\mathcal{F} \rightarrow \Box(q_1 \wedge p_1 \rightarrow ((((\neg p_7)\, \mathcal{U}\, p_7)\, \mathcal{U}\, (\neg p_7))\, \mathcal{U}\, q_7))$$

So, using different abstractions and augmenting the abstract system with extra fairness conditions we are able to verify Szymanski's mutual exclusion algorithm concerning both kinds of properties, safety and liveness.

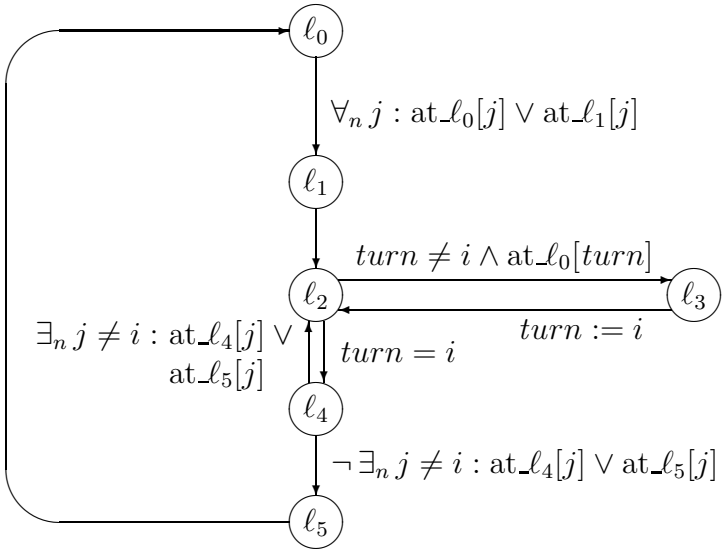


Figure 8.2: Dijkstra's mutual exclusion algorithm

8.2 Dijkstra's Mutual Exclusion Algorithm

Consider the following version of a mutual exclusion algorithm inspired by Dijkstra's algorithm [Dij65], where each process $\mathcal{S}(i, n)$ is given in Figure 8.2. In this description, we have a single global variable *turn* that ranges over natural numbers. The idea of the algorithm is as follows:

> Initially the *turn* has an arbitrary value and all processes start in ℓ_0. Processes that want to enter their critical section proceed to ℓ_1. The first one proceeding to ℓ_2 blocks new processes from entering ℓ_1. If the process denoted by *turn* resides at ℓ_0, the processes at ℓ_2 try to get the *turn* via ℓ_3. A process having the *turn* proceeds to ℓ_4. If it is the only one at ℓ_4 or ℓ_5, it is allowed to enter the critical section. Otherwise it has to return to ℓ_2. Eventually, all processes gather at ℓ_2, only the process that finally has the *turn* is then allowed to enter the critical section.

In fact, the algorithm is not the original algorithm from Dijkstra since Dijkstra's algorithm does not satisfy the individual accessibility property, which we like to prove on our algorithm. To achieve individual accessibility we block processes in ℓ_0 if the 'waiting room' ℓ_2, ℓ_3, ℓ_4 is non-empty. This causes an interesting behavior in the waiting room. And this is what we are aiming for in this example: to show how to deal with different kinds of fairness requirements.

Individual accessibility bases on several fairness requirements; all transitions are weak fair and the transitions leaving ℓ_4 are strong fair. I.e., for the MPS $\mathcal{S}$ illustrated in Figure 8.2 we have $\mathcal{J} = \{\tau_{01}, \tau_{12}, \tau_{23}, \tau_{32}, \tau_{24}, \tau_{42}, \tau_{45}, \tau_{50}\}$ and $\mathcal{C} = \{\tau_{42}, \tau_{45}\}$.

Obviously, the algorithm is an MPS with a global variable (see Definition 2.8). When the algorithm is translated into a WS1S system according to Definition 3.7 the *turn* variable is translated identically. Note that the control flow is again modeled with boolean variables $\text{at\_}\ell_i$. In the translation to WS1S systems this introduces six set variables $\text{At\_}\ell_0, \ldots, \text{At\_}\ell_5$. The only non-boolean variable *turn* ranging over the domain of involved process indices may be read and written by all processes.

To verify liveness properties we have to express fairness also for the WS1S system (see Definition 7.1). Therefore, we have to adapt Definition 7.2. The definition introduces set variables E_τ, T_τ for all fair transitions τ to keep track of enabledness and application of transitions for each process.

For the local abstraction α, we monitor the behavior of an arbitrary process p and of the current *turn* process, i.e., the process having the turn. Moreover, the variable ξ expresses mutual exclusion and some others are used to express enabledness of transitions:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{turn}_i & \equiv & \mathit{turn} \in \text{At\_}\ell_i \quad \text{for } 0 \le i \le 5 & \xi & \equiv & \exists_P k : \text{At\_}\ell_5 \subseteq \{k\} \\ p_i & \equiv & p \in \text{At\_}\ell_i \quad \text{for } 0 \le i \le 5 & \gamma & \equiv & p = \mathit{turn} \\ \psi_5 & \equiv & \text{At\_}\ell_5 \neq \emptyset & \delta & \equiv & \text{At\_}\ell_4 \setminus \{\mathit{turn}\} \neq \emptyset \end{array}$$

Here we leave out the abstract variables monitoring whether p or *turn* takes a transition, but they are part of the abstract system.

Verification shows that we need weak fairness for p at ℓ_1. e^p_{12} is equivalent to $p \in \text{At\_}\ell_1$, and $\Box\Diamond t^p_{12}$ is equivalent to $\Box\Diamond(p \in \text{At\_}\ell_2)$. Hence, we can lift weak fairness for p according to Lemma 7.6 and express it as:

$$\Diamond\Box p_1 \rightarrow \Box\Diamond p_2 \ .$$

Moreover, we need that every process eventually leaves the critical section. Unfortunately, we cannot lift weak fairness for all processes to our abstract system. As already discussed processes may take turns. Fortunately, we need weak fairness at ℓ_5, as we expect mutual exclusion to hold, and we can conclude $\exists_P q : \Diamond\Box\text{At\_}\ell_5$ from $\Diamond\Box\psi_5$. Hence, in this special case we can lift weak fairness:

$$\Diamond\Box\psi_5 \rightarrow \Box\Diamond t_{50} \ .$$

Strong fairness for τ_{42} and τ_{45} can be lifted to the abstract level according to Lemma 7.5 and be expressed as

$$\Diamond\Box\delta \rightarrow \Box\Diamond(t^o_{42} \vee t^o_{45}) \ ,$$

where t^o_{42} and t^o_{45} are defined as $\exists_P i : i \neq turn \wedge i \in T_{42}$, respectively, $\exists_P i : i \neq turn \wedge i \in T_{45}$.

Now, we generate some fairness conditions using the marking algorithm in order to guarantee general progress. For the simple ranking predicates $\chi_0 \equiv i \in Atl_0$, $\chi_1 \equiv i \in Atl_1$, $\chi_2 \equiv i \in Atl_2$, $\chi_3 \equiv i \in Atl_3$, and $\chi_4 \equiv i \in Atl_4$ we can strengthen $\mathcal{F}$ with:

$$\begin{array}{l}(\Box\Diamond t_{01} \rightarrow \Box\Diamond t_{50}) \wedge \\ (\Box\Diamond t_{12} \rightarrow \Box\Diamond t_{01}) \wedge \\ (\Box\Diamond t_{23} \rightarrow \Box\Diamond(t_{32} \vee t_{42} \vee t_{12})) \wedge \\ (\Box\Diamond t_{32} \rightarrow \Box\Diamond t_{23}) \wedge \\ (\Box\Diamond t_{42} \rightarrow \Box\Diamond t_{24}) \ .\end{array}$$

Some advanced fairness requirements controlling the waiting room ℓ_2, ℓ_3, and ℓ_4 defined as $\chi_w \equiv \mathrm{At}\_\ell_2 \cup \mathrm{At}\_\ell_3 \cup \mathrm{At}\_\ell_4$ and $\chi_e \equiv (\mathrm{At}\_\ell_4 \setminus \{turn\}) \cup (\mathrm{At}\_\ell_2 \cap \{turn\})$ give us:

$$(\Box\Diamond t_{45} \rightarrow \Box\Diamond t_{12})$$

and

$$(\Box\Diamond t^o_{42} \rightarrow \Box\Diamond(t_{32} \vee t_{12})) \ .$$

Then, using NuSMV we are able to prove the following LTL formula:

$$\Box\xi \wedge (\mathcal{F} \rightarrow \Box(p_1 \rightarrow \Diamond p_5)) \ .$$

When checking separately, the mutual exclusion property, i.e., the formula $\Box\xi$, only takes 2 sec, whereas checking the liveness property takes 263 sec. We discuss the various run-time results at the end of this chapter.

8.3 Token Rings

The previous sections present intricate mutual exclusion algorithms. A lot of complications arise as those algorithms try to coordinate only those processes that actually request access to the critical section. Another approach is to grant access in a *round robin* manner. Processes that do not need access simply give back their access right. Token rings implement such an approach by passing a token in a ring structure. A process having the token is eligible to perform special actions, entering its critical section, for example.

When the token does not transport any value, the verification of such token rings is decidable. [EN95] shows that for universal properties concerning 1, 2, or 3 processes, it is sufficient to check for ring networks of size 3, 4,

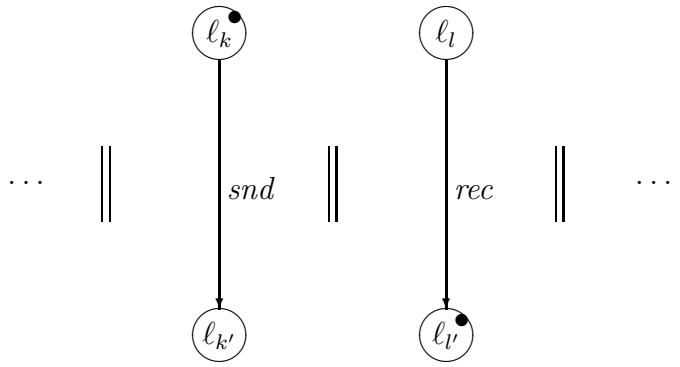


Figure 8.3: Synchronized token passing

respectively, 5 in order to establish CTL\X properties. Here, it is natural to abstain from the next operator as the time needed for the token to go around the ring varies with the ring size.

The systems handled in [EN95] are quite simple: all actions of a process are local, they depend on its local state and change its local state. Only the global property whether a process has the token or not can be checked. The token is modeled by a special boolean array variable $tok[0..n-1]$ that is *true* at the index of the unique process having the token. The token is passed by a synchronized action between a process and its right (increasing process index) neighbor. In Figure 8.3 we have indicated the token as a bullet $\bullet$.

Formally, to model such systems as MPS, we specify such a step as:

$$\begin{aligned}\exists_n\, i : {} & \mathrm{at}\_\ell_k[i] \wedge \mathrm{at}\_\ell_l[i \oplus_n 1] \wedge \mathrm{at}\_\ell'_{k'}[i] \wedge \mathrm{at}\_\ell'_{l'}[i \oplus_n 1] \wedge \\ & tok[i] \wedge \neg tok'[i] \wedge tok'[i \oplus_n 1] \wedge \\ & \textstyle\bigwedge_{l \in \mathcal{L}\setminus\{k'\}} \neg \mathrm{at}\_\ell'_l[i] \wedge \bigwedge_{k \in \mathcal{L}\setminus\{l'\}} \neg \mathrm{at}\_\ell'_k[i \oplus_n 1] \wedge \\ & \textstyle\bigwedge_{j<n, j\neq i, i\oplus_n 1} tok'[j] \leftrightarrow tok[j] \wedge \bigwedge_{l\in\mathcal{L}} \mathrm{at}\_\ell'_l[j] \leftrightarrow \mathrm{at}\_\ell_l[j] \ .\end{aligned}$$

All other steps are executed asynchronously and alter the state space of a single process (w.l.o.g. we assume $\Sigma = \mathcal{L}$). A local transition from l to l' may depend on having the token or not:

$$\begin{aligned}\exists_n\, i : {} & \mathrm{at}\_\ell_l[i] \wedge ((\neg) tok[i]) \wedge \mathrm{at}\_\ell'_{l'}[i] \wedge \\ & tok'[i] \leftrightarrow tok[i] \wedge \textstyle\bigwedge_{k \in \mathcal{L}\setminus\{l'\}} \neg \mathrm{at}\_\ell'_k[i] \wedge \\ & \textstyle\bigwedge_{j<n, j\neq i} tok'[j] \leftrightarrow tok[j] \wedge \bigwedge_{l\in\mathcal{L}} \mathrm{at}\_\ell'_l[j] \leftrightarrow \mathrm{at}\_\ell_l[j] \ .\end{aligned}$$

To illustrate the verification of a token ring consider the example from [WL89] shown in Figure 8.4. Initially, all processes except the one having the token start at ℓ_0. The process possessing the token starts at ℓ_4.

$$\begin{aligned}\Theta \equiv \exists_n\, i : {} & tok[i] \wedge \mathrm{at}\_\ell_4[i] \wedge \textstyle\bigwedge_{l<4} \neg \mathrm{at}\_\ell_l[i] \wedge \\ & \textstyle\bigwedge_{i\neq j<n} \neg tok[j] \wedge \mathrm{at}\_\ell_0[j] \wedge \bigwedge_{0<l<5} \neg \mathrm{at}\_\ell_l[j]\end{aligned}$$

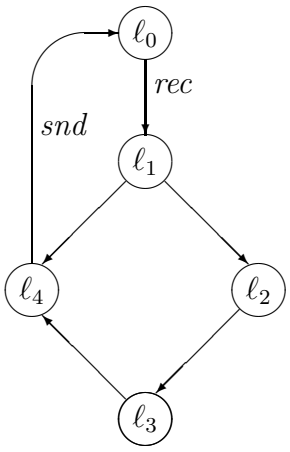


Figure 8.4: Mutual exclusion in token rings

A process that receives the token proceeds to location ℓ_1. There it has to decide whether to enter its critical section at ℓ_3 or to pass the token to its neighbor immediately.

To verify safety and liveness properties of the algorithm we translate it into a WS1S system. The translation is straightforward using our translation function *Tr*. When choosing an abstraction in order to verify token rings, we have to express that the token is unique and that it cycles around forever. Therefore, we add an abstract variable expressing this uniqueness

$$\mathit{unique} \equiv \exists_P i : \{i\} = \mathit{Tok} \ .$$

To characterize that every process possesses the token infinitely often, we would like to add fairness conditions. Therefore, we have to know that the process with the token always passes it to its neighbor and that its neighbor is always willing to receive it. The second property can be expressed as

$$\mathit{ready} \equiv \mathit{Tok} = \mathrm{At}\_\ell_1 \cup \mathrm{At}\_\ell_2 \cup \mathrm{At}\_\ell_3 \cup \mathrm{At}\_\ell_4 \ ,$$

i.e., all processes without token are in ℓ_0 and hence ready to receive the token.

With an abstraction focusing on a single process p, i.e., $p_l \equiv p \in \mathrm{At}\_\ell$ for $0 \leq l \leq 4$, we can establish:

$$\forall_P p : \Box(p \in \mathrm{At}\_\ell_1 \rightarrow \Diamond \mathrm{At}\_\ell_4) \ .$$

Hence, for further verification we can safely add:

$$\forall_P p : \Box\Diamond p \in \mathit{Tok} \ .$$

According to the properties we verify, i.e., $\forall_n i : \varphi(i)$, $\forall_n i : \varphi(i, i \oplus_n 1)$, or $\forall_n i, j, j \neq i : \varphi(i, i \oplus_n 1, j)$, we add such a fairness constraint for each process that is part of the abstraction.

In our example we can easily verify the safety property

$$\forall_n p, q, p \neq q : \Box\neg(p \in \text{At\_}\ell_4 \wedge q \in \text{At\_}\ell_4) \ ,$$

which can also be derived from *unique* and *ready* that are invariant in the abstract system $\mathcal{S}_A$.

With the fairness constraints we can also state that each process can enter its critical section whenever it wants to:

$$\forall_P p : \Box\Diamond(p \in \text{At\_}\ell_2 \vee p \in \text{At\_}\ell_4) \ .$$

The example presents a general strategy how to tackle the verification of token rings and how to model the ring topology. This basic idea also appears when verifying time-triggered systems.

