Exploring the Limits of Parameterized System Verification

Dissertation

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

Karsten Stahl

Kiel
2003

1. Gutachter	Prof. Dr. Yassine Lakhnech
2. Gutachter	Prof. Dr. Willem-Paul de Roever
3. Gutachter	Prof. Dr. Thomas Wilke
Datum der mündlichen Prüfung	25. Juni 2003

Contents

Chapter 1

Introduction

This thesis addresses the problem of verification of parameterized systems. In particular, this thesis focuses on (un)decidability results concerning these systems, discusses a verification method and tool (PAX) for these systems (based on abstraction and model-checking), and illustrates these using an extensive example.

A parameterized system is given by a system description containing some parameters that can be instantiated by different values, such that different parameter instantiations may lead to different finite-state systems.

Parameterized systems arise in many areas, for instance data base applications which are expected to work for any number of users accessing the data base, web applications, communication protocols, operating systems, or hardware cache-coherence protocols. All these examples have in common that correctness of the system should not depend on the parameters, hence, every system instantiation is required to be correct.

Many of these systems and algorithms are safety-critical, and, therefore, correctness is an important issue. For small system instances, existing techniques can be used for verification and debugging. Nevertheless, even if a number of instances is checked, the question remains whether the system is correct for *all* parameter instantiations.

An abstract description of the verification problem for parameterized systems can be stated as follows:

> Given a system $\mathcal{S}(i_1, \ldots, i_k)$ with k parameters $i_1, \ldots, i_k$, and a property ϕ, prove for all possible values $n_1, \ldots n_k$ whether $\mathcal{S}(n_1, \ldots, n_k)$ satisfies property ϕ.

The parameters are not bounded in general.

Hence, a parameterized system describes a *family of systems*, which is infinite in general. The members of the families are also called system in-

stances. The verification problem for parameterized systems results in proving correctness *for every* system instance. Consequently, one has to prove a property for infinitely many system instances.

1.1 Modeling and Decidability Results

It is known that the general verification problem for parameterized systems is undecidable [AK86, Suz88].

On the other hand, verification of several strongly restricted classes of parameterized systems are known to be decidable. Results of this type are given in [GS92, EN95, EN96, EK00, Mai01].

Inspired by the discrepancy of general undecidability on the one hand, and decidability results for different restricted classes on the other, we focus on investigating the *boundary* of decidability in this area.

We approach this question by presenting a uniform general framework for modeling parameterized systems, and by investigating the verification problem for a number of derived subclasses. The general framework allows to represent many of the known decidable system classes. By defining a number of constraints on this general model, a classification for parameterized systems is derived.

A general framework for verification of parameterized systems must provide:

- a formal definition for parameterized systems $\mathcal{S}(i_1, \ldots, i_k)$,
- a domain for the parameters $i_1, \ldots, i_k$,
- a formal language to express system properties, and
- a formalization of the satisfiability notion.

The parameter domain is given by the natural numbers $\mathbb{N}_{\geq 1}$. This choice of the domain was made in all the work mentioned previously, although, in many of them not all the properties of the natural numbers are used, such that, in principle, this domain can often be substituted by simple unordered finite sets of identifiers.

Our framework is based on a finite set of finite state variables and array variables. The systems are parameterized by the size of the array variables. Predicates are used to describe initial states and transition relations, where quantification over array indices can be used. The semantics of a system instance is given by the set of its maximal traces. Properties are expressed

in linear time temporal logic, where additionally universal quantification over array indices is allowed.

The choice of this framework is natural; it is a classical model of a transition system based on variables used, for example, by Manna and Pnueli in [MP95] and also in [APR+01, Mai01, PRZ01] as a computation model for parameterized systems.

Verification for the general system class is undecidable. Hence, we impose a number of independent constraints on systems to obtain a large number of parameterized system classes, among them systems which correspond to a parameterized number of parallel user processes connected to some finite state controller. The semantic model of these processes can be restricted to either asynchronous or synchronous execution, using simple syntactic constraints.

In particular, constraints are given upon the usage of index terms which appear in expressions containing arrays, the type and number of quantifiers, and the usage of index term comparisons. These restrictions apply to the initial state predicate and to transition relation predicates.

By imposing these constraints on the general model we also obtain system classes corresponding to decidable classes known from literature, or extending them.

1.2 Verification by Abstraction

As already mentioned, the general verification problem for parameterized systems is undecidable. Hence, a complete and sound verification method for the general system class does not exist. Nevertheless, one can develop an incomplete and sound verification method and hope that it can be applied to systems of interest.

We present a sound but necessarily incomplete verification method based on abstraction which can be applied to parameterized systems not belonging to decidable classes. This method is, however, not fully automatic and needs user interaction. We show how to verify both safety and liveness properties expressed in linear time temporal logic (LTL), and discuss how this is implemented in our tool called PAX.

Verification by abstraction is a popular verification approach applied in many areas. The general idea is to verify a more abstract system which is easier to verify than the original concrete one. If the abstract system is finite, for example, model-checking can be used to establish properties about it.

In our context of universally path-quantified properties, abstraction means that abstract systems have "more" behavior than the original one. Intu-

itively, if one over-approximates the system behavior and all the traces of the over-approximation satisfy a given property, one can conclude that also the original concrete system satisfies this property.

On the other hand, in case verification of the abstract system fails, this does not imply that also the concrete system does not satisfy the property, since some of the "added" traces may violate the property. Counterexamples which are introduced by the abstraction process and do not have a concrete counterpart are called *false negatives*

In case verification of the abstract system gives rise to a counterexample, our verification method requires user interaction. It is not clear whether the abstract trace corresponds to any behavior of the concrete system, due to the over-approximation. If the user is able to find a concrete counterpart for the abstract counterexample, obviously the concrete system does not satisfy the property and verification ends with a negative result. Otherwise, the user has to refine the abstract system and can again try to verify it.

Giving an abstract system and an abstraction relation, and showing validity of the abstraction relation are tedious tasks. Moreover, during the verification process the abstract system is usually modified several times, and each time the abstraction relation has to be established anew.

Therefore, we follow the approach to give an abstraction relation that relates concrete with abstract states and *compute the abstraction* automatically instead of giving the abstract system and establishing the abstraction relation.

The user interaction then reduces to refining the abstraction relation if a counterexample for the abstract system is found. The construction of the abstract system and its verification can be done automatically.

However, it is not obvious how to use the verification-by-abstraction approach for parameterized systems, since a parameterized system corresponds to a *family of systems*, and each member of the set should be verified, whereas the notion of abstraction refers only to one system. Our idea is to model a parameterized system as one *higher-order* system and compute finite abstractions of this system to be able to use existing model-checking techniques to verify properties of the abstractions. Our verification method is based on the following:

- We model parameterized systems as higher-order transition systems in the weak monadic second-order logic of one successor, WS1S in short. These systems have variables ranging over finite sets of natural numbers. The logic WS1S is known to be decidable.

- Given an abstraction relation, the decidability of WS1S allows one to

automatically compute a finite abstraction of the whole system. This finite abstraction will be an abstraction for every instance of the system.

- Model-checking techniques are then used to establish properties valid for all system instances.

Unfortunately, this approach is hardly applicable for liveness properties. Intuitively, liveness properties express that the system eventually reaches a state which is considered to be "good". The abstraction process basically merges concrete states. Therefore, if one concrete transition leaves a state and enters some other, and both are merged, a cycle is introduced in the abstraction. If such a cycle is taken forever, the good states will never be reached, which prevents liveness verification. In fact, abstraction usually introduces a lot of cycles, and these cycles invalidate most liveness properties. Therefore, we extend the method to be able to prove liveness properties as well.

To extend the method for liveness verification we compute fairness constraints of the concrete system which are satisfied by every system instance. The constructed abstract system corresponds closely to the concrete one. Hence, we are able to lift the constraints valid for every concrete system instance to the abstract level. The computed constraints rule out the infinite execution of cycles introduced by the abstraction, thus allowing to verify liveness properties of the abstract system.

1.3 Contributions

In this thesis we present a framework for modeling parameterized systems and for deriving a classification of subclasses. This classification is used to investigate the boundary between decidable and a number of undecidable parameterized system classes.

Moreover, we give a sound (but necessarily incomplete) verification method for parameterized systems based on abstraction. This method can be used to verify liveness properties as well, by deriving fairness constraints from the concrete system.

The method is implemented in a tool set we've called PAX. These tools can be used to automatically compute abstractions of WS1S systems, to translate these constructed abstract systems to input languages of existing model-checkers, and to compute fairness constraints which can be used to enrich abstract systems, allowing liveness verification.