8.4 Run-time Results

For Dijkstra's mutual exclusion algorithm we have already seen that NuSMV needs quite a long computation time to prove the liveness property whereas safety is established within a second. This is caused by the need for fairness conditions. Model-checking with strong fairness is costly regarding computation time and memory space. LTL model-checking is an exponential complexity in the size of the checked formula [CGP99]. Since we have to state the fairness of the abstract system as premise to the desired property, checking of liveness properties gets rather expensive.

Before model-checking the abstract system, we have to construct it first. There the computation time and the memory space needed mainly depends on the chosen type of abstraction, local or global, and on the number of defined abstract variables. Figure 8.5 shows computation time and memory space needed for the different examples in the abstraction step as well as in the model-checking step. We obtained the results on a Sun Sparc 9 with 750 MHz using NuSMV 1.1 and MONA 1.4. For notational brevity we use the following abbreviations:

MA Marking Algorithm: The identified ranking predicates are used to label the transitions. Then, we can use the *taken* variables to express the generated fairness conditions.

GA Global Abstraction: Constructing the abstract system for a globally defined abstraction relation.

Algorithm	Type	PAX Time	Phase 1 Memory	Prop.	NuSMV Time	Phase 2 Memory
Szymanski	MA	8 s	1 MB	—	—	—
	GA	10:10 min	350 MB	ME	1 s	8 MB
		—	—	CA	4 s	16 MB
	LA1	10 s	5 MB	IA	33 s	16 MB
	LA2	3:42 min	518 MB	ME	3 s	12 MB
		—	—	IA	1:04 min	20 MB
		—	—	LW	34 s	22 MB
Dijkstra	MA	4:47 min	337 MB	—	—	—
	LA2	3:03 min	491 MB	ME	2 s	10 MB
		—	—	IA	4:16 min	26 MB
Token ring	LA2	2:51 min	327 MB	ME	1 s	8 MB
		—	—	IA	1 s	8 MB

Figure 8.5: Used computation time and memory space

LA1 Local Abstraction focusing on 1 process: local abstraction which is mainly defined to focus on a single process.

LA2 Local Abstraction focusing on 2 processes: local abstraction which is mainly defined to focus on two different processes.

For the verified properties we use the following abbreviations:

ME Mutual Exclusion

CA Communal Accessibility

IA Individual Accessibility

LW Linear Waiting

In a local abstraction we know: if $p_i \equiv p \in \text{At\_}\ell_i$ holds, all other p_j are necessarily *false*. In a global abstraction the variables $\psi_i \equiv \text{At\_}\ell_i \neq \emptyset$ are quite independent. Therefore, the memory requirements grow as the automaton recognizing the satisfying evaluations is mainly the cross product of those automata that decide for a single variable. As a result the local abstraction focusing on two processes needs less time and quite the same memory space although it is defined by far more abstract variables.

The same argumentation holds for the marking algorithm applied to Szymanski's and Dijkstra's algorithm. In case of Dijkstra's algorithm we labeled all transitions simultaneously concerning the ranking predicates $\chi_i \equiv i \in$

At_ℓ_k, $0 \leq k \leq 5$. This is quite expensive since they are maximal independent; to decide whether the transition decreases or increases a certain set the other sets are of no interest. Hence, the variables $+_{\chi_i}, -_{\chi_i}$ are totally unrelated from $+_{\chi_j}, -_{\chi_j}$, for $i \neq j$.

For Szymanski's algorithm is was impossible to compute a labeling concerning all 8 ranking predicates at once. Hence, we computed the labeling separately for each predicate which is quite fast and needs not much memory.

Chapter 9

Completeness Results

The first parts of this thesis explain how to model different classes of parameterized systems as WS1S systems and present an abstraction-based approach to their verification. This chapter illustrates that the framework presented is indeed a unifying approach for networks consisting of finite state processes. It does so by classifying systems presented in [EN96, EK00] as restricted MPS. Hence, we can represent these parameterized systems as WS1S systems.

We give generic abstractions for these systems that allow to compute a finite abstract system which can be subject to model-checking techniques. The abstract systems are "good" enough to check for invariance properties. For liveness properties we introduce ranking predicates to generate fairness conditions which allow to exclude cycles during model-checking that have no concrete counter-parts. We prove our approach to be strong enough to allow no false negatives. Hence, we can analyze the mentioned classes of parameterized systems automatically using model-checking techniques.

9.1 Restricted Synchronous Systems

In [EN96] Emerson and Namjoshi consider parameterized synchronous systems, where an instance $\mathcal{P}(m)$ of size m is a parallel composition of a control process C with m copies of the user process U, i.e., $\mathcal{P}(m) = C \| U_0 \| \ldots \| U_{m-1}$ where U_i is obtained from U by subscribing the states of U with i. Transitions in C and U are labeled with guards where the states of the processes serve as atomic propositions. User processes have no identifiers to distinguish from each other. Guards are only allowed to refer to the control process or to the existence of user processes at certain states.

As an example [EN96] presents the synchronous network that we consider in Example 2.14. We already classified the example as a synchronous MPS.

Hence, we can model the system as a WS1S system as done in Example 3.11.

To capture formally which guards are allowed we define a restricted version of AF^+.

Definition 9.1 (Restricted version of $\mathbf{AF}^+$)
Define AF_r^+ as:

$$f ::= a = const \mid b[i] \mid \neg f \mid f \vee f \mid \exists_n x : b[x] \ ,$$

where x is a *position variable* and a is *global variable* and b is a *boolean array variable.* □

The global variable a is meant to model the control process. Each user process i can check and set its own state $b[i]$, but can only check for the existence of other processes in certain states, it is not possible to address a special process.

For the remainder of the section let us fix a synchronous parameterized system $\mathcal{P} = \langle(\Sigma_C, \mathcal{T}_C, \mathcal{I}_C), (\Sigma_U, \mathcal{T}_U, \mathcal{I}_U)\rangle$ that is defined as an MPS using the restricted logic AF_r^+ and its WS1S translation $\mathcal{W} = (\mathcal{V}, \Theta, \mathcal{T})$.

9.1.1 Abstraction of Synchronous Systems

An abstraction is always chosen in order to prove certain properties one is interested in. Hence, we have to know what kind of properties we would like to establish. The synchronous systems defined above only allow the processes a restricted view on the actual configuration of the whole network. It is not possible to react on the behavior of a certain process except for the control process. The guards only check the existence of user processes at certain states. Hence, having a system's computation the rôles of the user processes are interchangeable. This is formalized in the following lemma.

Lemma 9.2 (Symmetry of user processes)
For every permutation π over $\{1, \ldots, n\}$ let f_π define a auto-bijection over $\mathcal{P}(n)$ such that

$$f_\pi : \Sigma_C \times \Sigma_U^n \longrightarrow \Sigma_C \times \Sigma_U^n, (c, u_1, \ldots, u_n) \mapsto (c, u_{\pi(1)}, \ldots, u_{\pi(n)}) \ .$$

Then, f_π is a bisimulation.

Proof It is obvious that for an initial state $s_0 \in \mathcal{I}_C \times \mathcal{I}_U^n$ also $f_\pi(s_0$ is in $\mathcal{I}_C \times \mathcal{I}_U^n$. Having s' as a successor of state s in $\mathcal{P}(n)$ we can use the transition taken by user process i $u_i \xrightarrow{g_i} u_i'$ as well for user process $\pi(i)$ and vice versa since g_i and $g_{\pi(i)}$ only allow to check for existence of user processes at certain states. ■

Since the user processes behave in a total symmetric manner the interesting properties to establish are of the universal form $\forall p, q, p \neq q : \varphi(p, q)$, where φ is an LTL formula with atomic propositions over control process states and over states of user processes p and q. I.e., every two distinct processes fulfill φ (for the sake of presentation we restrict ourselves to properties about 2 user processes). Note, when stating such properties about the original system $\mathcal{P}$, we have $c \in \Sigma_C, u \in \Sigma_{U_i}$ as propositions. For the WS1S system $\mathcal{W}$ we have c and $i \in X_u$ as propositions.

Having such properties in mind we give an abstraction function into a finite abstract domain that — which we show later — suffices to verify those properties defined above.

Definition 9.3 (Abstract system $\mathcal{S}_A^{\mathcal{F}}$)
Define an abstract system $\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ of $\mathcal{W}$ with $\mathcal{V}_A$ containing a boolean variable c for every $c \in \Sigma_C$ (and hence $c \in \mathcal{V}$) and a boolean variable b_u for every $u \in \Sigma_U$ ($X_u \in \mathcal{V}$). Let $\mathcal{S}_A$ be defined by the abstraction function α with $\alpha(s)(c) \leftrightarrow c$, i.e., $c \equiv c$, and $\alpha(s)(u) \leftrightarrow \exists i : i \in X_u$, i.e., $b_u \equiv \exists i : i \in X_u$.

Moreover, if fairness conditions for ranking predicates χ are added, then define $\mathcal{S}_A^{\mathcal{F}}$ as the extended fair abstract system according to Section 7.3. □

9.1.2 Proving Universal Properties

We construct the abstract system above in order to establish some universal properties of the form $\forall p, q, p \neq q : \psi(p, q)$ for the WS1S system $\mathcal{W}$ and hence for all instances of the original synchronous system $\mathcal{P}$. Previously, we suggested to use local abstractions to prove such universal properties. In this setting with an extra control process we present a global abstraction since Lemma 9.2 always allows to verify a global property φ instead of $\forall p, q, p \neq q : \psi(p, q)$ by considering the system $(C \oplus U \oplus U) \parallel U^n$. Then, the global abstraction of Definition 9.3 also focuses on two arbitrary processes. Since we have the control states also as propositions in the abstract system, we can equivalently state the universal property φ for the abstract system as a predicate φ_A.

We have chosen this solution for simplicity in the following proofs, Corollary 9.11 shows that one could also extend the abstract system by local abstractions $p_u \equiv p \in X_u, q_u \equiv q \in X_u$ to achieve the same result.

Suppose for the remainder that we have an abstract system as in Definition 9.3. It would be a nice result to show in general that $\mathcal{W} \models \varphi$ if and only if $\mathcal{S}_A^{\mathcal{F}} \models \varphi_A$.

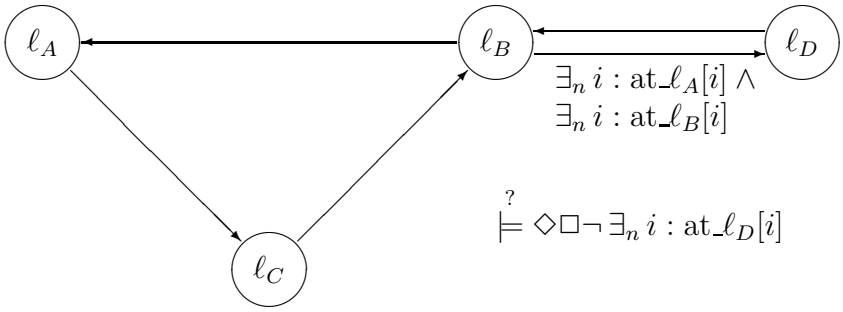


Figure 9.1: Process template for user processes

For safety properties φ this follows immediately from the following lemma stating that each abstract finite path has a concretization.

Lemma 9.4 (Concretization of finite paths)
For every finite path $\sigma : \sigma_0, \ldots$ in the abstract system $\mathcal{S}_A$ (Definition 9.3) we can find a concrete path $\widetilde{\sigma}$ in $\mathcal{W}$ such that $\sigma_i = \alpha(\widetilde{\sigma}_i)$.

Proof Initially, we need at least as much processes as $k = |\{b_u \mid b_u \in \mathcal{V}_A, \quad \sigma_0 \models b_u\}|$ since that corresponds to the existence of elements in X_u. But to cover all possible runs of a process we start with $k \cdot |\mathcal{V}_A|^{l'}$ processes, i.e., with $|\mathcal{V}_A|^{l'}$ processes in each initial user state. Then, we can find a state $\widetilde{\sigma}_1$ with $\alpha(\widetilde{\sigma}_1) = \sigma_1$ such that each $X_u \in \{X_u \mid, X_u \in \mathcal{V}, b_u \in \mathcal{V}_A, \quad \sigma_1 \models b_u\}$ contains at least $|\mathcal{V}_A|^{l'-1}$ elements. Moreover, we can distribute the $k \cdot |\mathcal{V}_A|^{l'}$ processes in such a way that $(\widetilde{\sigma}_0, \widetilde{\sigma}_1) \models \tau \wedge \widehat{\alpha}$ holds. By induction, we can find a concrete sequence $\widetilde{\sigma}_0, \ldots, \widetilde{\sigma}_{l'+1}$ such that $(\widetilde{\sigma}_i, \widetilde{\sigma}_{i+1}) \models \tau \wedge \widehat{\alpha}$ for every $i < l'$. ∎

We have proven so far that for every finite abstract path we can find a concretization. Moreover, we can quite freely alter the number of processes at a certain state.

Unfortunately, the guidelines to construct fairness conditions fail to exclude unfair cycles as the following example illustrates.

Example 9.5 (Counterexample (synchronous systems))
Consider a synchronous parameterized system $\mathcal{P} = \{U_0 \| \ldots \| U_{m-1} \mid m > 1\}$ (we need no control process at all). The user processes are described in Figure 9.1. Assume the processes to start initially in the locations ℓ_A or ℓ_B (it is easy to modify the example to have a unique initial state).

We are interested in whether it is possible or not to enter control point ℓ_D infinitely often. Intuitively, the processes starting in ℓ_A cycle around ℓ_A,

ℓ_C, ℓ_B, and ℓ_A again. Processes starting in ℓ_B have the choice to cycle ℓ_B, ℓ_A, ℓ_C, ℓ_B. Or, they go to ℓ_D and back to ℓ_B joining there the processes started in ℓ_A.

Hence, either at some point processes in ℓ_B decide not to go to ℓ_D although this may be possible, or, otherwise a process going to ℓ_D joins the ℓ_A-processes. Because we do have only a finite amount of processes, control point ℓ_D is not reached infinitely often.

Hence, we would like to prove $\Diamond\Box\neg\,\exists_n\, i : \text{at\_}\ell_D$.

In the abstract system $\mathcal{S}_A^{\mathcal{F}}$ according to Definition 9.3 we have the abstract variables b_A, b_B, b_C, b_D. The abstract variables are evaluated to *true* whenever a process resides in the corresponding location. Our property of interest is there characterized by $\Diamond\Box\neg b_D$.

The abstract cycle

$$\langle (b_A, b_B, \neg b_C, \neg b_D), (b_A, \neg b_B, b_C, b_D), (\neg b_A, b_B, b_C, \neg b_D), (b_A, b_B, \neg b_C, \neg b_D)\rangle$$

is not excluded by any of the fairness conditions that are proposed in Section 7.3.2. Every set and every union of sets decrease but also increase during that cycle. □

To overcome this problem we introduce *threaded graphs* as proposed by Emerson and Namjoshi in [EN96].

Definition 9.6 (Threaded graph)
Let $\mathcal{S}_A$ be an abstract system $(\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$ of $\mathcal{P}$. Having an abstract cycle $\langle \sigma_1, \ldots, \sigma_n\rangle$ where $\sigma_1 = \sigma_n$ in $\mathcal{S}_A$ we define a *threaded graph* $\mathcal{G} = (\Sigma_U \times [1..n], E)$ where $((u_1, k), (u_2, l)) \in E$ iff $k + 1 = l$, $\sigma_k \models b_{u_1}$, $\sigma_l \models b_{u_2}$ and the transition $u_1 \xrightarrow{g} u_2$ is in $\mathcal{T}_U$ and $\sigma_k \models g$ or $u_1 = u_2$ and $k = n, l = 1$. □

The intuition behind the definition above is to follow the paths of the processes. Then, an analysis of the graph allows us to check whether there is also an infinite path (cycle) in the concrete system.

Example 9.7 (Threaded graph)
For the abstract cycle

$$\langle (b_A, b_B, \neg b_C, \neg b_D), (b_A, \neg b_B, b_C, b_D), (\neg b_A, b_B, b_C, \neg b_D), (b_A, b_B, \neg b_C, \neg b_D)\rangle$$

the threaded graph is shown in Figure 9.2. □

For an abstract counterexample we have to check whether all SCC's of the corresponding threaded graph behave fair, i.e., they do not lose processes continuously.

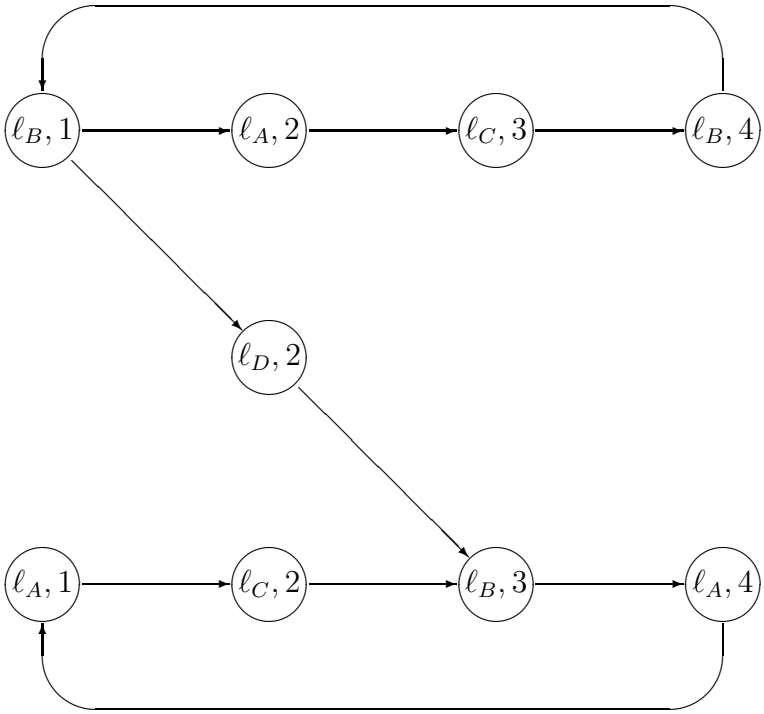


Figure 9.2: Threaded graph for σ

Example 9.8 (Ranking predicates on cycles)
The drain of processes being able to reach location ℓ_D in Example 9.5 gets obvious when using threaded graphs.

Obviously, the upper cycle in Figure 9.2 loses processes whenever a process decides to visit ℓ_D. In order to catch this behavior with some fairness requirements we have to characterize the processes in the upper cycle and use that characterization as a ranking predicate χ. Therefore we define χ as follows:

$$\begin{aligned} &i \in \mathrm{At}\_\ell_B \wedge \mathrm{At}\_\ell_A \neq \emptyset \wedge \mathrm{At}\_\ell_B \neq \emptyset \wedge \mathrm{At}\_\ell_C = \emptyset \wedge \mathrm{At}\_\ell_D = \emptyset \vee \\ &i \in \mathrm{At}\_\ell_A \wedge \mathrm{At}\_\ell_A \neq \emptyset \wedge \mathrm{At}\_\ell_B = \emptyset \wedge \mathrm{At}\_\ell_C \neq \emptyset \vee \\ &i \in \mathrm{At}\_\ell_C \wedge \mathrm{At}\_\ell_A = \emptyset \wedge \mathrm{At}\_\ell_B \neq \emptyset \wedge \mathrm{At}\_\ell_C \neq \emptyset \wedge \mathrm{At}\_\ell_D = \emptyset \end{aligned}$$

A boolean variable $-_\chi$ associated with χ indicates drain of the upper cycle, hence the fairness condition requiring $\Box\Diamond -_\chi \rightarrow \Box\Diamond +_\chi$ excludes the abstract counter example and allows to establish the desired property. □

If we are able to express unfairness of SCCs via ranking predicates as fairness conditions, we could establish

$$\mathcal{W} \models \varphi \quad \text{iff} \quad \mathcal{S}_A^{\mathcal{F}} \models \varphi_A$$

for an abstract system with the adequate fairness requirements.

9.1.3 Completeness Result

The example shows how to characterize the drain of processes in an abstract cycle that has no concrete counterpart. The example raises two questions. First, is the existence of connected SCCs in a threaded graph of an abstract cycle equivalent to the existence of a ranking predicate marking the abstract cycle as unfair, and, second, can we concretize the abstract cycle whenever the SCCs are unconnected.

The answers are given by the following lemma.

Lemma 9.9 (Ranking predicates on cycles)
Let $\mathcal{G}$ be the threaded graph for an abstract cycle $\sigma = \langle \sigma_1, \ldots, \sigma_n, \sigma_1 \rangle$ in $[\![\mathcal{S}_A]\!]$. Then, the three statements

1. *the strongly connected components (SCC) of $\mathcal{G}$ $sccs(\mathcal{G})$ are connected and for the connecting edge $e = ((u_k, i), (u_l, i+1))$ $(\sigma_i, \sigma_{i+1}) \models \exists_n i : i \in X_{u_k} \wedge i \in X_{u_l} \wedge \rho(\mathcal{V}, \mathcal{V}')$ holds,*
2. *there exists a ranking predicate χ such that for $\mathcal{S}_A^F$ augmented by the abstract variables $+_\chi, -_\chi$ σ is unfair, and*
3. *there exists no concrete path $\widetilde{\sigma} \in [\![\mathcal{W}]\!]$ such that $\alpha(\widetilde{\sigma}) = \sigma$ (α applied pointwisely)*

are equivalent.

Proof **(1) $\Longrightarrow$ (2):** Suppose $sccs(\mathcal{G})$ are connected. It is a known fact that the graph of *sccs* is acyclic. Choose an SCC $\mathcal{T}$ with only outgoing edges. This is the component to lose processes. Define a ranking predicate

$$\begin{aligned} \chi_\sigma \quad \equiv \quad & i \in \bigcup \{X_u \mid (X_u, 1) \in \mathcal{T}\} \wedge \sigma_1 \vee \\ & \cdots \\ & i \in \bigcup \{X_u \mid (X_u, n) \in \mathcal{T}\} \wedge \sigma_n \end{aligned}$$

Then, we consider the fair abstract system $\mathcal{S}_A^{\mathcal{F}}$ with $\mathcal{F} \equiv \Box\Diamond -_{\chi_\sigma} \rightarrow \Box\Diamond +_{\chi_\sigma}$. It remains to show $\sigma \notin [\![\mathcal{S}_A^{\mathcal{F}}]\!]$.

According to Definition 9.6 we have $(\sigma_i, \sigma_{i+1}) \models \rho(\mathcal{V}, \mathcal{V}') \rightarrow \chi(\mathcal{V}') \subseteq \chi(\mathcal{V})$ for all $i < n$, since $\mathcal{T}$ has no incoming edges. For the outgoing edge e we have by assumption $(\sigma_i, \sigma_{i+1}) \models \exists_n i : i \in X_{u_i} \wedge i \in X_{u_j} \wedge \rho(\mathcal{V}, \mathcal{V}')$, i.e., $(\sigma_i, \sigma_{i+1}) \models \rho(\mathcal{V}, \mathcal{V}') \rightarrow \chi(\mathcal{V}') \subset \chi(\mathcal{V})$ since $(u_k, i) \in \mathcal{T}$ and $(u_l, i+1) \notin \mathcal{T}$. Hence, the cycle σ sets $-_{\chi_\sigma}$ at least once to *true* but never $+_{\chi_\sigma}$ which excludes it from $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$.

(2) $\Longrightarrow$ (3): This direction follows directly from the definition of abstraction in Section 4.2 and Lemma 7.7.

(3) $\Longrightarrow$ (1): The proof can be found in [Nam98]. ∎

Now we are prepared to present the main theorem of this section.

Theorem 9.10 (Completeness for synchronous systems)
For every abstract system $\mathcal{S}_A$ (Definition 9.3) we can find additional ranking predicates χ_i such that for $\mathcal{S}_A^{\mathcal{F}}$ augmented by the abstract variables $+_{\chi_i}, -_{\chi_i}$

$$\mathcal{W} \models \varphi \quad \textit{iff} \quad \mathcal{S}_A^{\mathcal{F}} \models \varphi_A \ .$$

Proof $\Longleftarrow$: This direction follows directly from the definition of abstraction in Section 4.2 and Lemma 7.7.

$\Longrightarrow$: This implication states that $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ contains no false negatives. By contraposition we have to show that every counterexample in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ violating φ_A corresponds to a violation of φ by $\mathcal{W}$.

Let $\sigma : \sigma_0, \sigma_1, \ldots$ be in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ such that $\sigma \not\models \varphi_A$. Since φ_A is an LTL formula we can w.l.o.g. assume σ to have the form

$$\sigma_0, \ldots, \sigma_{l-1}, (\sigma_l, \ldots, \sigma_{l'})^{\omega} \ .$$

The literals in φ_A are of the form $c, \neg c, b_u, \neg b_u$ for $c, X_u \in \mathcal{V}$. Their occurrence corresponds to $c, \exists i : i \in X_u$ in φ. Let $\alpha^{-1}(\sigma)$ be the set of point-wise concretizations of σ, i.e., we take α as the original abstraction function without the fairness part. Then, for each $i < \omega$ $\alpha^{-1}(\sigma_i) \models \exists i : i \in X_u$ if and only if $\sigma_i \models b_u$. We have to prove the existence of a *consistent* trace in $\alpha^{-1}(\sigma) \cap [\![\mathcal{W}]\!]$. Hence, we have to show

$$\bigcap_{i<\omega} \{\sigma^{\mathcal{P}} \mid \sigma^{\mathcal{P}} \in \alpha^{-1}(\sigma), \quad (\sigma_i^{\mathcal{P}}, \sigma_{i+1}^{\mathcal{P}}) \models \tau \wedge \widetilde{\alpha}\} \neq \emptyset$$

where $\tau \in \mathcal{T}$ is the only transition in $\mathcal{W}$ (translation of synchronous systems).

According to Lemma 9.4 we can find concrete paths in $\mathcal{W}$ for every finite abstract path.

It remains to prove that we can extend the sequences above to infinity. Therefore, let $\mathcal{G}$ be the threaded graph of $\langle \sigma_l, \ldots, \sigma_{l'} \rangle$. Since, we could not generate any fairness condition to exclude the abstract cycle, the SCC's of $\mathcal{G}$ are unconnected according to Lemma 9.9. Then, according to Lemma 9.9 we can find a concretization $\langle \widetilde{\sigma}_1, \ldots, \widetilde{\sigma}_{(l'-l)k+1} \rangle$ of $\langle (\sigma_l, \ldots, \sigma_{l'})^k, \sigma_l \rangle$ for some k such that $\widetilde{\sigma}_1 = \widetilde{\sigma}_{(l'-l)k+1}$.

Hence, we have found a concrete path $\widetilde{\sigma}$ in $\mathcal{W}$ such that $\widetilde{\sigma} \not\models \varphi$ and therefore $\mathcal{W} \not\models \varphi$. ■

As it is always the question whether the choices made in a proof to show completeness of a certain method are useful in practice, here the answer is yes. When analyzing a concrete system by abstraction, the counterexamples generated by a model-checker lead us to finer abstractions or, as it is in this section, to missing fairness requirements to exclude some abstract cycles. Then, Lemma 9.9 gives us indeed a good choice for a ranking predicate.

The theorem shows that the abstract system simulates the parameterized network exactly. Since Lemma 9.2 states the symmetric behavior of all user processes, and moreover, the abstract system refines all guards of the concrete system, we can extend the abstract system by a local abstraction for p, q. Instead of constructing the synchronous product of two user processes with the control process we add abstract variables $p_u \equiv p \in X_u, q_u \equiv q \in X_u$ to establish universal properties.

Corollary 9.11 (Completeness for $\mathcal{S}_{A,p,q}$)
For every abstract system $\mathcal{S}_A$ (Definition 9.3) extended by a local part $p_u \equiv p \in X_u, q_u \equiv q \in X_u$ for every $X_u \in \mathcal{V}$ we can find additional ranking predicates χ_i such that for $\mathcal{S}^{\mathcal{F}}_{A,p,q}$ augmented by the abstract variables $+_{\chi_i}, -_{\chi_i}$

$$\mathcal{S}^{\mathcal{F}}_{A,p,q} \models \psi_A \quad \textit{iff} \quad \mathcal{W} \models \forall p, q, p \neq q : \psi(p, q) \ ,$$

where in ψ_A every occurrence of $p \in X_u$, resp. $q \in X_u$, is replaced by p_u, resp. q_u. □

9.2 Restricted MPS

In [EK00] Emerson and Kahlon prove decidability for a restricted class of monadic parameterized networks (see Definition 2.6). Their work also handles the general case of networks of the form $\mathcal{S}_1^{n_1} \| \ldots \| \mathcal{S}_m^{n_m}$. For the sake of brevity we show how to analyze systems of the form $\mathcal{S}^n$, although our framework also allows to deal with the general case.

To achieve decidability, [EK00] restricts the guards labeling the transitions in an MPS $\mathcal{P}$ based on $\mathcal{S} = (\Sigma, \mathcal{T}, \mathcal{I})$. We allow either to check for the existence of some process in some state or to check that all processes stay in certain states. The next definition formalizes the two new classes of MPS.

Definition 9.12 (Restricted MPS)
Let $\mathcal{P}$ be an MPS based on $\mathcal{S} = (\Sigma, \mathcal{T}, \mathcal{I})$. If for every $\tau \in \mathcal{T}$ the guard of τ is of the form $\mathrm{at\_}\ell_s[i] \wedge \forall_n i \neq j : \bigwedge_{s \in \Sigma} \mathrm{at\_}\ell_s[j]$ we call $\mathcal{P}$ an *MPS with*

conjunctive guards. In the case all the guards are of the form $\text{at\_}\ell_s[i] \wedge \exists i \neq j : \bigwedge_{s\in\Sigma} \text{at\_}\ell_s[j]$ we call U an *MPS with disjunctive guards*. □

As in the previous section we could also define fragments $\text{AF}_\wedge$ and $\text{AF}_\vee$ of AF that allows to characterize the systems defined above.

The restrictions do not allow processes to use their PID to tag themselves as being a special process nor to compare themselves with their neighbors or other distinguished processes. Hence, we can, as in the synchronous setting, provide a lemma stating the total symmetry between processes.

Lemma 9.13 (Symmetry of processes)
For every permutation π over $\{1,\ldots,n\}$ let f_π define a auto-bijection over $\mathcal{P}(n)$ such that $f_\pi : \Sigma^n \longrightarrow \Sigma^n, (s_1,\ldots,s_n) \mapsto (s_{\pi(1)},\ldots,s_{\pi(n)})$.

Then, f_π is a bisimulation.

Proof The proof is similar to the one of Lemma 9.2. ■

Hence, the interesting properties of $\mathcal{P}$ are those that can be stated as $\forall_n p : \varphi(p)$ or $\forall_n p, q, p \neq q : \varphi(p,q)$ (for brevity we restrict ourselves to properties involving at most 2 processes). φ is assumed to be an LTL$\backslash\bigcirc$ formula of atomic propositions over process states Σ_p and Σ_q.

That we have to abstain from the next operator is obvious, since in the asynchronous parameterized setting the time when a process may advance strongly depends on the number of involved processes.

9.2.1 Abstraction of MPS with Disjunctive Guards

The translation of MPS into a WS1S system $\mathcal{W}$ introduces set variables X_s for every state $s \in \Sigma$. According to our heuristics in Section 5.2 we construct the abstract system $\mathcal{S}_A$ by introducing abstract variables defined as $b_s \equiv \exists i : i \in X_s$ for every state $s \in \Sigma$. Additionally, we add also variables $\bar{2}_s \equiv \exists i, j : i \ = j \wedge i \in X_s \wedge j \in X_s$ for every state $s \in \Sigma$ to check for the existence of at least two processes in some state. Note that this is necessary because a process may check for the existence of a further process at the same location. This is enough to refine the guards of restricted MPS, i.e., every guard is expressible by some boolean combination of the abstract variables. Additionally, to express the desired universal properties we introduce abstract variables for the local abstraction of processes p and q. Hence, we add variables $p_s \equiv p \in X_s$ and $q_s \equiv q \in X_s$ for every $s \in \Sigma$.

Moreover, for every combination of set variables $X_1,\ldots,X_l$ we introduce $\chi \equiv i \in X_1 \cup \ldots \cup X_l$ as a ranking predicate and add the fairness condition

$\Box\Diamond -_\chi \rightarrow \Box\Diamond +_\chi$. In the remainder of this section we denote the resulting abstract system with $\mathcal{S}_A^{\mathcal{F}} = (\mathcal{V}_A, \mathcal{T}_A, \Theta_A, \mathcal{F})$.