The PAX tool is based on decision procedures for the WS1S logic, implemented in the MONA [HJJ+96] tool. Basically, the PAX tool gives MONA

several WS1S predicates characterizing abstract transitions, and MONA computes an automaton representing the set of all models of these predicates. The abstract transitions can be extracted from this automaton.

We demonstrate applicability of the verification method by presenting a case study of a cache-coherence protocol, and proving both its safety and its liveness properties.

1.4 Structure of the Thesis

After defining some basic notions in Chapter 2, we present our parameterized system classification in Chapter 3, defining formally a large variety of different system classes. All these classes are based on the underlying model of systems with a finite number of array variables of parameterized size, and some finite state variables. The system classification covers both synchronous execution models of a parameterized number of processes and asynchronous execution models. We restrict the general model in such a way that also a finite state controller can be connected both synchronously or asynchronously to these processes.

These classes are investigated in Chapter 4. For several system classes we prove or disprove decidability of the verification problem for different properties.

Chapter 5 presents our verification method for parameterized systems based on abstraction. We focus on the system classes for which the method is applicable. Chapter 6 discusses some extensions and problems for liveness verification based on abstraction.

The verification method is implemented in PAX. This tool is presented in Chapter 7. Especially, we present heuristics and methods which allow one to progress with the verification even when the computational power of the running computer is exceeded.

We end with a case study of a cache-coherence protocol in Chapter 8, showing the applicability of our method, and then present some conclusions.

1.5 Related Work

As observed already, it is known that the general verification problem for parameterized systems is undecidable [AK86, Suz88]. The undecidability proof of this result is based on the idea to use system instance $\mathcal{S}(n)$ to simulate n steps of a Turing machine. If the Turing machine halts, an instance in the size of the number of steps of a halting computation is able to reach

a state corresponding to the halting configuration. Hence, the non-halting problem for Turing machines can be reduced to the question whether every instance $\mathcal{S}(n)$ never reaches a state corresponding to a halting configuration.

There are two possible ways in which to proceed: one can either consider subproblems, i.e., one can investigate the verification problem for restricted classes where verification is decidable, or one can devise sound but necessarily incomplete methods and hope that the method is applicable to the systems of interest.

Of the work presenting decision procedures for restricted system families we discuss the following.

The verification problem for systems consisting of a control process and a parameterized number of identical user processes is investigated in [GS92]. Processes use synchronization labels to communicate, and no other kind of communication is possible. It is shown that linear time properties over the controller are decidable. Moreover, it is shown that properties which refer to the controller and which universally quantify over processes are also decidable by reducing these types of properties to controller properties.

Decidability of a class of synchronous systems consisting of a finite state controller and finite state user processes is shown in [EN95, EN96]. Processes in this class are allowed to refer to the state of the controller and user processes in a uniform way. Hence, a transition guard is able to express that some process is in some local state and that simultaneously all processes are in another set of local states. For this system class, algorithmic methods are presented for both safety and liveness properties.

A system class consisting of a parameterized number of asynchronously proceeding user processes from several process classes is investigated in [EK00]. It is shown that the verification problem can be reduced to the verification of system instances of sizes up to a computable bound k.

There are also approaches leading to decision procedures for restricted classes and to incomplete verification methods.

One of these approaches for model-checking safety properties is presented in [Mai01]. Starting from a state predicate describing the "bad states", a proof tree construction is given which performs a backward search using a (backwards) predicate transformer. By giving sufficient conditions for termination of this algorithm, classes of parameterized systems are characterized for which the verification problem is decidable.

In [BD02a, BD02b] verification techniques are investigated which can be used to verify systems which are parameterized in several dimensions. These papers approach the problem by combining multiset rewriting with constraints and the theory of well-quasi orderings. New classes of parameter-

ized systems with unbounded local data are presented, for which verification of safety properties is decidable. Moreover, automated abstractions are given which can be used for systems outside the decidable fragment. Especially, verification of broadcast protocols is investigated in [BD02a].

Broadcast protocol verification is also addressed in [EFM99]. The approach used in these papers falls under the paradigm of using well-ordered sets for the verification of infinite-state systems [AČJT96]. All sets considered are upward closed with respect to some well-order, and guards are restricted to characterize this type of upward closed set. It is shown that liveness properties are undecidable for this system class.

A considerable amount of related work essentially tries to find incomplete verification methods. Among this work is the following.

The first line of research addresses the verification problem by the use of *network invariants*, which can be considered as a form of structural induction to reason about systems with an unboundedly large number of identical components. The first methods based on this approach were [KM89, WL89]. This method requires the invariant to be defined and relies on proof obligations which correspond to the base case and to the inductive step. This approach was extended by the use of network grammars and a method to define invariants in [CGJ95]. Characterizations of system classes for which invariants can be found were given in [Sis97]. The work presented in [BCG89, SG89, HLR92, LHR97] is also based on finding network invariants which have to be found during the verification process and which abstract an arbitrary number of processes.

In [CR99a, CR99b, CR00], the notion of network invariants is combined with a generalization of the *data independence* technique [Wol86]. This enhances the applicability of the verification method.

A group membership algorithm of a time-triggered protocol called TTP/C is automatically verified in [BM02], combining a non-trivial abstraction approach to unbounded (parametric) counter automata with symbolic reachability analysis.

Regular model-checking is another verification approach. In [KMM+97] regular languages are used in a semi-automatic method to represent sets of states of parameterized systems. Intuitively, a word represents a process configuration such that each symbol of the word refers to the local state of one process. Hence, a regular language can be used to represent a set of linear configurations of different system instances.

Finite-state transducers are used to compute predecessors of these representations. Unfortunately, the analysis procedure diverges in many cases. In [ABJN99, JN00] acceleration techniques are applied to consider the effect

of taking a transition arbitrary often and obtaining termination of the algorithmic method. System classes are characterized for which the acceleration techniques can be applied.

In [PS00] acceleration techniques are proposed to compute transitions described in WS1S which correspond to iterations of a system transition. For simple examples the exact set of reachable states can be computed. An incomplete but fully automatic method for proving invariance properties is presented in [APR+01, PRZ01]. Model-checking is applied on small instances to compute candidates for invariant assertions. Deductive methods are used to check whether they are inductive and can be used to prove the property. The deductiveness of these candidates is also checked using symbolic model-checking techniques. In [PXZ02] an abstraction based verification method similar to ours is proposed. Additional fairness constraints are derived to allow liveness verification.

1.6 Bibliographic Notes

Some parts of this thesis have been published already. In particular, the verification method for parameterized systems based on abstraction and its extension to liveness explained in Chapter 5 and Chapter 6 are published in [BLS00a, BLS00b]. The case study from Chapter 8 was published in [BLS02]. Certain ambiguities in these publications are clarified. Moreover, several ideas of [SBLS99], especially the incremental verification approach, influenced the verification method in Chapter 5. The PAX tool explained in Chapter 7 has been presented at the FM-Tools 2000 workshop at Reisenburg castle. Chapter 3 and Chapter 4 (the main chapters of this thesis) are entirely the work of the current author, as are Chapter 7 and Chapter 8.

Acknowledgments

I thank Yassine Lakhnech and Willem-Paul de Roever for many things. The lectures they gave introduced me to the field of verification and to concurrent systems. Yassine supervised this thesis and taught me a lot. His clear vision on how to simplify complicated issues was always very enlightening. Willem-Paul gave me the opportunity to join his group as an assistant and provided me with every imaginable support.

I thank Kai Baukus for his fruitful collaboration over the last years which led to the joint publication of several papers.

Moreover, I wish to thank all my colleagues for a nice working atmosphere.

Especially, I thank Martin Steffen for sharing his encyclopedic knowledge with me and for taking over lecturing obligations from time to time, Ben Lukoschus for all his good advices on LaTeX problems, and Ralf Huuck for his frequent laughter. They, together with Kai, also deserve my gratitude for proof reading this thesis.

I thank Änne Straßner for all the help concerning official bureaucratic topics, and backing me up when necessary.

I wish to thank my parents for their continuous support during my studies.

Above all, I thank my wife Tanja and my children Merle and Yannik. Without their invaluable encouragement and support finishing this work would not have been possible.

Chapter 2

Preliminaries

2.1 Notational Conventions

Let us first define some very basic notions.

- For sets A and B, we denote the set of all functions from A to B by $[A \to B]$. For $A' \subseteq A$ and a function $f : A \to B$, $f|_{A'}$ denotes the usual restriction of f on A'.
- By $\mathbb{N}$ we denote the set of the natural numbers including 0.
- For $m, n \in \mathbb{N}$, let $[m \ldots n]$ be the set $\{j \mid j \in \mathbb{N}, m \leq j \leq n\}$.
- For $n \in \mathbb{N}$, the set $[0 \ldots (n-1)]$ is abbreviated by $[n]$. $[0]$ is the empty set.
- For a set A and $n \geq 1$, A^n denotes the Cartesian product $\times_{i=1}^{n} A$.

Definition 2.1 (Sequence)
Let A be a set. A *sequence* over A is a function from a $\mathbb{N}_{<n}$ or from $\mathbb{N}$ to A.

- $A^{\mathcal{F}}$ is the set of all finite sequences over A, namely $\bigcup_{n \in \mathbb{N}} [\mathbb{N}_{<n} \to A]$. The empty sequence is denoted by ε.
- A^{ω} denotes the set of all infinite sequences over A, namely the set of all functions $[\mathbb{N} \to A]$.
- $A^{\underline{\omega}} \stackrel{\text{def}}{=} A^{\mathcal{F}} \cup A^{\omega}$ is the set of all finite and infinite sequences over A.

□

Definition 2.2 (Sequence length)
For a sequence $\alpha \in A^{\underline{\omega}}$, let $|\alpha|$ be the length of the sequence, i.e., $|\alpha| = \mathbb{N}$ if α is infinite, and $|\alpha| = k$ if $\alpha \in [\mathbb{N}_{<k} \to A]$. We define the abbreviations $\text{last}(\alpha) \stackrel{\text{def}}{=} \alpha(|\alpha| - 1)$ and $\text{first}(\alpha) \stackrel{\text{def}}{=} \alpha(0)$. □

Definition 2.3 (Prefix, postfix, subsequence)
For a sequence $\alpha \in A^{\underline{\omega}}$, and $k < |\alpha|$, let

$$\alpha|^k$$

be the postfix of α starting at position k. Formally speaking, it is the sequence $\beta \in A^{\underline{\omega}}$ with length $|\alpha| - k$ if α is finite and length $\mathbb{N}$ otherwise, such that $\beta(j) = \alpha(j + k)$ (for $0 \leq j < |\alpha| - k$).

For $k < |\alpha|$ let

$$\alpha|_k \stackrel{\text{def}}{=} \alpha|_{[k]} = \alpha|_{[0\ldots k-1]}$$

be the prefix of the first k sequence members of α, from position 0 to $k - 1$ (excluding position k).

Define for $m \leq n < |\alpha|$

$$\alpha[m \ldots n] \stackrel{\text{def}}{=} (\alpha|^m)|_{n-m+1} \ ,$$

which is the subsequence of α from position m to position n, including both limits. □

Remark 2.4
Let $\alpha \in A^{\underline{\omega}}$ and $k \in \mathbb{N}$.

- If α is finite and $k < |\alpha|$, then $|(\alpha|^k)| = |\alpha| - k$.
- $\alpha|^0 = \alpha$.
- $\alpha|^m(n - m) = \alpha((n - m) + m) = \alpha(n)$ for $m \leq n < |\alpha|$.

♣

Definition 2.5 (Function variant)
For a function $f : A \to B$, elements $a \in A$ and $b \in B$, the variant of f which maps a to b is defined by

$$(f : a \mapsto b)(x) \stackrel{\text{def}}{=} \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

□

Definition 2.6 (Concatenation)
For $a \in A, \alpha \in A^{\underline{\omega}}$ define the *concatenation* of a and α by

$$a \cdot \alpha \stackrel{\text{def}}{=} \{(0, a)\} \cup \{(n+1, x) \mid (n, x) \in \alpha\} \ .$$

For $\alpha \in A^{\mathcal{F}}, \beta \in A^{\underline{\omega}}$ define the *concatenation* of α and β inductively by

$$\alpha \cdot \beta \stackrel{\text{def}}{=} \begin{cases} \beta & \text{if } \alpha = \varepsilon \\ \alpha|_{|\alpha|-1} \cdot (\text{last}(\alpha) \cdot \beta) & \text{if } \alpha \neq \varepsilon \end{cases}$$

□

A *stutter step* in a sequence is a repeated occurrence of the same element. In the following chapters, we sometimes will be interested in sequences without *stuttering*, where any two successors in a sequence are different. This leads to the following definition.

Definition 2.7
Define $\natural : A^{\mathcal{F}} \to A^{\mathcal{F}}$ by

$$\natural\varepsilon = \varepsilon$$
$$\natural(a \cdot \alpha) = \begin{cases} a \cdot \varepsilon & \text{if } \alpha = \varepsilon \\ a \cdot \natural\alpha & \text{if } \alpha \neq \varepsilon \wedge a \neq \text{first}(\alpha) \\ \natural\alpha & \text{if } \alpha \neq \varepsilon \wedge a = \text{first}(\alpha) \end{cases}$$

This operator can be extended to work on infinite sequences as well. So we assume that $\natural\alpha$ is the (finite or infinite) sequence which results in removing all stuttering steps of an infinite sequence α. □

Similarly, one can also define the stuttering closure, which is the set of all prefixes of all stutterings derivable from a set of sequences.

Definition 2.8 (Stuttering closure)
Define $\text{SC} : A^{\underline{\omega}} \to A^{\underline{\omega}}$ by

$$\text{SC}(\alpha) = \{\beta \mid \natural\beta \text{ is prefix of } \natural\alpha\} \ .$$

As usual, this operator is extended to sets of sequences. □

2.2 Basic Definitions

Let $\mathcal{V}$ be a finite set of variables, and $\mathcal{V}_{index}$ a set of index variables. We assume that a type $\text{type}(v)$ is associated to each variable $v \in \mathcal{V}$ and that we basically have four different types of variables in $\mathcal{V}$:

- one special parameter variable N of type $\mathbb{N}$,
- array variables (of identical size), each having a finite type,
- finite type variables, and
- numeric variables with values in $\mathbb{N}$.

For an array variable $x \in \mathcal{V}$, the type of x is the set of values which can be entered into an array position $x[i]$. Let $\mathbb{V}$ be the union of all the types of variables from $\mathcal{V}$. We will use x as array variable, y will denote a finite type variable, and z is a typical numeric variable.

Definition 2.9 (Index term, term, predicate)
An *index term* is a term

$$j ::= i \mid z \mid j + c \mid j - c \mid c$$

where $i \in \mathcal{V}_{index}$ is an index variable, $z \in \mathcal{V}$ is a numeric variable, and $c \in \mathbb{N}$ is a constant[1].

A *term* t over $\mathcal{V}$ is built by the rule

$$t ::= v \mid x[j] \mid y$$

where $v \in \mathbb{V}$ is a value, $x \in \mathcal{V}$ is an array variable, $y \in \mathcal{V}$ is a normal variable, and j is an index term. The *type* of a term is the type of its variable or constant.

A *predicate* over $\mathcal{V}$ is built using the following rule

$$\begin{aligned} p ::= {} & tt \mid ff \mid t_1 = t_2 \mid j_1 = j_2 \mid j_1 < j_2 \mid p_1 \wedge p_2 \mid \neg p \\ & \mid \forall_N i : p \mid \exists_N i : p \mid (p) \end{aligned}$$

where t_1 and t_2 are terms of the same type, j_1 and j_2 are index terms, $i \in \mathcal{V}_{index}$ is an index variable. $\forall_N i : p$ and $\exists_N i : p$ are bounded quantifications over the natural numbers less than N. Since we restrict predicates to this bounded quantification only, we do not allow to negate formulae with quantification inside, and usually omit the subscript N.

The set of *free index variables* free(p) is defined as usual. If $\{i_1, \ldots, i_k\}$ is the set of free index variables of a predicate p, then this is denoted by $p(i_1, \ldots, i_k)$ or $p(\{i_1, \ldots, i_k\})$. If a predicate has no free variables, it is called *closed*. Analogously, a term or index term is called closed, if it does not have any free index variables. □

[1] We will later restrict the index terms used in various predicates, especially the use of constants.

We will use the usual boolean connectives as abbreviations, e.g., $p \vee q$ for $\neg(\neg p \wedge \neg q)$.