Since we have defined conservative abstractions only, it is clear that for every computation in $[\![\mathcal{W}]\!]$, we find a corresponding computation in $[\![\mathcal{S}_A]\!]$. As in Lemma 9.4 we would like to state that all finite prefixes of computations in $[\![\mathcal{S}_A]\!]$ can be simulated in the original system. Unfortunately, the asynchronous semantics gives us somehow the possibility to count as shown in the next example.

Example 9.14 (Counterexample (asynchronous))
Assume a parameterized network given by the process template shown in Figure 9.3 as a labeled transition graph. The computations $[\![\mathcal{S}_A]\!]$ of the

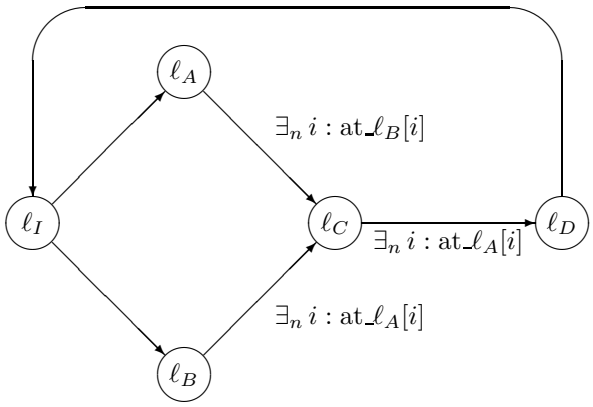


Figure 9.3: Process template

abstract system contains the finite path

$$\begin{aligned}
\langle\ & (b_I, \bar{2}_I, \neg b_A, \neg\bar{2}_A, \neg b_B, \neg\bar{2}_B, \neg b_C, \neg\bar{2}_C, \neg b_D, \neg\bar{2}_D) && (9.1)\\
& (b_I, \bar{2}_I, b_A, \neg\bar{2}_A, \neg b_B, \neg\bar{2}_B, \neg b_C, \neg\bar{2}_C, \neg b_D, \neg\bar{2}_D) && (9.2)\\
& (b_I, \bar{2}_I, b_A, \neg\bar{2}_A, b_B, \neg\bar{2}_B, \neg b_C, \neg\bar{2}_C, \neg b_D, \neg\bar{2}_D) && (9.3)\\
& (b_I, \bar{2}_I, \neg b_A, \neg\bar{2}_A, b_B, \neg\bar{2}_B, b_C, \neg\bar{2}_C, \neg b_D, \neg\bar{2}_D) && (9.4)\\
& (b_I, \bar{2}_I, \neg b_A, \neg\bar{2}_A, \neg b_B, \neg\bar{2}_B, b_C, \bar{2}_C, \neg b_D, \neg\bar{2}_D) && (9.5)\\
& (b_I, \bar{2}_I, \neg b_A, \neg\bar{2}_A, \neg b_B, \neg\bar{2}_B, b_C, \bar{2}_C, b_D, \neg\bar{2}_D)\rangle\ . && (9.6)
\end{aligned}$$

Via ℓ_a and ℓ_B two processes enter ℓ_C, hence $\bar{2}_C$ gets *true*. Then, one process leaves ℓ_C towards ℓ_D. Hence, we would expect $\bar{2}_C$ to become *false*. This is not the case above, since $\bar{2}_C$ indicates the existence of one or more processes at ℓ_C. Hence, in the abstract model $\bar{2}_C$ still holds, although this is not possible in the concrete model. □

The solution to that problem is the fact that we cannot observe the complete system. In order to verify universal properties, the observables are the locations of one or two processes in certain states. That is what φ talks about. Hence, to concretize a counter example for φ found in the abstract system, we have only to take care on the moves of the processes under observation. As shown in the next lemma, the definition of restricted MPS with disjunctive guards allows to change the context for the observed processes such that the counter example can be simulated at the concrete level. Those introduced transitions do not alter the atomic propositions of φ, hence they occur as stutter transitions to φ. Since we have no next operator this suffices to prove the next lemma.

Lemma 9.15 (Concretization of finite paths (asynchronous)) *For every finite path $\sigma : s_0, \ldots, s_{d-1}$ in the abstract system $\mathcal{S}_A$ we can find a concrete path $\widetilde{\sigma}$ in $\mathcal{W}$ such that $\alpha^1(\widetilde{\sigma})$ is stuttering of σ, where α^1 omits the $\bar{2}$ variables. Moreover, we have $\widetilde{\sigma} \not\models \varphi$.*

Proof Initially, we need at least as much processes as $k = |\{b_s \mid b_s \in \mathcal{V}_A, \quad \sigma_0 \models b_s\}|$ since that corresponds to the existence of elements in X_u. But to cover all possible runs of a process we start with $k \cdot d$ processes, i.e., with d processes in each initial user state. In every step only one process changes its state, hence, at most d processes may move. Therefore, we start with d processes in every initial state. By Lemma 9.13 we can choose processes 1 and 2 as concretization of p and q and let them start as indicated by σ.

Since we have $(\sigma_i, \sigma_{i+1}) \models \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_A)$ for all $i < d-1$ and the abstract variables refine the guards, we know that τ is also enabled for a concrete state $\widetilde{\sigma}_i$ with $(\widetilde{\sigma}_i, \sigma_i) \models \widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$. Also, we know exactly when to move process 1 or 2 and it is possible due to the argumentation above. Moreover, the transition always stays enabled for further processes when having more than one process in $source(\tau)$. Hence, when the abstract transition requires $source(\tau)$ to be a singleton afterwards, we can model this on the concrete level by letting all but one process leave $source(\tau)$ via τ. Moreover, when $\bar{2}_{source(\tau)}$ stays *true* although the concretization so far contains not enough processes at $source(\tau)$ (see example above) we can duplicate transitions of processes leading to $source(\tau)$. This can be proved by simple induction and the fact that we have initially as much processes as needed. Thus, we can take care that we have the needed amount of processes in certain locations. Since we have initially d processes in each starting location and at most d different processes can move during σ we can find a $\widetilde{\sigma}$ with potentially more steps but with $\alpha^1(\widetilde{\sigma})$ being a stuttering of σ. Since up to stuttering the altering of p_s and q_s in σ and $\widetilde{\sigma}$ is the same, we have $\widetilde{\sigma} \not\models \varphi$. ∎

Up to stuttering the lemma above allows us to find concrete paths in $\mathcal{W}$ for every finite abstract path. Hence, it remains to show that also infinite computations in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ can be simulated at the concrete level. Then, we can state the following theorem.

Theorem 9.16 (Completeness for MPS with disjunctive guards) *For the translated WS1S system $\mathcal{W}$ of an MPS $\mathcal{P}$ with disjunctive guards and the abstract system $\mathcal{S}_A^{\mathcal{F}}$ as defined in Section 9.2.1 we can state*

$$\mathcal{P} \models \varphi \quad \textit{iff} \quad \mathcal{S}_A^{\mathcal{F}} \models \varphi_A \ ,$$

where φ is an universal LTL formula without $\bigcirc$ operator.

Proof $\Longleftarrow$: This direction follows directly from the definition of abstraction in Section 4.2 and Lemma 7.7.

$\Longrightarrow$: This implication states that $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ contains no false negatives. By contraposition we have to show that every counterexample in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ violating φ_A corresponds to a violation of φ by $\mathcal{W}$. The correspondence of $\mathcal{W}$ and $\mathcal{P}$ is established in Chapter 3.

Let $\sigma : s_0, s_1, \ldots$ be in $[\![\mathcal{S}_A^{\mathcal{F}}]\!]$ such that $\sigma \not\models \varphi_A$. Since φ_A is an LTL formula we can w.l.o.g. assume σ to have the form

$$\sigma_0, \ldots, \sigma_{l-1}, (\sigma_l, \ldots, \sigma_{l'})^\omega \ .$$

The literals in φ_A are of the form p_s, q_s for $s \in \Sigma$. Let $\widetilde{\sigma}$ be a concretization of $\sigma_0, \ldots, \sigma_{l'}$ as in Lemma 9.15. The behavior of processes 1 and 2 is determined by the abstract variables p_s, q_s. All processes occupying states indicated by σ_l perform a path to locations indicated by $\sigma_{l'}$. As in the proof for synchronous systems these paths allow to construct a graph showing the exchange of processes during the cycle $(\sigma_l, \ldots, \sigma_{l'})$. The SCCs of this graph are unconnected, otherwise the fairness conditions would have excluded it. Hence, we can find a k such that we find a concretization $\widetilde{\sigma}$ of $\sigma_0, \ldots, \sigma_{l-1}, (\sigma_l, \ldots, \sigma_{l'})^k$ and the concrete processes cycle in $(\sigma_l, \ldots, \sigma_{l'})^k$. Hence, we have found a concrete counterexample for φ. ∎

9.2.2 Abstraction of MPS with Conjunctive Guards

In the disjunctive case widening the set of occupied states enables more transitions to be taken. Hence, a path taken by one process could be taken by arbitrary many processes. This makes it possible to concretize abstract

paths even though the asynchronous semantics allows somehow to count processes due to the steps they have taken. Because the count can be altered arbitrarily in certain states.

On the contrary, in the conjunctive case spreading the processes over the states disables more and more transitions. This makes it more difficult to overcome the counting problem as indicated by the following example.

Example 9.17 (MPS with conjunctive guards)
Consider a process template as given in Figure 9.4. Processes entering loca-

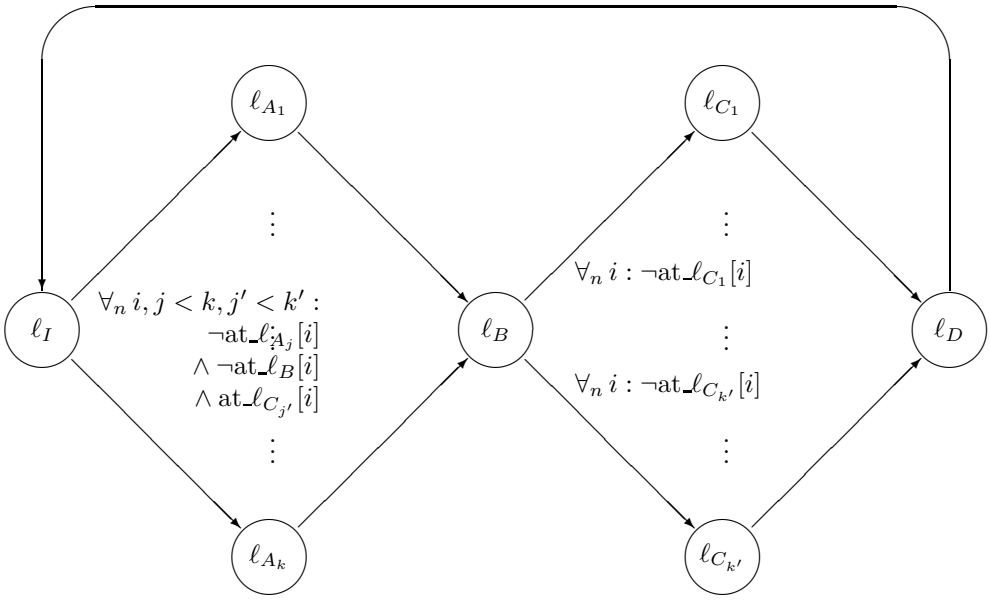


Figure 9.4: Process template for conjunctive MPS

tions ℓ_{A_j} disable the corresponding transition. Hence, at most k processes can gather at ℓ_B. In a situation where $k' > k$ it is obvious that not all ℓ_{C_j} can be covered by processes. This indicates that our abstraction is not strong enough to prove such a property. □

Indeed, in [EK00] they prove that for such systems it is necessary but also sufficient to verify the networks up to the size $|\Sigma|$, i.e., one needs to count up to the number of states. On the other hand, the problem of the example occurs because we are considering a global property. When restricting ourselves to universal properties the conjunctive case gets trivial (still assuming deadlock free systems). As already mentioned widening the set of occupied states disables more transitions. Hence, a counterexample generated by an local abstraction focusing on processes p and q can be concretized always.

One chooses a setting with only two processes and ignores steps of other processes.

Technically, one could also choose a counter abstraction counting the number of processes up to the number of total states as indicated by the solution of [EK00]. Practically, this yields $|\Sigma|^2$ states for the abstract system in our framework, since we would have a boolean variable for each locations and each count upto $|\Sigma|$. Therefore, it is in the same order of magnitude as the system consisting of $|\Sigma|$ processes ranging over $|\Sigma|$ states.

Part IV

Extending the Systems

Chapter 10

Multi-dimensional Parameters

In practice it often occurs that the processes in a network themselves are infinite because of unbounded data structures. One source of unboundedness can be the usage of a parameterized data structure. Another typical source may be the presence of structures ranging over subsets of participating processes. E.g., this is the case for group membership or distributed shared memory consistency protocols. In this chapter we use deductive methods to deal with such networks where the data structure is parameterized by the number of processes and an extra parameter. We show how to derive an abstract WS1S system which can be subject to algorithmic verification. For illustration of the method we verify the correctness of a time-triggered group membership protocol using strengthening and of a distributed shared memory consistency protocol using PVS for the deductive verification part and the tools PAX and SMV for the algorithmic part.

10.1 A Time-Triggered Group Membership Protocol

The group membership protocol we want to verify was presented by S. Katz, P. Lincoln, and J. Rushby in [KLR97]. It represents the interesting class of time-triggered algorithms which recently draws attention by the formal verification community. In [Rus02] an overview of verification approaches dedicated to this class can be found. The protocol chosen here is also tackled in [BM02] by means of counter abstraction. The protocol requires one bit of membership information piggybacked on regular broadcasts. With assumptions on the fault model, the protocol guarantees that a faulty processor will be diagnosed and removed from the agreed group of non-faulty processors. Therefore, each processor p keeps a set of processor IDs mem(p) that he

believes to be non-faulty.

10.1.1 Protocol Description

If $p \in \text{mem}(p)$, then process p takes one of the following transitions synchronously with the other processes.

$$
\begin{array}{rcl}
\neg\text{arrived}(p) \wedge \neg\text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p) \setminus \{turn, p\}; \\
 & & \text{ack}'(p) := \mathit{ff} \\
\neg\text{arrived}(p) \wedge \text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p) \setminus \{turn\}; \\
 & & \text{ack}'(p) := \mathit{ff} \\
\text{arrived}(p) \wedge \text{ack}(turn) \wedge \neg\text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p) \setminus \{p\}; \\
 & & \text{ack}'(p) := \mathit{tt} \\
\text{arrived}(p) \wedge \neg\text{ack}(turn) \wedge \text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p) \setminus \{turn\}; \\
 & & \text{ack}'(p) := \mathit{ff} \\
\text{arrived}(p) \wedge \text{ack}(turn) \wedge \text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p); \text{ack}'(p) := \mathit{tt} \\
\text{arrived}(p) \wedge \neg\text{ack}(turn) \wedge \neg\text{ack}(p) & \rightarrow & \text{mem}'(p) := \text{mem}(p); \text{ack}'(p) := \mathit{tt} \\
p = turn & \rightarrow & \text{mem}'(p) := \text{mem}(p); \text{ack}'(p) := \mathit{tt}
\end{array}
$$

It is process *turn* which has to send at the moment. arrived(p) stands for: $turn \neq p$, p is not receive-faulty, and process *turn* is not send-faulty at this step. The *turn* variable is increased by 1 modulo n in each step.

If $p \notin \text{mem}(p)$, process p does not send. The fault assumption is that new faults arrive at least $n+1$ time units apart, when there are n processes. A fault may be that a processor is unable to send or to receive. Let us denote with OK the set of non-faulty processors. Then, we like to prove the following properties:

Agreement: $p \in OK \wedge q \in OK \rightarrow \text{mem}(p) = \text{mem}(q)$

Validity: $p \in OK \rightarrow \exists q : \text{mem}(p) \cup \{q\} = OK$

Since p and q are arbitrary processes both properties are universal as defined in Chapter 5.

10.1.2 Generic Abstractions

Our goal is to characterize such time-triggered systems as WS1S systems. Then, we can apply the presented automatic abstraction and model-checking

techniques to analyze the system. To be able to do so each process of the system has to be finite state. This is not the case here, since each process stores the set mem(p). But, the properties only talk about the memory of one or two processes. Hence, the idea is to make an abstraction concentrating on these processes and give a finite-state abstraction for the others.