Definition 2.10 (Substitution)
For a predicate ϕ, a term or index term s, and a program or index variable w, the *substitution* of w by s in ϕ, denoted by $\phi[s/w]$, is defined inductively as follows:

- $tt[s/w] \stackrel{\text{def}}{=} tt$ and $\mathit{ff}[s/w] \stackrel{\text{def}}{=} \mathit{ff}$
- $(t_1 = t_2)[s/w] \stackrel{\text{def}}{=} t_1[s/w] = t_2[s/w]$
- $(j_1 < j_2)[s/w] \stackrel{\text{def}}{=} j_1[s/w] < j_2[s/w]$
- $(p_1 \wedge p_2)[s/w] \stackrel{\text{def}}{=} p_1[s/w] \wedge p_2[s/w]$
- $(\neg p)[s/w] \stackrel{\text{def}}{=} \neg(p[s/w])$
- for $Q \in \{\forall_N, \exists_N\}$,
$$(Qi : p)[s/w] \stackrel{\text{def}}{=} \begin{cases} Qi : p & \text{if } i = w \\ Qi : p[s/w] & \text{if } i \neq w \wedge i \notin \text{free}(s) \\ Qi' : p[i'/i][s/w] & \text{if } i \neq w \wedge i \in \text{free}(s) \\ & \quad \wedge\, i' \notin (\text{free}(p) \cup \text{free}(s)) \\ & \quad \wedge\, i' \neq w \end{cases}$$
- $(p)[s/w] \stackrel{\text{def}}{=} (p[s/w])$
- $v[s/w] \stackrel{\text{def}}{=} v$
- $c[s/w] \stackrel{\text{def}}{=} c$
- $(x[j])[s/w] \stackrel{\text{def}}{=} \begin{cases} s[j] & \text{if } x = w \\ x[j[s/w]] & \text{if } x \neq w \end{cases}$
- $y[s/w] \stackrel{\text{def}}{=} \begin{cases} s & \text{if } y = w \\ y & \text{if } y \neq w \end{cases}$

□

Definition 2.11 (Evaluation)
Assume a set $\mathcal{V}$ of variables containing the parameter variable N. An *evaluation* σ of $\mathcal{V}$ is a well-typed[2] function $\sigma : \mathcal{V} \to \mathbb{V} \cup [[n] \to \mathbb{V}]$, for some $n \geq 1$.

[2]Types are simply sets in this thesis.

An evaluation σ is *well-typed*, if it assigns to each array $x \in \mathcal{V}$ a value from $[[n] \to \text{type}(x)]$, to each non-array variable y a value from $\text{type}(x)$, and if $\sigma(N) = n$.

For a closed term t and an evaluation σ, the value of t in σ, $\mathcal{E}[t](\sigma)$, is defined as:

- $\mathcal{E}[v](\sigma) = v$
- $\mathcal{E}[x[j]](\sigma) = \sigma(x)(j)$
- $\mathcal{E}[y](\sigma) = \sigma(y)$

Here, $v \in \mathbb{V}$ is a constant, $x \in \mathcal{V}$ is an array variable, $y \in \mathcal{V}$ a non-array variable, and j is an index term. □

Definition 2.12
Let σ be an evaluation, and p be a closed predicate over variables $\mathcal{V}$ (where N is replaced by some constant n). The relation $\sigma \models p$ is defined by induction on the structure of p as follows:

- $\sigma \models tt$
- $\sigma \not\models ff$
- $\sigma \models t_1 = t_2$ iff $\mathcal{E}[t_1](\sigma) = \mathcal{E}[t_2](\sigma)$
- $\sigma \models j_1 < j_2$ iff $\mathcal{E}[j_1](\sigma)$ is less than $\mathcal{E}[j_2](\sigma)$
- $\sigma \models p_1 \wedge p_2$ iff $\sigma \models p_1$ and $\sigma \models p_2$
- $\sigma \models \neg p$ iff not $\sigma \models p$
- $\sigma \models \forall_N i : p$ iff $\sigma \models p[\mu/i]$ for each possible value $\mu < \sigma(N)$
- $\sigma \models \exists_N i : p$ iff $\sigma \models p[\mu/i]$ for some value $\mu < \sigma(N)$

For the definition of transition relations we use predicates over variables $\mathcal{V}$ and $\mathcal{V}'$, where $\mathcal{V}'$ contains a primed copy x' for each variable $x \in \mathcal{V}$. We assume that $\mathcal{V} \cap \mathcal{V}' = \emptyset$. The primed versions refer to the successor state of the transition, the unprimed versions refer to the state before the transition takes place. For two evaluations σ_1, σ_2 of variables $\mathcal{V}$ define

$$(\sigma_1, \sigma_2) \models p \text{ iff } \sigma \models p,$$

where σ is an evaluation of $\mathcal{V} \cup \mathcal{V}'$ such that

$$\sigma(x) = \begin{cases} \sigma_1(x) & \text{if } x \in \mathcal{V} \\ \sigma_2(x) & \text{if } x \in \mathcal{V}' \end{cases}$$

□

2.3 Decision Problems

In this section we briefly introduce the notion of a decision problem [TA95], and the reduction of one decision problem to another. The reduction can be used to show undecidability of a problem by reducing an already known undecidable problem to it. We will use these notions in Chapter 4 frequently.

Definition 2.13 (Decision problem)
A *decision problem* is given by a pair $\mathcal{P} = (E_P, P)$ consisting of a set of inputs E_P and a set $P \subseteq E_P$ for which the answer "yes" has to be given.

A decision problem $\mathcal{P} = (E_P, P)$ is called *decidable* if there exists an algorithm which answers for each $x \in E_P$, whether $x \in P$ or not. □

We are especially interested in the question whether there exist decision procedures for certain classes $\mathcal{C}$ of parameterized systems and (subclasses of) verification properties over $\mathcal{V}$ as given in Definition 3.20.

The precise formulation of this question as a decision problem $\mathcal{P} = (E_P, P)$ is given by

$$E_P \stackrel{\text{def}}{=} \{(\mathcal{S}, \phi) \mid \mathcal{S} \in \mathcal{C}, \phi \text{ is a verification property over } \mathcal{V}\}$$

and

$$P \stackrel{\text{def}}{=} \{(\mathcal{S}, \phi) \in E_P \mid \mathcal{S} \text{ satisfies } \phi\} \ .$$

Consequently, a procedure deciding this problem $\mathcal{P}$ has to decide for arbitrary systems $\mathcal{S} \in \mathcal{C}$ and properties $\phi \in \mathcal{V}$, whether $\mathcal{S} \models \phi$ is valid or not.

In case such a problem is undecidable, which means that there cannot exist an algorithm deciding the problem, one usually does not prove this directly. Instead, one chooses a known undecidable problem and *reduces* this problem to the new one which is be proven undecidable. So we first define the very important notion of *reducibility*, which essentially means that one decision problem can be decided using the decision procedure for another decision problem.

Definition 2.14 (Reducibility)
Given two decision problems $\mathcal{P} = (E_P, P)$ and $\mathcal{Q} = (E_Q, Q)$. $\mathcal{P}$ is called *reducible* to $\mathcal{Q}$, if and only if there exists a computable total function $f : E_P \to E_Q$ such that

$$\forall x \in E_P : x \in P \Leftrightarrow f(x) \in Q \ .$$

□

The importance of this notion is given by the following lemma.

Lemma 2.15 (Reduction lemma)
Given two decision problems $\mathcal{P}$ and $\mathcal{Q}$. If $\mathcal{P}$ is undecidable and $\mathcal{P}$ is reducible to $\mathcal{Q}$, then $\mathcal{Q}$ is undecidable.

Proof: Assume that $\mathcal{P}$ is reducible to $\mathcal{Q}$ and $\mathcal{Q}$ is decidable by an algorithm A. Then there exists a computable function f such that $x \in P \Leftrightarrow f(x) \in Q$. Then the algorithm which, given an input $x \in P$, first computes $f(x)$, and then works like A on $f(x)$ decides $\mathcal{P}$. ■

The computable total function f in Definition 2.14 is usually simply an (algorithmic) construction given in an undecidability proof. Such a construction defines, given an arbitrary system $\mathcal{S}$ in a certain class, a system $\mathcal{S}'$ in another class. In our examples, $\mathcal{S}$ is usually a two counter machine (for which the halting problem, formally defined in Definition 2.20, is known to be undecidable), and $\mathcal{S}'$ is a parameterized system in a certain class. The algorithmic construction then ensures that $\mathcal{S}$ is halting if and only if a certain property is valid for all instances of $\mathcal{S}'$.

Next we formalize these two counter machines.

2.4 Counter Machines

A *counter machine* [HMU01] is an automaton provided with a set of counters (or registers). These counters can hold any nonnegative integer, but can only distinguish between zero and nonzero values. Consequently, each step of the machine depends on its state and the information which of the counters are zero. In one step, a machine is able to

- change the state, and
- independently add 1 to or subtract 1 from any of its counters. Counters may also be left unmodified, and it is not allowed to subtract from a counter with value zero.