For the first property we keep two original processes p, q and leave the others as chaotic processes. Nevertheless, we prove that p, q agree on the set of non-faulty processes in every synchronous step. The definition of the abstract system is straightforward and not presented here.

For the property of validity we concentrate on one process p. Since, we already have proven agreement the others may use p's memory Mem_p as their own.

Our generic abstraction gives us an parameterized number of finite-state processes and a finite number of processes with unbounded state space. Since all finite-state processes are similar we introduce for each of the possible states a set variable holding those process IDs which are in the corresponding state. The remaining ones are modeled as they are.

In each computation step of the abstract time-triggered systems one process is broadcasting and all others are receiving this message. The broadcasting process is denoted by the special variable *turn* which is increased by 1 modulo n in each step.

The fault model can be encoded in these WS1S transitions using the variables $s\_fault, r\_fault, err\_prop$ which characterize whether a send or receive error has occurred and whether a new error is allowed yet or not. The variable *malfunc* holds the ID of the faulty processor.

For the abstract system and the property of validity we define our abstract system as shown in Table 10.1.

10.1.3 Verification Results

Using this abstraction we can automatically construct the abstract system with our tool PAX. Translating the abstract system to the SMV input language, we can use NuSMV to prove the following properties:

$$\begin{aligned}&\Box(p\_ok \wedge \neg r\_fault) \rightarrow \\ &\quad \Box(s\_fault \wedge \bigcirc((\neg s\_fault \wedge \neg \bigcirc s\_fault)\, \mathcal{U}\, t\_mem) \\ &\qquad \rightarrow \Diamond(equal \wedge stable))\end{aligned}$$

and

$$\Box(p\_ok \wedge \neg s\_fault \wedge (r\_fault \rightarrow \bigcirc \Box(\neg r\_fault))) \rightarrow$$

Table 10.1: Definition of the abstract system

abstract variable	description
$p\_ok \equiv p \in OK$	process we focus on is non-faulty
$t\_ack \equiv turn \in \text{Ack}$	turn process believes everything is ok
$t\_mal \equiv turn = malfunc$	turn process is faulty
$t\_mem \equiv turn \in \text{Mem}_p$	p believes the *turn* process to be ok
$conf \equiv \text{Ack} \cap \text{Mem}_p = \emptyset$	non-flty procs agree there is something wrong
$equal \equiv \text{Mem}_p = OK$	non-flty procs have correct membership set
$stable \equiv \text{Mem}_p \subseteq \text{Ack}$	everything is fine
$super \equiv (\exists q : \text{Mem}_p = OK \cup \{q\} \wedge \text{Mem}_p \subseteq \text{Ack} \cup \{q\} \wedge q = malfunc)$ one faulty process not registered by the non-flty ones	
s_fault	send error occurred
r_fault	receive error occurred
err_prop	fault model allows no new error yet

$$\Box(r\_fault \wedge \Diamond(t\_mal \wedge \Diamond \bigcirc t\_mem) \rightarrow \Diamond(equal \wedge stable))$$

The first one states under the assumption that the process we focus on is ok and in absence of receive errors; whenever a send error occurs all non-faulty processes eventually notice that and remove the faulty process from their membership set, provided that no new errors occur. The second one expresses a corresponding behavior for receive errors.

10.2 Cache Coherence

As another example we verify a distributed shared memory consistency protocol by Li and Hudak presented in [LH89]. The idea of distributed shared memory (DSM) is to allow processors in a distributed environment to utilize each other's local memories. DSM systems provide a virtual address space for a network of processors. They replicate or migrate data across the network to handle requests of threads running on the processors. The verification results presented here are already published in [BBLS01].

10.2.1 Protocol Description

Li and Hudak's protocol is a multiple-reader, single-writer protocol; several processes are allowed to have read access to the same data while write access is exclusive. The data which is subject of the read/write requests is organized into *pages*. A *page table* on each processor maintains the current access status of the processor for each page. The status may be *read* or *write* and keeps the information whether the processor owns the page. The owner is the processor last having write privileges to a page. Moreover, the status may be *nil* if the processor has no local copy of the page or if the page has been modified by some other processor.

<table>
<tr>
<td>req_rd(p)
PTable[p].lock:=true;
broadcast read request for p;

get_rd_ack(p)
PTable[p].access:=read;
PTable[p].lock:=false;</td>
<td>send_rd_ack(p,i)
PTable[p].lock:=true;
IF PTable[p].owner THEN
PTable[p].copyset:=PTable[p].copyset ∪ {i};
PTable[p].access:=read;
send ack and p to i; FI
PTable[p].lock:=false;</td>
</tr>
<tr>
<td>req_wr(p)
PTable[p].lock:=true;
broadcast write request for p;

get_wr_ack(p,copy-set)
PTable[p].access:=inv;
req_invalidate(p,copyset);</td>
<td>send_wr_ack(p,i)
PTable[p].lock:=true;
IF PTable[p].owner THEN
send p and copyset to i;
PTable[p].access:=nil;
PTable[p].owner:=false; FI
PTable[p].lock:=false;</td>
</tr>
<tr>
<td>req_invalidate(p, copyset)
FOR k in copyset DO
send inv request for p to k;

read_nil_ack(i)
PTable[p].copyset:=
PTable[p].copyset \ {i};</td>
<td>send_inv_ack(p)
IF PTable[p].access=read THEN
PTable[p].access:=nil;
send acknowledgment; FI</td>
</tr>
<tr>
<td colspan="2">get_owner
IF PTable[p].copyset=∅ AND PTable[p].access=inv THEN
PTable[p].access:=write; PTable[p].copyset:=∅;
PTable[p].owner:=true; PTable[p].lock:=false; FI</td>
</tr>
</table>

Figure 10.1: Pseudo-code of one processor according to the DSM protocol

When a processor wants to upgrade its privileges (from nil to read or from read to write) it sends a corresponding request via broadcast. The owner handles the request; for a read request it adds the requesting processor to the *copy-set*, the list of processors with read access. Processors in the copy-

set have to be informed when a processor wants write access. Each of these processors has to acknowledge that it changed its page table for that page to nil. Then, the ownership changes to the requesting processor. The pseudo-code in Figure 10.1 describes the various actions [FG98].

As communication mechanism we assume to have one central request queue where all the requests are sent to. Acknowledgments are sent directly to the requesting processors. The description of the protocol immediately gives us four verification goals:

Exclusive Ownership: For each page p there is always at most one owner.

Exclusive Write: Whenever there is a processor having write access for p then there is no processor with read access.

Copy-Set Adequacy: Processors having read access to page p are always in the copy-set of the owner of p except for the owner itself.

Liveness: Whenever a processor requests access privileges it eventually obtains them.

10.2.2 System Reduction using PVS

Our aim is to verify the DSM protocol by Li and Hudak using abstraction techniques. In [BBLS00] we presented how to compute abstractions automatically for networks with finite processes modeled in the decidable logic WS1S. Unfortunately, Li and Hudak's protocol handles a parameterized number of pages (beside the parameterized number of processors) and each page table entry contains a set (*copy-set*) of processor indices.

To reduce the state space of the processors we introduce a *global-copy* (see Figure 10.2 for the PVS declaration). This variable is global to all processors. We added assignments to it whenever the processor having the ownership for one page updates its local *copy-set*. If we can prove that the local *copy-set* of some processor, whenever needed, coincides with the *global-copy* we can get rid of all the local *copy-sets*.

To deal with the second parameter (the page parameter) we identify a class of parameterized networks for which we can verify certain properties by instantiating the second parameter with 1.

These intermediate steps give us a reduced system presentable in WS1S but still allows to prove properties for the original protocol. We use PVS [ORSvH95] to establish the reduction. Since PVS allows to use higher order logic for specifications it is straightforward to give a PVS model for the

```
Access: TYPE = {write, nil, read, invalidate}
Requests: TYPE = {read_req, write_req}
Direct: TYPE = {write_ack, read_ack, nil_req}

N, P: posnat

Processor: TYPE = below(N)
Page: TYPE = below(P)

Copy_set_type: TYPE = setof[Processor]

Req_channel: TYPE = list[[Processor, Requests, Page]]
Direct_channels: TYPE = [Processor → list[[Direct, Page]]]
Nil_Acks: TYPE = [Page → list[Processor]]

PageEntry: TYPE =
[# access: [Processor → Access],
   owner?, locked?: [Processor → bool],
   copy_set: [Processor → Copy_set_type],
   send_copy, global_copy: Copy_set_type #]

State: TYPE =
[# PageTable: [Page → PageEntry],
   Reqs: Req_channel,
   DirectCom: Direct_channels,
   Nils: Nil_Acks #]
```

Figure 10.2: Parts of PVS theory

```
s, s1: VAR State

i: VAR Processor
p: VAR Page

req_rd(s, s1, i, p): bool =
    access(PageTable(s)(p))(i) = nil ∧
     ¬ locked?(PageTable(s)(p))(i) ∧
      s1 = s WITH [(PageTable)(p) := PageTable(s)(p)
                                  WITH [(locked?)(i) := TRUE],
                  Reqs := cons((i, read_req, p), Reqs(s))]
```

Figure 10.3: Requesting read access

pseudo-code in Figure 10.1. Figure 10.3 shows an example transition expressed as a relation between pre- and post-state.

Using PVS it was straightforward to prove that indeed the *global-copy* matches with the local *copy-set* whenever needed. The property (see Figure 10.4) is inductive over all transitions and initially fulfilled and therefore invariant.

```
Global_Local(s, i, p): bool =
    (owner?(PageTable(s)(p))(i) ⊃
      copy_set(PageTable(s)(p))(i) = global_copy(PageTable(s)(p)))
```

Figure 10.4: Equality of *global-copy* and *copy-set* for owner

Generality of this step As already mentioned network protocols often handle data structures ranging over sets of processors in the network. As examples we listed shared memory protocols such as the one described in Section 10.2.1 or group membership protocols. To illustrate how the idea to reduce the state space of each processor by introducing a global structure used by all processors can be applied to other protocols, let us consider a group membership protocol. Briefly, each processors keeps track which processors are still alive and vital part of the network. Due to continuous communication errors are detected and processors are removed from the membership-list of the well functioning processors. Two properties are of interest for those protocols; *agreement*, meaning that the well functioning processors agree on their membership-lists, and *validity*, meaning that an error will be detected eventually and then the membership-lists corresponds to the set of well functioning processors. We analyzed a synchronous group membership protocol by proving first agreement deductively. Then, we could reduce the system by using a global membership-list maintained by the processors working properly.

Now, back to our DSM protocol. Even if we remove all local copy-sets we have a system of processors where each of them maintains a page table for a parameterized number of pages. But, we observe (and can prove with PVS) that all transitions alter only the page entry for exactly one page and consume or produce only messages concerning this page. We fix these assumptions in the next definition.

Definition 10.1 (Strict parameterization)
Let $\mathcal{S}(N, P)$ be a parameterized system with N processors and a second parameter P. Let the processors communicate over some message queues

$q_1, \ldots, q_k$ where each message is of the form (i, msg, p) with $i < N$, $p < P$, and msg of some finite type M. The state space of each processor j is an array $a[j][0..P-1]$ of size P and of finite type T.

We call $\mathcal{S}(N, P)$ *strictly parameterized* in P if each processor has initially the same configuration for each page:

$$\forall i < N : \forall p_1, p_2 < P : a[i][p_1] = a[i][p_2] \wedge \bigwedge_{l=1}^{k} q_l = \langle\rangle \ ,$$

and each transition in $\mathcal{S}(N, P)$ is either

- an internal step $a[i][p] = t_1 \wedge a' = a([i][p] \mapsto t_1') \wedge \bigwedge_{l=1}^{k} q_l' = q_l$
- or a communication step

$$\begin{aligned} &a[i][p] = t_1 \wedge a[j][p] = t_2 \wedge a' = a([i][p] \mapsto t_1', [j][p] \mapsto t_2') \wedge \\ &[q_l = \langle j, msg_1, p\rangle \cdot q_l' \wedge msg_1 = m \wedge][q_o' = q_o \cdot \langle j, msg_2, p\rangle \wedge] \\ &\bigwedge_{r \neq l,o} q_r' = q_r \ , \end{aligned}$$

where t_1, t_2, t_1', t_2' are constants in T and m, msg_1, msg_2 are constants in M. Concerning the communication step either the message consuming part $q_l = \langle j, msg_1, p\rangle \cdot q_l'$ or the message producing part $q_o' = q_o \cdot \langle j, msg_2, p\rangle$ may be empty. We call such transitions changing only array entries for p and handling only *p-messages*, *p-transitions*. □

As usual we denote with $[\![\mathcal{S}(N, P)]\!]$ the set of computations $\sigma = s_0, s_1, \ldots$ of $\mathcal{S}(N, P)$. Now, let $[\![\mathcal{S}(N, P)]\!] \downarrow_p$ denote those computations $\sigma^p = s_0^p, s_1^p, \ldots$ derived from the original ones by projecting the array $a[0..N-1][0..P-1]$ to the array entry $a[0..N-1][p]$ and projecting the contents of the communication channels to messages with tag p. Then, we can show:

Lemma 10.2 (Reducing the dimension)
For a system $\mathcal{S}(N, P)$ strictly parameterized in P we have for all P and $p < P$

$$[\![\mathcal{S}(N, P)]\!] \downarrow_p = \ [\![\mathcal{S}(N, P)]\!] \downarrow_1 = \ [\![\mathcal{S}(N, 1)]\!]^\tau \ ,$$

where $[\![\cdot]\!]^\tau$ denotes the set of computations with arbitrary interleaved idle transitions.

Proof 'first equality': Concerning one processor i each computation starts with $a[i][p] = a[i][m] = t$ for $t \in T(N)$ and $p, m < P$. The message queues are empty. Hence, the same transitions for p and m are enabled which produce

the same messages (msg, p) resp. (msg, m) and yield the same array entry t'. Hence, in each computation we can exchange p- and m-transitions. This gives us the first equality.
'second equality': If we consider in $\mathcal{S}(N, P)$ only 1-transitions to happen, trivially we have $[\![\mathcal{S}(N, 1)]\!] \subseteq [\![\mathcal{S}(N, P)]\!]$. Since all p-transitions with $p \neq 1$ do not alter $a[i][1]$ for any processor i and do not consume or produce 1-messages they are idle transitions in $[\![\cdot]\!] \downarrow_1$. Hence, we have $[\![\mathcal{S}(N, P)]\!] \downarrow_1 \subseteq [\![\mathcal{S}(N, 1)]\!]^\tau$.
■

The introduction of idling transitions is needed because we have made no assumptions about fairness. Hence, it is not guaranteed that some p transitions occur or when they occur as long as other transitions are enabled. Nevertheless concerning invariance properties we get immediately:

Corollary 10.3 (Safety for two-dimensional systems)
Let $\mathcal{S}(N, P)$ a system strictly parameterized in P. Let $\varphi(p)$ be a state formula. Then, we have:

$$\mathcal{S}(N, 1) \models \Box\varphi(1) \text{ iff } \mathcal{S}(N, P) \models \forall p < P : \Box\varphi(p) \ .$$

□

To deal with liveness properties we need some assumptions about fairness. We call $\mathcal{S}(N, P)$ *weak fair* if all transitions continuously enabled from a certain point in a computation are eventually taken. If moreover $\mathcal{S}(N, 1)$ never blocks a queue, i.e., messages in the queues are eventually consumed, we can deduce from Lemma 10.2:

Corollary 10.4 (Liveness for two-dimensional systems)
Let $\mathcal{S}(N, P)$ be a system strictly parameterized in P and assume $\mathcal{S}(N, P)$ is weak fair. Let $\varphi(p)$ be a property expressed in LTL/X. Then, if $\mathcal{S}(N, 1)$ never blocks a queue we have:

$$\mathcal{S}(N, 1) \models \varphi(1) \text{ iff } \mathcal{S}(N, P) \models \forall p < P : \varphi(p) \ .$$

□

Now, we observe that our reduced system is indeed strictly parameterized in P. Hence, we can get rid of the second parameter and are prepared to represent the resulting network in the framework of WS1S.

10.2.3 WS1S Model

We have proven that we can replace the local variables *copy-set* of each processor by one global variable *global-copy*. This and the argument that allows us to verify the system instantiated with only one page concerning the page parameter gives us the possibility to model the reduced system as a WS1S system.