Such a counter machine may also be seen as a special multi-tape Turing machine [HMU01]. A multi-tape Turing machine has a finite number of tapes, with one tape head for each of them. It starts with input given on the first tape, and with the blank symbol on all cells of the other tapes. A move of a multi-tape Turing machine allows to write tape symbols independently on each of the tape, and also move the tape heads independently in any direction.

Counter machines can then be modeled by having only one symbol, and using one tape for each of the counters. An incrementation is then modeled by going right and writing down the symbol. A counter is decremented by writing blank (if the head is on the one non-blank symbol), and going left. The counter modeled by a tape has value zero, if the head is on a blank, otherwise the counter is greater zero.

The interesting result is that counter machines with only two counters are already strong enough to simulate a Turing machine, hence, they are able to accept every recursively enumerable language. This result can be shown by first showing that a counter machine with "enough" counters allows to simulate a Turing machine, and then showing that a two counter machine can simulate a counter machine with more counters.

Theorem 2.16
Every recursively enumerable language is accepted by a counter machine.

Proof: Assume we have only one symbol "I" and the blank symbol. Then, the information on each field is just one bit, and a sequence of fields corresponds to a binary number. So, one can use two counters c_r and c_l to encode the right and the left tape (seen from the head) as number $\prod_{i=0}^{n} f_i \cdot 2^i$, where $f_i = 1$ if the symbol on position n is "I", and $f_i = 0$ otherwise (the left tape is encoded from the head to the left, i.e., in "reverse" order compared to the right tape).

A third counter c_h could hold the head field, so this one is always 0 or 1. Then, writing a symbol is just changing this counter. Moving the head to the right corresponds to multiplying c_l by 2, adding c_h to the result, and dividing c_r by 2, setting c_h to the remainder of this division. Moving the head to the left can be done analogously. These operations are possible using some more counters. The reading of the head's actual symbol is just testing for $c_h = 0$.

■

Theorem 2.17
A counter machine with $n \geq 2$ counters can be simulated by a two counter machine.

Proof: The idea here is that by using Gödel encoding one can encode the n counters in one counter: If we have n counters $k_1, \ldots, k_n$, then the counter

$$c_1 \stackrel{\text{def}}{=} \prod_{j=1}^{n} (p_j)^{k_j}$$

encodes them, where p_j is the j'th prime. For $n = 3$ this encoding is $2^{k_1} \cdot 3^{k_2} \cdot 5^{k_3}$. Then, incrementing the j'th counter is simply multiplying c_1 with p_j, decrementation is dividing. The test whether counter j is zero corresponds to testing whether c_1 is dividable by p_j; if this is not the case, counter j is zero. All these operations can be done using the second counter. ■

We use a slight variation here in which at most one counter is modified by a transition. It is easy to see that this model is computationally equivalent to the previous version.

Definition 2.18 (Two counter machine)
A *two counter machine* $M = (Q, q_0, R_0, R_1, T)$ consists of a set of states Q with $q_0 \in Q$ being the initial state, two registers R_0 and R_1 ranging over the natural numbers, and a set T of instructions (q, b_0, b_1, j, D, q') which consist of the following components:

- $q, q' \in Q$ is the initial resp. target state of the instruction.
- b_0 and b_1 are boolean values. They are enabling conditions, such that the instruction can only be executed in case $b_i \Rightarrow (R_i = 0)$.
- $j \in \{0, 1\}$ is the number of the register to be modified.
- $D \in \{inc, dec, nop\}$ is the operation on register R_j.

A two counter machine M is called *deterministic*, if for each $q \in Q$ and all boolean values b_0, b_1, there is at most one instruction $(q, b_0, b_1, \cdot, \cdot, \cdot)$ in T. □

Remark: We will sometimes abbreviate "two counter machine" with 2CM.

It is straightforward to define configurations and computations for this model.

Definition 2.19 (Configuration, computation)
Let $M = (Q, q_0, R_0, R_1, T)$ be a two counter machine.

A *configuration* of M is a triple (q, k_0, k_1) with $q \in Q$ and $k_0, k_1 \in \mathbb{N}$.

For two configurations $c = (q, k_0, k_1)$ and $c' = (q', k_0', k_1')$ of M and an instruction $t = (p, b_0, b_1, j, D, p')$ we say that c' is a *t-successor* of c if

- $p = q$ and $p' = q'$,
- $k_i = 0$ if b_i, for $i = 0, 1$,
- $k_{1-j}' = k_{1-j}$, and

- $k'_j = \begin{cases} k_j & \text{if } D = nop \\ k_j + 1 & \text{if } D = inc \\ k_j - 1 & \text{if } D = dec \end{cases}$

A *computation* (or *run*) of M is a sequence $r \in (Q, \mathbb{N}, \mathbb{N})^{\underline{\omega}}$ such that

- $r(0) = (q_0, 0, 0)$, and
- for each $i < |r| - 1$, there exists an instruction $t \in T$ such that $r(i+1)$ is a t-successor of $r(i)$.

□

Definition 2.20 (Halting)
We say that a two counter machine is *halting*, if there exists a finite computation $r \in (Q, \mathbb{N}, \mathbb{N})^{\underline{\omega}}$ of that machine such that there is no successor of its final configuration. □

The following result follows directly from Theorem 2.16, Theorem 2.17, and the well-known undecidability of the halting problem for Turing machine.

Theorem 2.21
Let M be a two counter machine. It is not decidable whether M is halting or not.

Without loss of generality, we can also restrict ourselves to deterministic two counter machines which stop in a certain stop configuration.

Corollary 2.22
Let M be a deterministic two counter machine. It is not decidable whether M can reach the halting configuration in state q_{stop} with $R_0 = R_1 = 0$.

Chapter 3

Classes of Parameterized Systems

This chapter defines a formal computation model for parameterized systems, and discusses how to restrict the general model to get interesting subclasses. Moreover, we define the properties that we like to prove of these systems.

We start with a very general model for parameterized systems presented in Section 3.1 and the definition of its semantics given in Section 3.2. In Section 3.3 we present the class of properties we will investigate. Afterwards, we discuss in Section 3.4 how to restrict the general system to get a number of interesting different system classes.

Having thus defined the formal verification problem, we proceed by discussing in Section 3.5 how our system relates to parameterized systems with many parameters. Finally, after presenting some technical notions in Section 3.6 that we need in the following chapter, we prove some general properties valid for all system classes in Section 3.7.

3.1 Parameterized Systems

In this section we present our general computation model for parameterized systems.

Definition 3.1 (Parameterized system)
A *parameterized system* $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ consists of:

- A (parameter) variable N ranging over the natural numbers.
- A set $\mathcal{V}$ of variables as follows

- $x_1, \ldots x_a$: array $[N]$, each array is of a fixed finite type[1],
- $y_1, \ldots y_b$: each of a fixed finite type,
- $z_1, \ldots z_c$: of type $\mathbb{N}$.

We will refer to the variables $y_1, \ldots y_b$ as *normal* or *global* variables, to $z_1, \ldots z_c$ as *numeric variables*. They are denoted by $\mathcal{V}_Y$ resp. $\mathcal{V}_Z$. The array variables are denoted by $\mathcal{V}_X$.

- A closed predicate $\theta(N, \mathcal{V})$ describing the initial states.
- A closed transition relation predicate $\rho(N, \mathcal{V}, \mathcal{V}')$ which relates pre-state variables $\mathcal{V}$ with their primed counterparts, the post-state variables $\mathcal{V}'$.

□

Before proceeding by defining the semantics of a parameterized system, we give a small example.

Example 3.2 (Simple resource allocation)
As an example we use a very simple resource allocation algorithm. Assume that a resource is given which a number of clients need to access. It should be ensured by the algorithm that at most two clients at a time have simultaneous access. Using a (global) semaphore variable, one client i can be modeled informally as shown in Figure 3.1.

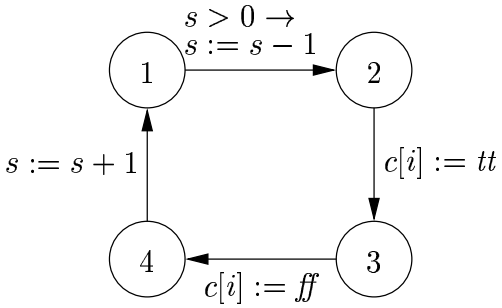

Figure 3.1: Simple resource allocation

In this algorithm, s is the semaphore variable, initially set to 2. A flag $c[i]$ is set to tt whenever the resource is accessed by client i.