As set of second order variables we choose

$$\begin{aligned}\mathcal{V} \;=\; \{\; & \mathit{Procs}, \mathit{Read}, \mathit{Write}, \mathit{Invalidate}, \mathit{Owner}, \mathit{Locked}, \mathit{Global\_Copy}, \\ & \mathit{Read\_Req}, \mathit{Write\_Req}, \mathit{Nil\_Req}, \mathit{Read\_Ack}, \mathit{Write\_Ack}, \mathit{Nil\_Ack} \;\}\ .\end{aligned}$$

We have no set to represent the processors being in state *nil*. We assume those which are not in *read*, *write*, and *invalidate* to have nil access to the page. In WS1S it is not possible to represent queues, hence, we model them as sets. This, of course, gives us an abstraction. On the other hand we lose some fairness, e.g., when a request of processor i is in the queue, then the request is eventually granted when read by the owner, or the owner eventually stops to handle any request:

$$\Box(i \in \mathit{Read\_Req} \rightarrow \Diamond(i \in \mathit{Read\_Ack}) \vee \Diamond\Box\neg\texttt{send\_rd\_ack})$$

Hence, we add those fairness conditions $\mathcal{F}$ to the WS1S system. The initial condition of our WS1S system reads as follows:

$$\begin{aligned} & (\exists n : \forall i : i < n \rightarrow i \in \mathit{Procs}) \\ \wedge\ & (\exists j : j \in \mathit{Procs} \wedge \mathit{Read} = \{j\} \wedge \mathit{Owner} = \{j\}) \\ \wedge\ & \mathit{Locked} = \emptyset \wedge \mathit{Write} \cup \mathit{Invalidate} = \emptyset \\ \wedge\ & \mathit{Read\_Req} \cup \mathit{Write\_Req} \cup \mathit{Nil\_Req} = \emptyset \\ \wedge\ & \mathit{Read\_Ack} \cup \mathit{Write\_Ack} \cup \mathit{Nil\_Ack} = \emptyset\ . \end{aligned}$$

To illustrate how the transitions of our PVS specification (simplified by reduction to one page) can be expressed in WS1S we give here $\rho_{\texttt{req\_rd}}$:

$$\begin{aligned} \exists i : \quad & i \notin \mathit{Read} \cup \mathit{Write} \cup \mathit{Invalidate} \cup \mathit{Locked} \\ \wedge\ & \mathit{Locked}' = \mathit{Locked} \cup \{i\} \wedge \mathit{Read\_Req}' = \mathit{Read\_Req} \cup \{i\} \\ \wedge\ & \bigwedge_{X \in \mathcal{V} \setminus \{\mathit{Read\_Req}, \mathit{Locked}\}} X' = X\ . \end{aligned}$$

Next, we want to analyze the WS1S system defined above by abstraction.

10.2.4 Verification Results

Using PVS it was easy to prove some intuitive properties to be inductive over all transitions and therefore being invariant. These properties state that the processor having write privileges for some page is also the owner, that ownership is exclusive, and that the owner knows the processors with read access (see Figure 10.5). Hence, we have already two of our four properties,

```
Invariant1(s, i, p): bool =
    access(PageTable(s)(p))(i) = write ⊃
     ¬ locked?(PageTable(s)(p))(i) ∧ owner?(PageTable(s)(p))(i)

Invariant2(s, p): bool =
  (∃ (i: Processor):
    owner?(PageTable(s)(p))(i) ⊃
     (∀ (j: Processor): ¬ (j = i) ∧ ¬ owner?(PageTable(s)(p))(j)))

Invariant3(s, p): bool =
   ∀ (j: Processor): ∃ (i: Processor):
      (access(PageTable(s)(p))(j) = read ∧
        ¬ owner?(PageTable(s)(p))(j) ∧ owner?(PageTable(s)(p))(i))
         ⊃ (j ∈ copy_set(PageTable(s)(p))(i))
```

Figure 10.5: Invariants proven with PVS

namely exclusive ownership and copy-set adequacy. As stated in Section 4.2 we can use these invariants to strengthen our WS1S system.

Using PAX we can successively check for more invariants. It is possible to strengthen the exclusive ownership property to

$$\begin{aligned} & \mathit{Owner} \cap \mathit{Write\_Ack} = \emptyset \\ \wedge \quad & \mathit{Owner} \cap \mathit{Invalidate} = \emptyset \\ \wedge \quad & \mathit{Write\_Ack} \cap \mathit{Invalidate} = \emptyset \\ \wedge \quad & \exists j : \{j\} = \mathit{Owner} \cup \mathit{Write\_Ack} \cup \mathit{Invalidate} \end{aligned}$$

which states that to grant a write request the owner gives away the ownership of the page to acknowledge the request. The acknowledgment causes the requesting processor to go in the invalidate state to inform all processors having read privileges. We take the property given above as the definition for one abstract variable (or we can take one variable for each conjunct). The resulting state space of the abstract system has exactly one state, namely the state where the variable is (or all variables are) *true*.

The first of the remaining two properties, exclusive write access, can be expressed as:

$$\forall i, j : \Box(i \in \mathit{Write} \rightarrow j \notin \mathit{Read}) \tag{10.1}$$

This is because we have already proven exclusive ownership and that the processor with write privileges is the owner. The second one, that a request will be eventually granted, is described by the LTL formula:

$$\forall i : \Box(i \in \mathit{Write\_Req} \rightarrow \Diamond(i \in \mathit{Write} \wedge i \in \mathit{Owner})) \tag{10.2}$$

Both properties are universal as defined in the previous section, hence, we define an abstraction function concentrating on two processors i, j:

$$\begin{aligned} \alpha(\mathcal{V}, \mathcal{V}_A, i, j) \quad \equiv \quad & i \neq j \wedge \\ & \bigwedge_{X \in \mathcal{V}} a_i^X \leftrightarrow i \in X \wedge \\ & \bigwedge_{X \in \mathcal{V}} a_j^X \leftrightarrow j \in X \end{aligned}$$

The PAX tool generates a finite abstract system and translates it to the SMV input language. Moreover, PAX automatically adds new boolean variables to the SMV specification for each transition to monitor which transition was taken in the last step.

SMV verifies property 10.1 in about half a second. But SMV fails to prove property 10.2. It generates a counter example which loops with the transitions requesting for read or write access. At the concrete level this is not possible, in fact, each processor locks the page when performing a request. The next request by that processor can only be done after unlocking the page which corresponds to receiving the requested privileges. Hence, in the concrete system we can have only as much requests as processors.

To generate an adequate fairness condition to exclude such counterexamples, we define a ranking predicate $\chi \equiv i \in \mathit{Procs} \setminus \mathit{Locked}$. Hence, we get the fairness condition:

$$\Box\Diamond(\texttt{req\_rd} \vee \texttt{req\_wr}) \rightarrow \Box\Diamond(\texttt{get\_rd\_ack} \vee \texttt{get\_wr\_ack})$$

Note, the generated fairness conditions are guaranteed to hold in the concrete system and are therefore not part of its fairness requirements $\mathcal{F}$. In contrast, we also need some fairness condition $\mathcal{F}_A$ at the abstract level corresponding to the weak fairness assumptions $\mathcal{F}$ in the concrete system as defined in Section 4.2. Taking these fairness conditions together with the generated ones as assumptions to constrain our abstract system, SMV needs 15 seconds to establish property 10.2. By Corollary 10.4 we have established the correctness of the original distributed shared memory protocol.

Chapter 11

Induction on Processes

In Chapter 5 we give several heuristics to choose adequate abstraction predicates. There, we introduce two kinds of abstractions, global and local abstractions. A global abstraction describes the overall system configuration. We use these abstractions to prove that certain configurations will or cannot occur in the computations of a system. For example, we are able to establish mutual exclusion by using a global abstraction. In contrast, for universal properties, i.e., properties that are expected to hold for every single process, we suggest to apply local abstractions which focus on those single but arbitrary processes. If that approach succeeds the property holds for every process in the network.

One reason for not succeeding with that approach is that a process does not fulfill the expected property without any additional assumptions on the behavior of its neighbors. In the framework of MPS we consider systems with broadcast communication via shared variables. Alternatively, we can also model linear networks or rings when using the successor function in WS1S to address the neighbors of a process. For example, in a network with ring topology we may know that a certain process has some information and that this information is traveling around the ring. Hence, we can conclude that eventually every process has that information. In this chapter we present a proof rule to establish such a behavior formally.

11.1 Induction on Linear Topologies

Given a linear network topology we often have the situation that the first process naturally takes a special position inside the system. It informs its right (or left) neighbor to proceed in a certain state. The neighbor is going to pass on this information to its right neighbor and so on. Since we are

dealing with finite networks only, this propagation clearly terminates and all processes have entered the particular state. This induction principle on linear network topologies is formalized in the following theorem.

Theorem 11.1 (INDUCTION rule)
For an arbitrary MPS and an individual property Prop characterizing those process identifiers that are in a certain subset of process states, the following proof rule is correct:

$$\frac{\begin{array}{l}1.\ \Diamond(0 \in \mathit{Prop}) \\ 2.\ \forall p < n-1 : \Box(p \in \mathit{Prop} \rightarrow \Diamond(p+1 \in \mathit{Prop}))\end{array}}{\forall_n\, p : \Diamond(p \in \mathit{Prop})}$$

PROOF Let $\mathcal{P}$ be an MPS and *Prop* be an individual property characterized by some state formulae. Let $\sigma \in [\![\mathcal{P}]\!]$ be a computation of the system. Let $m \in \omega$ such that σ is a run of instance $\mathcal{P}_m$. Assume premises 1 and 2 to hold. According to the definition of LTL formulae in Section 4.1.3, there exists a point i in the computation $\sigma : \sigma_0, \sigma_1, \ldots$ with $\sigma_i \models 0 \in \mathit{Prop}$. Since premise 2 is a statement over all processes, from $0 \in \mathit{Prop}$ we can conclude that eventually $1 \in \mathit{Prop}$. Inductively, we can derive for all processes p part of $\mathcal{P}_m$ that they are eventually in *Prop*. ∎

In practice, premise 1 can often be derived from the initial condition of the MPS. Then, $0 \in \mathit{Prop}$ holds even initially. Otherwise, a local abstraction focusing on process 0 can be used to establish premise 1. In any case, premise 2 is tailored to be proven by a local abstraction focusing on two consecutive processes.

11.2 Example

The standard example for symbolic model-checking with rich assertional languages introduced in [KMM+97] is a linear network passing a token from left to right. In Section 1.4.3 we explain their method of proving invariants by using transducers to compute the set of reachable configurations. The reachable configurations are expressed by regular expressions and can be intersected with another regular expression characterizing the "bad" states.

In our framework we can easily model such a network as WS1S system with a set of participating processes $P = \{0, \ldots, n-1\}$ and a set *Token* containing the processes that have a token (of course, we expect *Token* to be a singleton). Assume that we have initially $0 \in \mathit{Token}$. The only transition passing the token is characterized by

$$\exists p : p < n-1 \wedge p \in \mathit{Token} \wedge \mathit{Token}' = (\mathit{Token} \setminus \{p\}) \cup \{p+1\}\ .$$

To prove the safety property of the network, i.e., that there is always only one token in the system, we choose as single abstract variable

$$\mathit{unique} \equiv \exists_P p : \mathit{Token} = \{p\} \ .$$

The abstract system consisting of a single state shows that there is indeed only one token at every moment.

In [PS00, BJNT00] methods are presented to tread the verification of liveness properties. But, at least in [PS00] the authors state that the presented method is highly inefficient and is only applicable to simple algorithms. In our abstraction-based framework we can efficiently check the abstract system for all kind of properties including liveness properties. We showed that ranking predicates are sufficient to handle the problems of possible false negatives. The INDUCTION rule gives us another approach to establish liveness properties for a parameterized network.

In the case of the token token passing example the desired liveness property is that eventually the token passes every process. We can now use the INDUCTION rule to prove this property. Premise 1 follows immediately from the initial condition. To establish premise 2 we use the local abstraction $\mathit{phastok} \equiv p \in \mathit{Token}$ resp. $\mathit{p1hastok} \equiv p + 1 \in \mathit{Token}$. The abstract system can then be used to prove the stronger property

$$\forall p < n - 1 : \Box(p \in \mathit{Prop} \rightarrow \bigcirc(p + 1 \in \mathit{Prop})) \ .$$

Hence, by the INDUCTION rule we can derive $\forall_n p : \Diamond(p \in \mathit{Token})$.

11.3 Induction on Ring Topologies

The INDUCTION rule can be also applied to ring topologies. Let the neighbor of process i be characterized as $i \oplus_n 1$. In such a setting we can even strengthen the conclusion to state that each process is infinitely often in *Prop*.

Corollary 11.2 (INDRING (induction on rings))
For an arbitrary MPS and an individual property Prop characterizing those process identifiers that are in a certain subset of process states, the following proof rule is correct:

$$\frac{\begin{array}{l}1.\ \Diamond(0 \in \mathit{Prop}) \\ 2.\ \forall_n p : \Box(p \in \mathit{Prop} \rightarrow \Diamond(p \oplus_n 1 \in \mathit{Prop}))\end{array}}{\forall_n p : \Box\Diamond(p \in \mathit{Prop})}$$

□

We can use the corollary to simplify the verification of the token ring of Section 8.3. Instead of proving separately that the process possessing the token passes it on and proving that its neighbor is ready to receive it, we can apply the IndRing rule.

We can also modify our simple example from above by changing the transition relation to

$$\exists p : p < n - 1 \wedge p \in \mathit{Token} \wedge \mathit{Token}' = (\mathit{Token} \setminus \{p\}) \cup \{p \oplus_n 1\} \ .$$

Again, we can even state

$$\forall p < n - 1 : \Box(p \in \mathit{Prop} \rightarrow \bigcirc(p \oplus_n 1 \in \mathit{Prop}))$$

analogously to the token passing example. Now, the IndRing rule allows us to conclude $\forall_n p : \Box\Diamond(p \in \mathit{Token})$.

Chapter 12

Networks with Tree Structures

So far in this work we have analyzed parameterized networks with broadcast communication or communication between neighbors in the underlying topology. As topologies we introduced linear networks or rings. This kind of topology can be easily encoded in the logic WS1S, since there we have, as indicated by its name, the possibility to address the neighbors of a process using the predecessor function -1 and the successor function $+1$.

Fortunately, the decidability of WS1S does not depend on the restriction to one successor. It can be generalized to an arbitrary number of successors. This allows us to model networks that have a tree structure with bounded branching. The methods presented throughout this work can be adapted to handle the generalized case.

12.1 The Logics WSnS

In contrast to WS1S where terms are built up from the constant 0 and first-order variables by applying the successor function $succ(t)$ ("$t+1$"), in WS$\boldsymbol{n}$S we have the constant ϵ and n successor functions $succ_i(t)$ denoting the i-th son of a node in the tree. The n functions over $\{0,\ldots,n-1\}^*$ are $succ_i(w) = wi$ for every $i < n$. Hence, terms are built up from the constant ϵ, variables $x, y, \ldots$, and by application of the n successor functions $succ_i(x)$ denoted by xi.

Similarly to WS1S *atomic formulae* of WS$\boldsymbol{n}$S are of the form b, $t = t'$, $t < t'$, $t \in X$, where b is a boolean variable, t and t' are terms, and X is a set variable (second-order variable). WS1S formulae are built up from atomic formulae by applying the boolean connectives as well as quantification over both first-order and second-order variables.

WS1S formulae are interpreted in models that assign finite sub-sets of

$\{0,\ldots,n-1\}^*$ to second-order variables and elements of $\{0,\ldots,n-1\}$ to first-order variables. The interpretation is defined in the usual way.

Given a WS$\boldsymbol{n}$S formula f, we denote by $[\![f]\!]$ the set of models of f. The set of free variables in f is denoted by $\mathit{free}(f)$.

In the case of $n = 2$ Rabin showed in [Rab69] that even the stronger logic S2S ranging over infinite subsets of $\{0,1\}^*$ is decidable. In [Rab70] Rabin showed that a proper subset of S2S is Büchi recognizable and establishes also a characterization for WS2S defined trees. In [MSS86] Muller, Saoudi, and Schupp give a direct automata-theoretic characterization of WS2S. For an overview see [Tho90].

Although the decidability result of Rabin for (W)S2S can be generalized to the case of full n-ary trees, we concentrate on WS2S in the remainder.