In our framework, such a parameterized system can be modeled as a system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ consisting of

[1]Instead of a finite type one could also choose booleans for all arrays, since an array of arbitrary type can easily be modeled using a number of boolean arrays. But for the constructions following later, arbitrary finite types are more convenient.

- $\mathcal{V} \stackrel{\text{def}}{=} \{s, c, l\}$ with
  - $\text{type}(s) = \{0, 1, 2\}$,
  - $\text{type}(c) = \text{array } [0 \ldots N-1] \text{ of Bool}$,
  - $\text{type}(l) = \text{array } [0 \ldots N-1] \text{ of } \{1, 2, 3, 4\}$,
- transition relation predicate

$$\begin{aligned}\rho \stackrel{\text{def}}{=} \exists i < N : \; & (l[i] = 1 \wedge s > 0 \wedge l'[i] = 2 \wedge s' = s - 1 \\ & \wedge c' = c \wedge \forall j < N.j \neq i \Rightarrow l'[j] = l[j]) \\ & \vee (l[i] = 2 \wedge l'[i] = 3 \wedge c'[i] = tt \wedge s' = s \\ & \wedge \forall j < N.j \neq i \Rightarrow l'[j] = l[j] \wedge c'[j] = c[j]) \\ & \vee (l[i] = 3 \wedge l'[i] = 4 \wedge c'[i] = ff \wedge s' = s \\ & \wedge \forall j < N.j \neq i \Rightarrow l'[j] = l[j] \wedge c'[j] = c[j]) \\ & \vee (l[i] = 4 \wedge l'[i] = 2 \wedge s' = s + 1 \\ & \wedge c' = c \wedge \forall j < N.j \neq i \Rightarrow l'[j] = l[j]) \;\; ,\end{aligned}$$

- and initial state predicate

$$\theta \stackrel{\text{def}}{=} s = 2 \wedge \forall i < N.l[i] = 1 \wedge \neg c[i] \;\; .$$

Note that we need to model the locations (program control flow) explicitly by a parameterized array l. ♣

3.2 Semantics

Let $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ be a parameterized system in this section, and assume $n \in \mathbb{N}$. Intuitively, the system instance $\mathcal{S}(n)$ is the system which we get by instantiating the parameter variable N by n. Formally, this can be expressed as follows.

Definition 3.3 (State, System instance)
A *state* σ is a well-typed evaluation $\sigma : \mathcal{V} \to \mathbb{V} \cup [[n] \to \mathbb{V}]$ for the variables $\mathcal{V}$ fulfilling $\sigma(N) = n$, where $\mathbb{V}$ is again the union of all types of variables of $\mathcal{S}$, for some $n \in \mathbb{N}$.

Define $\Sigma_{\mathcal{V},n}$ as the set of all states over variables $\mathcal{V}$ with array size n. Sometimes we also use the notation $\Sigma_{\mathcal{S},n}$, meaning the states over the variables of $\mathcal{S}$.

The *instance* of a parameterized system (or short *system instance*) is the system consisting of the same components as $\mathcal{S}$, to which we associate the

state space $\Sigma_{\mathcal{Y},n}$. This can be seen as the instantiation of N with the natural number $n \in \mathbb{N}$. We denote such an instance by $\mathcal{S}(n)$.

If the context is clear, i.e., the system instance we refer to is obvious, we will denote the set of states of the system instance by Σ. □

We are now able to define the computational behavior of a system instance.

Definition 3.4 (Trace)
Let $\mathcal{S}(n)$ be a system instance of a parameterized system with transition relation ρ, for some $n \in \mathbb{N}$, and assume $\sigma, \sigma' \in \Sigma$. We call σ' a ρ-successor of σ, if $(\sigma, \sigma') \models \rho$.

A *trace* of instance $\mathcal{S}(n)$ is a sequence $\alpha \in \Sigma^{\omega}_{=}$ of states of $\mathcal{S}(n)$ fulfilling

1. $\alpha(0) \models \theta$,
2. $\forall j < |\alpha| - 1 : \alpha(j+1)$ is a ρ-successor of $\alpha(j)$, and
3. α is maximal, i.e., it is either infinite, or α is finite and there is no ρ-successor of $\text{last}(\alpha)$.

□

Definition 3.5 (Semantics of an instance)
Let $\mathcal{S}$ be a parameterized system and $n \in \mathbb{N}$. The semantics of instance $\mathcal{S}(n)$ is the set of all traces of $\mathcal{S}(n)$. This set is denoted by $[\![\mathcal{S}(n)]\!]$. □

Example 3.6
Let us go back to the resource allocation algorithm $\mathcal{S}$ given in Example 3.2. The instance of size 3 of this system, $\mathcal{S}(3)$ has, e.g., such a state:

$$\sigma = (s \mapsto 0, l \mapsto \bar{l}, c \mapsto \bar{c}),$$

where

- $\bar{l} = (0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3)$, and
- $\bar{c} = (0 \mapsto \mathit{ff}, 1 \mapsto \mathit{ff}, 2 \mapsto \mathit{tt})$.

So, in this state σ, client 0 is located in program location 1 and its c value is currently ff, client 1 is at location 2 with the same c value, and client 2 is at 3 with its c set to tt. ♣

3.3 Verification Properties

In this section we will formally define the class of properties which we want to prove about parameterized systems. We start by defining Kripke structures as a model of computations for concurrent systems, and define the linear temporal logic to express properties. Finally, we show how to extend the framework to reason about parameterized systems.

Verification properties are extended with universal quantification over indices. By this universal quantification, properties are expressible which should be valid for all processes, if one has the picture in mind that each system instance $\mathcal{S}(n)$ consists of n processes. In Section 3.4, we explain this point of view in more detail.

Definition 3.7 (Kripke structure)
A *Kripke structure* M over the set of atomic propositions $\mathcal{P}$ is a quadruple $M = (S, S_0, R, L)$ where

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation, and
- $L : S \to 2^{\mathcal{P}}$ is a labeling function, i.e., is a function that associates with each state S the set of atomic propositions that are true in the state.

A *path* in M from state s is a maximal sequence of states $\alpha = s_0, s_1, \ldots,$ such that $s_0 = s$ and $(s_i, s_{i+1}) \in R$, for all positions $i < |\alpha| - 1$. The path is *maximal*, if it is infinite or if there does not exist a successor of the final state.

A *computation* of M is a path in M starting in an initial state $s_0 \in S_0$. □

3.3.1 Propositional Linear Temporal Logic

For terminating sequential and concurrent programs, correctness, or more generally system properties, can be formulated in terms of pre- and post-condition pairs in formalisms such as Hoare's Logic [Hoa69], and they can be verified with proof methods such as Floyd's inductive assertion method which basically consists in finding an inductive assertion for a program, such that the postcondition is implied by the assertion of final program locations [Flo67, dRdBH+01].

In contrast, if one wants to reason about reactive, *nonterminating* systems, such as operating systems, where no final state is ever reached, these formalisms are of little use, because one needs to reason about infinite, ongoing computations.

Temporal logic [Pnu77, Eme91, MP89, MP95] is a useful formalism to express properties for this type of *reactive systems*, which can be used both for formal specification and verification of computer programs, especially nonterminating programs.

In this section we will define the formal syntax and semantics of propositional temporal logic. Formulae are built up from atomic propositions, the boolean connectives, and some temporal operators. We use a restricted variant of this logic without past operators.

We assume throughout this section that a set $\mathcal{P}$ of atomic propositions is given.

Definition 3.8 (PTL)
A *propositional linear temporal logic formula* (PTL formula) over $\mathcal{P}$ is built from the atomic propositions using the time modalities $\bigcirc$ ("next") and U ("until") as follows

$$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathsf{U}\, \phi_2$$

where $p \in \mathcal{P}$.

As usual, we use the known logical operators like $\Rightarrow$ as abbreviations and further define the temporal operators

$$\Diamond\phi \stackrel{\text{def}}{=} tt \,\mathsf{U}\, \phi$$
$$\Box\phi \stackrel{\text{def}}{=} \neg\Diamond\neg\phi$$

which are called *eventually* and *always*. □

A PTL formula is interpreted over sequences of *states* of a system.