12.2 Adapting the Methods

In the framework of WS1S we introduced a restricted class of parameterized networks called MPS. For MPS we defined a translation to WS1S. We gave heuristics to define an abstraction relation adequate to verify the WS1S translations. In the case of WS2S we follow the same strategy. As indicated earlier we use the logic WS2S to model a network with tree topology. Hence, we introduce communication between neighbors which we express by reading or writing data of a process. This process is addressed by one of the successor functions $succ_1$ or $succ_2$. Therefore, we extend the logic AF used to define MPS.

Let n be a variable. We define the set $\mathrm{AF}_2(n)$ of formulae f analogously to $\mathrm{AF}(n)$:

$$f \ ::= \ b[x] \mid x = x \mid x0 = x \mid x1 = x \mid \neg f \mid f \wedge f \mid \forall_n\, x\!:\! f \mid \exists_n\, x\!:\! f \ ,$$

where again x is a *position variable* that ranges over nodes in a binary tree, b is a *boolean function* that assigns booleans to positions in a full binary tree. Let $m \in \omega$. We denote by Σ_m the set of evaluations s such that $s(n) = m$, $s(x) \in \{0,1\}^m$ and $s(b) : \{0,1\}^m \to \{\mathit{true}, \mathit{false}\}$. Then, formulae in $\mathrm{AF}_2(n)$ are interpreted over evaluations $s \in \bigcup_{m\in\omega} \Sigma_m$ in the usual way.

Now, we define boolean transition systems with tree topology similarly to BTS as done in Definition 2.5.

Definition 12.1 (BTS with tree topology)
A *BTS with binary tree topology* $\mathcal{S}_2(i,n)$ is parameterized by n and i, where n is a variable ranging over natural numbers whereas i is now a position variable indicating positions in a full binary tree of depth n. $\mathcal{S}_2(i,n)$ is defined as the triple $(\mathcal{V}, \Theta, \mathcal{T})$, where

- $\mathcal{V} = \{b_1, \ldots, b_k\}$ and each b_j, $1 \leq j \leq k$, is a boolean function assigning a boolean value to each position in a full binary tree with depth n.
- Θ is a formula in $\mathrm{AF}_2(n)$ with $\mathit{free}(\Theta) \subseteq \mathcal{V} \cup \{i\}$ and which describes the set of initial states.
- $\mathcal{T}$ is a finite set of transitions where each $\tau \in \mathcal{T}$ is given by a formula $\rho_\tau \in \mathrm{AF}_2(n)$ such that $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}' \cup \{i\}$ and $\mathcal{V}'$ contains a primed copy for each variable in $\mathcal{V}$.

A network $\mathcal{P} = \{\mathcal{P}_m \mid m \geq 1\}$ is called *monadic parameterized with tree topology* if it is built from BTS with tree topology $\mathcal{S}_2(i, n)$, where $\mathcal{P}_m$ is illustrated in Figure 12.1. We denote the class according to MPS for linear topologies by MPS_2. □

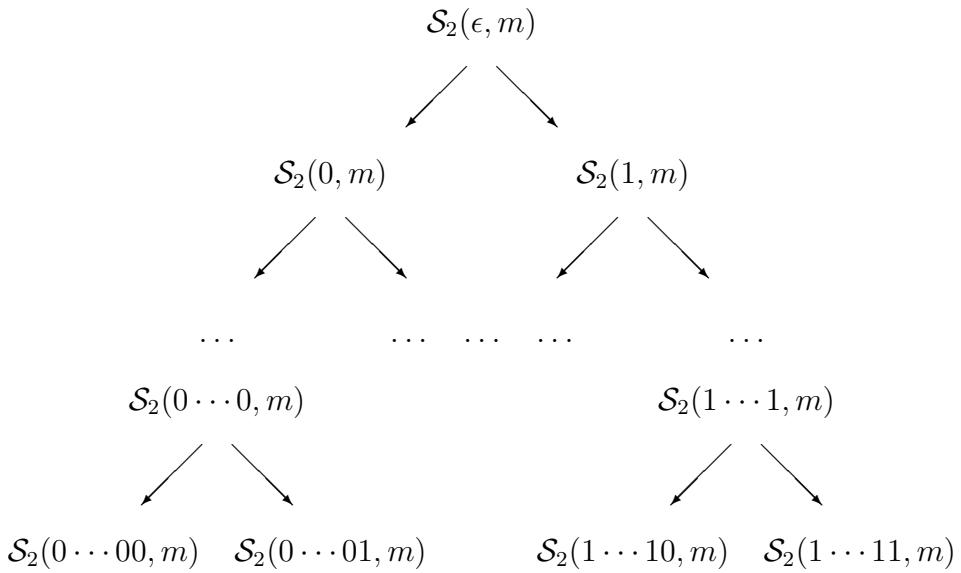


Figure 12.1: Instance $\mathcal{P}(m)$ of an MPS_2

The next step is to translate MPS_2 into WS2S transition systems. The definition of WS2S transition systems is straightforward.

Definition 12.2 (WS2S transition system)
A *WS2S transition system* $\mathcal{S}_2 = (\mathcal{V}, \mathcal{T}, \Theta)$ is given by the following components:

- $\mathcal{V} = \{X_1, \ldots, X_k\}$: A finite set of second order variables where each variable is interpreted as a finite set of positions in a binary tree.

- Θ: A WS2S formula with $\mathit{free}(\Theta) \subseteq \mathcal{V}$ describing the initial condition of the system.
- $\mathcal{T}$: A finite set of transitions where each $\tau \in \mathcal{T}$ is represented as a WS2S formula $\rho_\tau(\mathcal{V}, \mathcal{V}')$, i.e., $\mathit{free}(\rho_\tau) \subseteq \mathcal{V} \cup \mathcal{V}'$. We use the primed version of the variables to denote the post-state.

The computations of $\mathcal{S}_2$ are defined as usual. Again, let $[\![\mathcal{S}_2]\!]$ denote the set of computations of $\mathcal{S}_2$. □

The translation of MPS_2 into WS2S systems follows the same idea applied in Chapter 3. The boolean functions assigning each position in the tree a boolean value is translated into a set variable containing those positions in the tree assigned the value *true*. We omit the formal definition of the translation function and give an example instead.

For a WS2S system the next steps are exactly the same as in the case of WS1S. An abstraction function is defined through predicates characterizing the configuration of the system. The predicates have to be chosen in such a way that their evaluation suffices to simulate the concrete system close enough to establish a desired property.

12.3 Example

Let us consider a simple example of a token marching up the tree structure. The system can be described as MPS_2 with one boolean array variable b stating whether a considered node has the token or not. A node (process) which has the token can hand it over to its father node in the tree:

$$\exists_n i : b[i] \wedge i \neq \epsilon \rightarrow (\exists_n j : (j0 = i \vee j1 = i) \wedge \neg b[i] \wedge b'[j]) \ .$$

The transition predicate shows that we can easily address the father of node unequal to ϵ. We introduce the notation $i\hat{}$ to express the father node of i.

The translation to a WS2S system contains two set variables *Nodes* and *Token*. Initially we start with an arbitrary binary tree such that one node has the token:

$$\begin{aligned} \exists \mathit{Nodes}, \mathit{Token} : \quad & (\forall i : i \in \mathit{Nodes} \rightarrow i\hat{} \in \mathit{Nodes}) \\ \wedge \quad & \mathit{Token} \subseteq \mathit{Nodes} \wedge \exists i : i \in \mathit{Nodes} \wedge \mathit{Token} = \{i\} \end{aligned}$$

The transition predicate looks like:

$$\exists i : i \in \mathit{Token} \wedge (\mathit{Token}' = \mathit{Token} \setminus \{i\} \cup \{i\hat{}\})$$

If we want to establish the safety property that always only one token moves around the tree, we define an abstract variable:

$$\mathit{unique} \equiv \exists i : \mathit{Token} = \{i\}$$

To compute the abstract system we use exactly the same technique described in Section 5.4. Again, we use MONA [KM98, HJJ+96] to decide the WS2S logic. Since we still work with binary abstractions we can interpret the MONA output in the same way to construct the abstract system. Model-checking the constructed system proves that the variable *unique* is constantly *true*. Hence, the described uniqueness of the token is indeed an invariant.

A natural liveness property in our example is whether eventually the root node gets the token. With a minor modification we can apply the technique presented in the previous chapter to prove that property.

Without proof (since it is quite similar to the one of Theorem 11.1) we state the following lemma.

Lemma 12.3 (ROOT rule)
For an arbitrary MPS_2 and an individual property Prop containing those nodes where the property holds, the following proof rule is correct:

$$\frac{\begin{array}{l}1.\ \mathit{Prop} \neq \emptyset \\ 2.\ \Box(p \in \mathit{Prop} \rightarrow \Diamond(p\hat{} \in \mathit{Prop}))\end{array}}{\Diamond(\epsilon \in \mathit{Prop})}$$

□

Choosing a local abstraction focusing on two nodes i and $i\hat{}$ we can establish $\Box(i \in \mathit{Token} \rightarrow \bigcirc(i\hat{} \in \mathit{Token}))$. Premise 1 is also correct since we start with one process that owns the token initially. Hence, we can establish the desired liveness property $\Diamond(\epsilon \in \mathit{Token})$.

For the ROOT rule we can state a dual BROADCAST rule. Assuming messages to be forwarded from a node to its children it states that a message originating from the root node gets eventually broadcasted throughout the whole network.

$$\frac{\begin{array}{l}1.\ \epsilon \in \mathit{Prop} \\ 2.\ \Box(p \in \mathit{Prop} \rightarrow \Diamond(p0 \in \mathit{Prop} \wedge p1 \in \mathit{Prop})\end{array}}{\Diamond(\mathit{Prop} = \mathit{Nodes})}$$

Both rules can be used to verify termination behavior of algorithms communicating in a tree structure. Progress of such algorithms is guaranteed by broadcasting messages initiated by the root node downwards and receiving acknowledgments from the leafs upwards. Those algorithms often base their communication on a spanning tree constructed in a first phase of the algorithm. Several of those algorithms are described in [Lyn96, AW98].

Chapter 13

Conclusion

Parameterized networks and their verification appear naturally when dealing with distributed algorithms. Such distributed algorithms are needed whenever computers communicate with each other. Hence, the world wide web is based on distributed algorithms, but also the electronic control of modern cars or aircrafts is built on communication mechanisms to connect distributed control units.

In this thesis we gave an abstraction-based approach to the verification of parameterized systems. To present a feasible approach to parameterized verification we first restrict ourselves to networks built of finite state processes.

13.1 Dealing With Undecidability

Even for the restricted class of parameterized networks consisting of finite state processes the undecidability result form Apt & Kozen does not allow to search for fully automatic verification techniques. In our approach we need user interaction to find the right abstraction relation. Since we use WS1S to express the abstraction as well as the systems, we can exploit the fact that WS1S is decidable. We are able to construct fully automatically the abstract system which is finite and can be model-checked. The abstract system is defined via abstract boolean variables describing the actual configuration of the concrete system. The user interaction needed to figure out the adequate set of these abstract variables is compensated by an efficient verification cycle; choosing an abstraction relation, model-checking the abstract system to check for desired properties, analyzing the results, and refining the abstraction relation. The method is implemented in our tool-set PAX and requires minimal insight in the techniques applied. In case of false negatives refinement of the abstraction supports the overall understanding of the system.

We achieve the ability to construct the abstract systems automatically by restricting the processes not only to be finite state, but also to have a standard topology, i.e., we expect it to be a linear or ring network, or to communicate in a broadcast manner via shared variables. We call those systems monadic parameterized (MPS). Those systems may act synchronously or asynchronously, in both cases the restrictions allow us to model these classes as WS1S systems. These systems have second order variables ranging over finite subsets of the natural numbers. We use the sets of natural numbers to model sets of processes, identified by their PIDs, such that a set contains those processes that are in certain states. The initial configuration as well as the transition relation are characterized as WS1S predicates. Their are two advantages of modeling MPS as WS1S systems: First, the parameterization gets hidden in the initial condition of the WS1S system, there the existence of an arbitrary n is postulated which characterizes the number of participating processes. Secondly, WS1S is decidable and, moreover, the decision procedure constructs an automaton accepting all satisfying models of a given predicate. This allows to construct the abstract system for given abstract variables automatically. Hence, the abstract system incorporates abstractions for all instances of the original MPS.

We apply the abstract variables chosen by the user to construct a conservative abstraction, i.e., every computation of the concrete system can be simulated by the abstract system. We state the correctness conditions for the systems as future LTL formulae. An LTL formula is satisfied by a system if all its computations satisfy the formula. Since our abstraction over-approximates the computations, a property established for the abstract system also holds for the original one.

13.2 Incorporating Fairness

The verification of future LTL clearly includes liveness properties. Since a formula has to hold for all computations, also validity of formulae expressing liveness can be transferred to the original system when the formulae are satisfied by the abstract one. Nevertheless, it is a well-known obstacle to the verification of liveness properties via abstraction, that the abstract system contains cycles not corresponding to paths in the concrete system. This is not surprising since the main goal of an abstraction relation is to identify and merge states, which may obviously introduce new cycles. A way to overcome this difficulty is to enrich the abstract system with fairness conditions or, more generally, ranking functions over well-founded sets that eliminate undesirable cycles. These are cycles that do not correspond to concrete com-

putations. The main problem is, however, to find such fairness conditions.

The main observation is that the abstract system cannot count the processes in the network. This may allow to apply a single transition infinitely often in the abstract system, although at the concrete level the transition eventually gets disabled. The reason is that there are only finitely many processes in the network. We translate this observation into a fairness condition requiring that whenever infinitely many processes leave some location, infinitely many have to enter it. This idea is generalized by the introduction of ranking predicates. Each ranking predicate introduces two new variables to the abstract system. The variables indicate decrease or increase of a ranking value defined by the predicate. Therefore, a condition that states "infinite decrease implies infinite increase" can be added safely.

For an abstract system with additional fairness conditions, we can also think about lifting fairness requirements from the original system to the abstract one. The correctness of this approach so depends on the kind of liveness, weak or strong, and on the set of chosen abstract variables.

Besides illustrating all modeling and verification steps with numerous examples, we present completeness results for the developed framework. This means, for restricted classes we suggest abstractions and ranking predicates that are sufficient to verify a given class of properties. This shows that decidability for networks expressible in our framework, can also be proven with our approach. The completeness results are mainly based on finding the right ranking predicates.

13.3 Applicability

In contrast to our motivation, the first examples of parameterized systems that we used to illustrate our approach present mutual exclusion algorithms. Such algorithms are needed in operating systems: when introducing multiprocessor or multi-tasking operating systems the problem of concurrent processes arises naturally. The problem is that those concurrent processes share the same global data structures of the operating system. To coordinate access to such critical resources mutual exclusion algorithms are needed. The construction of those algorithms was an important and complex research topic in the seventies. Until now the verification of the developed algorithms is used as a benchmark for new verification approaches. Hence, as running examples we show the verification of Szymanski's algorithm and other mutual exclusion algorithms.

In practice, most networks consist of infinite state processes itself. But, depending on the communication or coordination mechanisms one wants to

analyze for those networks, one can simplify them to make them fit in our framework. Such simplifications are based on reducing the verification problem to the core, neglecting unimportant details of the processes. Some of the simplifications may be obvious or trivial, others have to be established formally. This can be done by theorem proving. We illustrate the approach by proving a DSM protocol to be correct. Another idea is to apply *incremental verification*. One starts with proving some invariants of the system. Then, one simplifies the system by using those invariants to make it fit in our MPS framework. In this framework more complex properties can be established. The correctness prove for the group membership protocol was given as an example for that approach.

13.4 Contribution of this Thesis

We introduce the class of WS1S systems to model parameterized networks. In contrast to [KMM+97] introducing regular expressions as adequate language for symbolic model-checking and [ABJN99] where acceleration techniques are presented to characterize the set of reachable states as a single regular expression, we exploit decidability of WS1S directly to automatically construct abstract systems for given abstractions. In principle, the classes of systems that can be verified are quite similar due to the close connection of WS1S and M2L-Str, the Monadic Second-order Logic on Strings. M2L-Str corresponds exactly to the regular languages and can be emulated efficiently in WS1S.

The drawback of losing precision when using an abstract system to verify the original gives us on the other hand the advantage of efficient verification techniques for liveness properties. We introduce ranking predicates to express progress in computation and show how to construct fairness conditions that can be safely added to the abstract system. The user's preferred model-checker can be applied to verify the fair abstract system. False negatives lead to refinement of the abstract system and increase insight into the original system.

Beside numerous examples illustrating applicability of our methods, we prove the capabilities of our approach by proving the model-checking problem to be decidable for restricted networks that are representable in our framework.

13.5 Future Work

In this work we have focussed on modeling parameterized networks. The parameter fixes the size of the network. In other protocols the parameter is used to define the size of certain data structures. One direction for future research is to find a class of those data structures that are representable in WS1S and to apply our approach in such a setting. Probably, one has to find new heuristics to define the abstraction relation and the ranking predicates.

Another topic of interest is already addressed in this work: how to deal with real life examples in order to make them fit into the WS1S framework. We presented examples where a first abstraction step guided by theorem proving leads to such systems that can be subject to our approach.

List of Theorems

Definitions

Examples

List of Figures

Glossary of Symbols

Chapter 2

$\mathcal{A}$	assertion language	18
Σ	set of states	18
s	state in Σ	18
φ	predicate	18
$\mathcal{S}$	transition system	18
$\mathcal{V}$	set of variables	18
$\mathcal{L}$	set of control locations	18
$\mathcal{T}$	set of transitions	18
τ	transition in $\mathcal{T}$	18
Θ	initial condition	18
ℓ_i	control location in $\mathcal{L}$	19
$\rightarrow_\tau$	transition relation	19
ρ_τ	transition predicate	19
$\sigma : s_0, s_1, \ldots$	computations	19
$[\![\mathcal{S}]\!]$	set of all computations of $\mathcal{S}$	19
$\mathcal{S}_1 \otimes \mathcal{S}_2$	asynchronous product of $\mathcal{S}_1$ and $\mathcal{S}_2$	20
$\mathcal{S}_1 \| \mathcal{S}_2$	parallel execution of $\mathcal{S}_1$ and $\mathcal{S}_2$	20
$\mathcal{S}^{\mathcal{F}}$	fair transition system	21
$\mathcal{J}$	set of just (weak fair) transitions	21
$\mathcal{C}$	set of compassionate (strong fair) transitions	21
$\mathcal{F}$	single fairness condition	21
$\mathcal{P}$	parameterized network $\{\mathcal{S}(1,n) \| \cdots \| \mathcal{S}(n,n) \mid n \in \omega\}$	23
$\mathcal{P}(m), \mathcal{S}^m$	instance of network with m processes	23
AF(n)	restricted parameterized first-order fragment of WS1S	24
$\forall_n x : f$, $\exists_n x : f$	abbreviates $\forall x : x < n \rightarrow f$, $\exists x : x < n \wedge f$	24

$(\mathcal{L}, \mathcal{T}, \mathcal{I})$	labeled transition graph	28
$\mathcal{I}$	set of initial states	28
$\mathcal{S}_1 \oplus \mathcal{S}_2$	synchronous product of $\mathcal{S}_1$ and $\mathcal{S}_2$	31
$\langle C, U \rangle$	network with control $\mathcal{P} = \{C \| U^m \mid m > 1\}$	32

Chapter 3

$\mathcal{W}$	WS1S transition system	38
tr	translation of AF(n)-formulae into WS1S	39
$\forall_P : f$, $\exists_P x : f$	abbreviates $\forall x : x \in P \rightarrow f$, $\exists x : x \in P \wedge f$	39
Tr	translation of MPS into WS1S systems	40
h	bisimulation between $\mathcal{P}(m)$ and $Tr^*(\mathcal{P})$	41

Chapter 4

P	traceproperty	51
$\bigcirc, \mathcal{U}$	LTL operators next and until	52
$\Box$, $\Diamond$	LTL abbreviations for always and eventually	52
α	abstraction relation	55
$\mathcal{S}_A = (\mathcal{V}_A, \Theta_A, \mathcal{T}_A)$	abstract transition system	55
$\mathcal{S} \sqsubseteq_\alpha \mathcal{S}_A$	$\mathcal{S}_A$ is an abstraction of $\mathcal{S}$ w.r.t. α	55
$\mathcal{S} \sqsubseteq_\alpha^{\mathcal{F}} \mathcal{S}_A$	$\mathcal{S}_A$ is a fair abstraction of $\mathcal{S}$ w.r.t. α	56

Chapter 5

$\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A)$	WS1S formula expressing the abstraction α	59
$\widehat{\alpha}(\mathcal{V}, \mathcal{V}_A, i, j)$	local abstraction focussing on processes i, j	60
$\equiv$	defines abstract variables	62
b_φ	predicate abstraction for φ	62
p_X	abstract variable expressing $p \in X$	62

Chapter 7

$\chi(i, X_1, \cdots, X_k)$	ranking predicate	87
ζ_χ	ranking value for χ	87
$+_\chi$, $-_\chi$	abstract variables indicating in- and decrease of ζ_χ	87
$\mathcal{W}^{\mathcal{F}}$	fair WS1S transition system	81
E_τ, T_τ	additional set variables to model fair transitions	83
e_τ^p, t_τ^p	abstract enabledness and taken variables for p	85
e_τ, t_τ	abstract variables to characterize fairness for τ	85

Bibliography

[AAB+99] P.A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [HP99], pages 146–159.

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In Halbwachs and Peled [HP99], pages 134–145.

[AFdR80] K.R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In Brinksma and Larsen [BL02].

[AK86] K. Apt and D. Kozen. Limits for Automatic Verification of Finit-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 1986.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, October 1981.

[Apt85] K.R. Apt, editor. *Logics and Models of Concurrent Systems*, volume 13 of *NATO Advanced Science Institutes Series F: Computer and System Sciences*. Springer-Verlag, 1985.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AW98] H. Attiya and J. Welch. *Distributed Computing*. Mc Graw Hill, 1998.

[Bau98] Kai Baukus. Generation of auxiliary invariants. Diploma thesis, CAU Kiel, 1998.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In Graf and Schwartzbach [GS00], pages 188 – 203.

[BBLS01] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Networks of processes with parameterized state space. In Bouajjani [Bou01], pages 388–402.

[BCF01] G. Berry, H. Comon, and A. Finkel, editors. *Proceedings of the 13th International Conference on Computer-Aided Verification, Paris, France*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[BCL+96] Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors. *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95*, volume 1019 of *Lecture Notes in Computer Science*, Aarhus, Denmark, May 1996. Springer.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla [ES00].

[BL02] E. Brinksma and K.G. Larsen, editors. *Proceedings of the 14th International Conference on Computer-Aided Verification, Copenhagen, Denmark*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[BLO98a] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In Hu and Vardi [HV98], pages 319–331.

[BLO98b] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In Hu and Vardi [HV98], pages 505–510.

[BLS00] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. Technical Report TR-ST-00-4, CAU Kiel, 2000.

[BLS01] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.

[BM01] R. E. Bryant and C. Meinel. *Logic Synthesis and Verification*, chapter Ordered Binary Decision Diagrams. Kluwer Academic Publishers, 2001.

[BM02] A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In Damm and Olderog [DO02].

[Bou01] A. Bouajjani, editor. *ICALP 2001 Satellite Workshop on Verification of Parameterized Systems, VEPAS 2001*, volume 50 of *Electronic Notes in Computer Science*, Crete, Greece, 2001. Elsevier.

[Bry85] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *22nd Design Automation Conference*, pages 688–694, June, 1985.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Büc60] J.R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1), July 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM symp. of Prog. Lang.*, pages 238–252. ACM Press, 1977.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In Halbwachs and Peled [HP99], pages 495–499.

[CCK+02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and Dong Wang. Refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD'02)*, 2002.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of Synchronisation Skeletons for Branching Time Temporal Logic. In Kozen [Koz81].

[CGJ95] E. Clarke, O. Grumberg, and S. Jha. Verifying Parameterized Networks using Abstraction and Regular Languages. In I. Lee and S.A. Smolka, editors, *Proceedings of CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example guided abstraction refinement. In Emerson and Sistla [ES00].

[CGKS02] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In Brinksma and Larsen [BL02].

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technical University of Eindhoven, 1996.

[DFN01] M. Duflot, L. Fribourg, and U. Nilsson. Unavoidable configurations of parameterized rings of processes. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 472–486. Springer-Verlag, 2001.

[DGG94] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In Olderog [Old94].

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In Berry et al. [BCF01].

[DO02] W. Damm and E.-R. Olderog, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, Oldenburg, 2002. Springer-Verlag.

[dR85] W.-P. de Roever. The cooperation test: a syntax-oriented verification method. In Apt [Apt85].

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, November 2001.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data Refinement*. Cambridge University Press, 1998.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In Emerson and Sistla [ES00].

[Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *22nd ACM Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur, editor, *Proceedings of CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 87–98, 1996.

[ES00] E. A. Emerson and A. P. Sistla, editors. *CAV '00, Proceedings of the 12th International Conference on Computer-Aided Verification, Chicago IL*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[FG98] K. Fisler and C. Girault. Modelling and Model Checking a Distributed Shared Memory Consistency Protocol. In *ICATPN'98*. Springer, 1998.

[FLBB79] M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254, San Juan, Puerto Rico, October 1979. IEEE.

[FLBB89] M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[FO97a] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 213–227. Springer-Verlag, 1997.

[FO97b] L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. In Infinity'97 [Inf97].

[Gle96] J. Gleick. A bug and a crash. *New York Times Magazine*, December 1996.

[Gri77] D. Gries. An exercise in proving parallel programs correct. *CACM*, 20(12):921–930, 1977.

[Gru97] Orna Grumberg, editor. *CAV '97, Proceedings of the 9th International Conference on Computer-Aided Verification, Haifa. Israel*, volume 1254 of *Lecture Notes in Computer Science*. Springer, June 1997.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In Grumberg [Gru97].

[GS00] S. Graf and M. Schwartzbach, editors. *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[HJ89] C. A. R. Hoare and Cliff B. Jones, editors. *Essays in Computing Science*. Prentice Hall, 1989.

[HJJ+96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic Second-Order Logic in Practice. In Brinksma et al. [BCL+96].

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 22(6/7), 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [HJ89].

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol99] G. Holzmann. Spin. http://netlib.bell-labs.com/netlib/spin, 1999.

[HP99] N. Halbwachs and D. Peled, editors. *Proceedings of the 11th International Conference on Computer-Aided Verification, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98, 10th International Conference, Vancouver, BC, Canada, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[Inf97] *2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronic Notes in Theoretical Computer Science*, Bologna, Italy, 1997. Elsevier Science.

[JL98] H.E. Jensen and N.A. Lynch. A proof of Burns n-process mutual exclusion algorithm using abstraction. In *TACAS '98*, pages 409–423, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In Graf and Schwartzbach [GS00].

[Kel95] P. Kelb. *Abstraktionstechniken für Automatische Verifikationsmethoden*. PhD thesis, University of Oldenburg, 1995.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[KLR97] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 155–169, Saarbrücken Germany, September 1997. Springer-Verlag.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *ACM Symp. on Principles of Distributed Computing, Canada*, pages 239–247, Edmonton, Alberta, 1989.

[KM98] N. Klarlund and A. Møller. MONA Version 1.3 User Manual. BRICS, 1998.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In Grumberg [Gru97], pages 424–435.

[Kna28] B. Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Société Polonaise de Mathématique*, 6:133–134, 1928.

[Kos82] S.R. Kosaraju. Decidability of reachability in vector addition systems. In *ACM Symposium on Theory of Computing*, pages 267–281. ACM, New York, 1982.

[Koz81] D. Kozen, editor. *Workshop on Logic of Programs 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[Kya01] M. Kyas. Verifying a network invariant for all configurations of the futurebus+ cache coherence protocol. In Bouajjani [Bou01].

[Lam76] L. Lamport. Formal correctness proofs for multiprocess algorithms. In *Proc. of the 2ème Colloque International sur la programmation*, Paris, April 1976.

[LBOB01] Y. Lakhnech, S. Bensalem, S. Owre, and S. Berezin. Incremental verification by abstraction. In Margaria and Yi [MY01].

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97*, Paris, 1997.

[Lon93] D. E. Long. *"Model Checking, Abstraction, and Compositional Reasoning"*. PhD thesis, Carnegie Mellon, 1993.

[LP89] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th ACM Annual Symposium on Principles of Programming Languages*, pages 97–107. ACM, New York, 1989.

[LS97] D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction. In Infinity'97 [Inf97].

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[Mai01] M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In Berry et al. [BCF01].

[May81] E. Mayr. An algorithm for the general Petri net reachability problem. In *ACM Symposium on Theory of Computing*, pages 238–246. ACM, New York, 1981.

[McM92] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, CMU Tech Rpt. CMU-CS-92-131, 1992.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM99a] K.L. McMillan. Circular compositional reasoning about liveness. Technical report, Cadence Berkeley Labs, 1999.

[McM99b] K.L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. Technical report, Cadence Berkeley Labs, 1999.

[Meh95] W. Mehl. Siemens-Rechner legt Stellwerk in Hamburg für zwei Tage lahm. *Computerwoche*, 12:27, 1995.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1981.

[MP91] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. Rashid, editor, *Carnegie Mellon Computer Science: A 25-year Commemorative*, pages 125–159. ACM Press and Addison-Wesley, 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[MP95a] Z. Manna and A. Pnueli. *Specification and Validation Methods*, chapter Verification of Parameterized Programs, pages 167–230. Oxford University Press, 1995.

[MP95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems, Safety*. Springer Verlag, 1995.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree, and its complexity. In L. Kott, editor, *Internat. Coll. on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 275–283. Springer-Verlag, 1986.

[MY01] Tiziana Margaria and Wang Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Genova, Italy, April 2-6, 2001*, volume 2031 of *Lecture Notes in Computer Science*. Springer, 2001.

[Nam98] K. Namjoshi. *Ameliorating the State Explosion Problem*. PhD thesis, The University of Texas at Austin, 1998.

[NDOG86] V. Nguyen, A. Demers, S. Owicki, and D. Gries. A modal and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, 1986.

[NN99] T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In *FASE'99*, number 1577 in Lecture Notes in Computer Science, pages 188–203. Springer-Verlag, 1999.

[NN01] T. Nipkow and L. Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *FMPPTA 2001*, 2001.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–286, May 1976.

[Old94] E.-R. Olderog, editor. *Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy*. North-Holland, 1994.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Owi75] S. Owicki. *Axiomatic proof techniques for parallel programs*. Report tr 75-251, Cornell University, Ithaca N.Y., Department of Computer Science, 1975.

[Owi76] S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In *Proc. of the 8th ACM Symposium on the Theory of Computing (STOC '76), Hershey, Pennsylvania, May 3–5, 1976*, pages 73–86. ACM, 1976.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In Margaria and Yi [MY01].

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In Emerson and Sistla [ES00].

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In Brinksma and Larsen [BL02].

[PZ01] A. Pnueli and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Berry et al. [BCF01].

[QS81] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, 1981.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. In Y. Bar-Hillel, editor, *Mathematical Logic and Foundations of Set Theory*, pages 1–23. North-Holland, 1970.

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In Damm and Olderog [DO02].

[SC85] A.P. Sistla and E.M. Clarke. The complexity of linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[SG89] Z. Stadler and O. Grumberg. Network grammars, communication behaviours and automatic verification. In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, pages 151–165, Grenoble, France, 1989. Springer Verlag.

[Sti95] A. Stiller. Der Pentium und seine Fehler. *c't*, 7:186, July 1995.

[Szy88] B.K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of International Conference on Supercomputing Systems 1988*, pages 621–626, St. Malo, France, July 1988.

[Szy90] B.K. Szymanski. Mutual exclusion revisited. In *Fifth Jerusalem Conference on Information Technology*, pages 110–117, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific J. of Math.*, 5:285–309, 1955.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 134–191. Elsevier Science Publishers B. V., 1990.

[Uri99] T.E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1999.

[VW86] M. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *1st IEEE Symposium on Logic in Computer Science*. IEEE, New York, 1986.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants (extended abstract). In Sifakis, editor, *Workshop on Computer Aided Verification*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, 1989.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Thirteenth POPL (St. Peterburg Beach, FL)*, pages 184–193. ACM, January 1986.

[Wol92] H. Wolpe. Patriot missile software problem. Technical Report GAO/IMTEC-92-26, United States General Accounting Office, 1992.