Definition 3.9 (PTL Semantics)
Let $M = (S, S_0, R, L)$ be a Kripke structure over $\mathcal{P}$, $\alpha \in S^{\underline{\omega}}$ be a path in M, and ϕ a PTL formula. $\alpha \models \phi$ is defined inductively on the structure of ϕ:

- $\alpha \models p$ iff $p \in L(\alpha(0))$
- $\alpha \models \neg\phi$ iff not $\alpha \models \phi$
- $\alpha \models \phi_1 \wedge \phi_2$ iff $\alpha \models \phi_1$ and $\alpha \models \phi_2$
- $\alpha \models \bigcirc\phi$ iff $|\alpha| > 1 \wedge \alpha|^1 \models \phi$

- $\alpha \models \phi_1 \mathsf{U} \phi_2$ iff $\exists j < |\alpha|.(\alpha|^j \models \phi_2 \wedge \forall k < j.\alpha|^k \models \phi_1)$

The set of all models of a PTL formula ϕ is denoted by $[\![\phi]\!] = \{\alpha \in S^{\underline{\omega}} \mid \alpha \models \phi\}$. □

The most interesting operator is the "until". $p \mathsf{U} q$ intuitively expresses that there is some time in the future where q is valid, and p holds in the meantime. This is the so-called strong until operator. In the literature also the weak until, also called *unless*, can be found, which intuitively means that p holds for as long as q does not, even forever if need be.

There are also variations of PTL in the literature using past modalities [LPZ85, GPSS80]: a previous operator corresponding to the next operator is added, as well as an operator corresponding to the until operator, which expresses that in the past, some property was valid, and some other holds until now. It is proved that past modalities does not add to the logic's expressiveness, although they are sometimes convenient to express particular properties.

We denote with PTL\X the restricted logic without $\bigcirc$ ("next") operator. The "X" is also often used as symbol for this operator.

For an extensive introduction to temporal logic, its varieties, and comparisons of different formalisms, we refer the reader to [Eme91].

For the rest of the section let us assume that a Kripke structure $M = (S, S_0, R, L)$ is given.

Lemma 3.10 (Stuttering equivalence)
For all PTL\X formulae ϕ, for all $\alpha \in S^{\underline{\omega}}$:

$$\alpha \models \phi \text{ if and only if } \natural\alpha \models \phi \ .$$

Proof: Structural induction on the number of U operators in ϕ.

Base case: If ϕ has zero operators, then it is evaluated in the initial state, so this case is trivial since $\alpha(0) = \natural\alpha(0)$.

Inductive step: Let $\phi = \phi_1 \mathsf{U} \phi_2$, and assume the induction hypothesis holds for ϕ_1, ϕ_2.

"⇒": If $\alpha \models \phi$, then by Definition 3.9 there exists $j < |\alpha|$ such that $\alpha|^j \models \phi_2$, and $\forall k < j : \alpha|^k \models \phi_1$. Since α is a stuttering of $\natural\alpha$, there exists a corresponding position in $\natural\alpha$, namely $j' \stackrel{\text{def}}{=} |\natural(\alpha|^j)| - 1$:

By induction hypothesis we derive $\natural(\alpha|^j) \models \phi_2$. Since $\natural(\alpha|^j) = (\natural\alpha)|^{j'}$ holds, we can conclude $(\natural\alpha)|^{j'} \models \phi_2$. Now let $k' < j'$. Then there exists a position $k < j$ such that $|\natural(\alpha|^k)| - 1 = k'$ and $\natural(\alpha|^k) = (\natural\alpha)|^{k'}$. By induction hypothesis we conclude $\natural(\alpha|^k) \models \phi_1$, which implies $(\natural\alpha)|^{k'} \models \phi_1$, proving this direction.

"$\Leftarrow$": can be proved in a similar way.

■

Note that by using this lemma twice, this result is easily extended to two different stutterings of some trace.

Among temporal logic properties, there are the classical classes of safety and liveness properties [Lam77].

- A *safety property* is a property which intuitively expresses that never "something bad" happens. So, if the property is not fulfilled, then this can be observed by a finite execution of the system.
- A *liveness property* expresses that "something good" will eventually happen. Thus, every counterexample of such a property must be infinite, since one can never be sure if the "good" will happen later.
- Finally, an *invariance property* should be valid *at any time* in each computation, so this is basically a state property that is expected always to hold. This is a subclass of the class of safety properties.

To formalize these ideas [AS85] the following topology on (infinite) words is helpful.

Definition 3.11 (Cantor's topology)
Let A be a set. *Cantor's topology* on A^ω is the topology induced by the metric $d : A^\omega \times A^\omega \to \mathbb{R}_{\geq 0}$, defined by

$$d(\alpha, \alpha') \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \alpha = \alpha' \\ \frac{1}{2^n} & \text{if } \alpha \neq \alpha' \wedge n = \min\{i \mid \alpha(i) \neq \alpha'(i)\} \end{cases}$$

□

The next definition gives an operator useful for embedding the set of finite traces in the infinite traces by repeating the last sequence member infinitely often.

Definition 3.12
Let $\alpha \in A^{\underline{\omega}}$. We define $\alpha_\infty \in A^\omega$ by

$$\alpha_\infty(i) \stackrel{\text{def}}{=} \begin{cases} \alpha(i) & \text{if } i < |\alpha| \\ \text{last}(\alpha) & \text{otherwise} \end{cases}$$

So $\cdot_\infty : A^{\underline{\omega}} \to A^\omega$ is an operator leaving infinite traces untouched, and making finite traces infinite by stuttering. This operator is extended to sets of sequences as usual. □

We state that this embedding is consistent with PTL.

Lemma 3.13
For all PTL properties ϕ, for all $\alpha \in S^{\underline{\omega}}$,

$$\alpha \models \phi \text{ if and only if } \alpha_\infty \models \phi \ .$$

Proof: Using Lemma 3.10 we conclude:

$$\begin{aligned} & \alpha_\infty \models \phi \\ \text{iff } & \natural\alpha_\infty \models \phi \\ \text{iff } & \natural\alpha \models \phi \\ \text{iff } & \alpha \models \phi \end{aligned}$$

■

Using Cantor's topology, one can nicely define the property classes as follows.

Definition 3.14 (Safety, liveness, invariance)
A *safety set* is a closed set $L \subseteq S^\omega$. A *liveness set* is a dense set $L \subseteq S^\omega$, i.e., $\bar{L} = S^\omega$, where $\bar{L}$ is the usual topology closure.

A PTL formula ϕ is a *safety property*, if $[\![\phi]\!]_\infty$ is a safety set; it is a *liveness property*, if $[\![\phi]\!]_\infty$ is a liveness set.

An *invariance property* is given by a formula $\Box p$, where p is a state formula, i.e., p does not contain temporal modalities. □

Example 3.15
Let $\mathcal{P}$ be a set of propositions, and ψ, ψ_1, ψ_2 boolean combinations of propositions.

- The following formulae are safety properties:

  $$\Box\psi \qquad \neg(\psi_1 \,\mathsf{U}\, \psi_2)$$

  The second formula is an example for a safety property that is no invariance property.

- If ψ is satisfiable, then

  $$\Diamond\psi \qquad \Box\Diamond\psi \qquad \Diamond\Box\psi$$

  define liveness properties. The first one expresses that ψ is true at some future time, the second expresses that ψ holds infinitely often, and the last one that ψ holds continuously from a particular time on.

♣

It is not a trivial task to determine whether a given PTL formula is a safety property, as this is shown to be PSPACE-complete [Sis94, AS87]. On the other hand, there are sufficient syntactic characterizations for PTL formulae being safety properties, e.g., if the only time modalities used in a formula are $\Box$ and $\bigcirc$.

Another approach is to *syntactically* define the class of safety properties as done in [MP92, MP95]. There, safety properties are defined by formulae of the form $\Box\phi$, for some formula ϕ using only *past operators*, a definition that is equivalent to ours.

We give the following lemma to relate this definition for safety properties to another well-known characterization in the literature for PTL\X properties. Intuitively, this lemma states that the violation of a safety property occurs after a finite execution of the system, so every word not in a given safety set has a *finite bad prefix*. In fact, a similar lemma also holds for full PTL if one restricts to infinite traces only.

Lemma 3.16 (Safety property characterization)
Let ϕ be a safety property in PTL\X and $\alpha \in S^{\underline{\omega}}$. Then, $\alpha \not\models \phi$ if and only if there exists a finite prefix α' of α which is a bad prefix, i.e., for all $\beta \in S^{\underline{\omega}}$, $\alpha' \cdot \beta \not\models \phi$.

Proof: '$\Leftarrow$': trivial, choose β as the "tail" of α.

'$\Rightarrow$': Assume that $\alpha \not\models \phi$, where ϕ is a safety property. Let us first take the case that α is infinite.

Since $[\![\phi]\!]_\infty$ is a safety set, this set is closed. Consequently, $M \stackrel{\text{def}}{=} S^\omega \setminus [\![\phi]\!]_\infty$ is an open set, with $\alpha \in M$.

Since M is open and the topology is induced by a metric, there exists an ϵ-environment of α that is a subset of M, i.e., there exists an $\epsilon > 0$ such that $\forall \beta \in S^\omega : d(\alpha, \beta) < \epsilon \Rightarrow \beta \in M$. Choose ϵ correspondingly.

Now choose n such that $\frac{1}{2^n} < \epsilon$, and $\alpha' \stackrel{\text{def}}{=} \alpha[0 \ldots n]$. Then, for all $\beta \in S^{\underline{\omega}}$, the first position in which α and $(\alpha' \cdot \beta)_\infty$ differs (if there is any) lies beyond position n. Consequently, $d(\alpha, (\alpha' \cdot \beta)_\infty) < \frac{1}{2^n} < \epsilon$, which implies $(\alpha' \cdot \beta)_\infty \in M$. Therefore, $(\alpha' \cdot \beta)_\infty \notin [\![\phi]\!]_\infty$ for all $\beta \in S^{\underline{\omega}}$, which implies $(\alpha' \cdot \beta)_\infty \not\models \phi$. By Lemma 3.10 we conclude $\alpha' \cdot \beta \not\models \phi$.

Now assume that α is finite. By Lemma 3.13 we know that $\alpha_\infty \not\models \phi$. So from the first case we can conclude that there exists a prefix α' of α_∞ such that $\alpha' \cdot \beta \not\models \phi$ for all $\beta \in S^{\underline{\omega}}$.

In case α' is a prefix of α, everything is fine. Otherwise, let $\beta \in S^{\underline{\omega}}$. Then,

$$\begin{aligned} & \alpha' \cdot \beta \not\models \phi \\ \text{iff } & \alpha' \cdot \beta \models \neg\phi \\ \text{iff } & \alpha \cdot \beta \models \neg\phi && \text{(by Lemma 3.10)} \\ \text{iff } & \alpha \cdot \beta \not\models \phi \end{aligned}$$

Consequently, the lemma also holds in this case. ■

Corollary 3.17
For all safety properties ϕ in PTL\X, and finite sequences $\alpha \in S^{\mathcal{F}}$, if $\alpha \not\models \phi$, then $\alpha \cdot \beta \not\models \phi$ for all $\beta \in S^{\underline{\omega}}$.

Proof: Using Lemma 3.16, if $\alpha \not\models \phi$, then there exists a corresponding prefix of α prolongable with arbitrary sequences, none of them fulfilling ϕ. Consequently, each prolongation of α itself also does not satisfy ϕ. ■

3.3.2 Linear Temporal Logic

In this section we extend the linear temporal temporal logic by allowing (closed) predicates as given in Definition 2.9 in place of atomic propositions. This allows us to refer to program states of a particular system instance.

We restrict ourselves to a logic without the next operator, which is very natural for parameterized systems. In fact, for concurrent programs with an interleaving semantics, usually one is not interested in expressing that exactly the next computation step is of some particular form.

In the literature, also a fragment of indexed CTL*\X can be found as temporal logic for parameterized systems, e.g., in [EK00]. The fragment used there is in fact our logic LTL\X and its negation.

Let us assume that a set of variables $\mathcal{V}$ is given.

Definition 3.18 (LTL)
A *linear temporal logic formula* (LTL formula) over the variables $\mathcal{V}$ is built as a PTL\X formula, where the atomic propositions are replaced by quantifier-free predicates over $\mathcal{V}$ without quantifiers and index term comparisons.

An LTL formula is called *closed*, if the predicates from which it is built are all closed. □

The predicates are interpreted in a given state as usual. An LTL formula is interpreted over a sequence of states.

Definition 3.19 (LTL Semantics)
Let $\alpha \in \Sigma^{\underline{\omega}}$ be a sequence of states and ϕ a closed LTL formula. $\alpha \models \phi$ is defined as in Definition 3.9, where the base case

$$\alpha \models p \text{ iff } p \in L(\alpha(0))$$

for a proposition $p \in \mathcal{P}$ is replaced by

$$\alpha \models p \text{ iff } \alpha(0) \models p$$

for a predicate p over $\mathcal{V}$. □

Using LTL formulae, we can define the class of verification properties of interest.

Definition 3.20 (Verification properties)
Given a parameterized system $\mathcal{S}$ with variables $\mathcal{V}$, natural numbers $n_0, k \in \mathbb{N}$ such that $k \leq n_0$, and an LTL formula $\phi(i_1, \ldots, i_k)$ over $\mathcal{V}$ (with free index variables $i_1, \ldots, i_k$), we will focus on verification of properties of the form

$$\begin{aligned} \forall n \geq n_0 : \mathcal{S}(n) \models \forall i_1, \ldots, i_k < n. \\ (i_j \neq i_{j'} \text{ for all } j \neq j') \Rightarrow \phi(i_1, \ldots, i_k) \ , \end{aligned} \tag{3.1}$$

which we abbreviate by

$$\mathcal{S}(\geq n_0) \models \forall_d \, \phi(i_1, \ldots, i_k) \ .$$

In case we do not require the indices to be disjoint, we have properties like

$$\forall n \geq n_0 : \mathcal{S}(n) \models \forall i_1, \ldots, i_k < n. \phi(i_1, \ldots, i_k) \ ,$$

abbreviated similarly as before, omitting the subscript.

The special cases $k = 1$ (the property should hold for every one process), $n_0 = 1 = k$ (property should hold for every one process and for all system instances), and $k = 0$ (the property does not contain array variables, so it involves only the controller) result in the formulae given below:

- $\forall n \geq n_0 : \mathcal{S}(n) \models \forall i < n. \phi(i)$
- $\forall n \geq 1 : \mathcal{S}(n) \models \forall i < n. \phi(i)$
- $\forall n \geq n_0 : \mathcal{S}(n) \models \phi$

Another important special case is that we restrict $\phi(i)$ to be a invariant or safety property. □

For safety properties, verification is usually simpler since one only needs a bad prefix of a trace as counterexample, not a maximal trace of the system, as stated in Lemma 3.16. We make use of this fact for instance to prove decidability of the verification problem for safety properties in Section 4.1.1.

We give an example property for our running example.

Example 3.21
Now we are able to formally define some properties of interest for the resource allocation algorithm from Example 3.2. The first property expresses that at most two clients use the resource simultaneously:

$$\forall n \geq 3 : \mathcal{S}(n) \models \forall i_1, i_2, i_3 < n. i_1 \neq i_2 \wedge i_1 \neq i_3 \wedge i_2 \neq i_3 \Rightarrow \\ \Box\neg(c[i_1] \wedge c[i_2] \wedge c[i_3])$$

which can be abbreviated by

$$\mathcal{S}(\geq 3) \models \forall_d \Box\neg(c[i_1] \wedge c[i_2] \wedge c[i_3]) \ .$$

♣

Now we have formally defined the parameterized systems and the verification properties we are interested in. What is still missing is the notion of validity, i.e., the formal definition when a system $\mathcal{S}$ satisfies a verification property, which is straightforward.

Definition 3.22
Let $\phi(i_1, \ldots, i_k)$ be an LTL formula over $\mathcal{V}$, $\mathcal{S}$ be a parameterized system with variables $\mathcal{V}$, and $\alpha \in \Sigma^{\omega}_{\mathcal{V},n}$. The formula

$$\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k)$$

is valid, if for each $n \geq n_0$,

$$\mathcal{S}(n) \models \phi(n_1, \ldots, n_k)$$

for all possible values $n_1, \ldots, n_k < n$ with $|\{n_1, \ldots, n_k\}| = k$.

Similarly,

$$\mathcal{S}(\geq n_0) \models \forall \phi(i_1, \ldots, i_k)$$

is valid, if for all $n \geq n_0$ and values $n_1, \ldots, n_k < n$,

$$\mathcal{S}(n) \models \phi(n_1, \ldots, n_k)$$

holds. □

Remark 3.23
If $n < k$, then $\mathcal{S}(n) \models \forall_d \phi(i_1, \ldots, i_k)$ is always valid. By definition, one has to prove for all $n_1, \ldots, n_k < n$, with $n_i \neq n_j$ for $i \neq j$, that $\mathcal{S}(n) \models \phi(n_1, \ldots, n_k)$. Since no such n_j exist, this holds trivially. ♣

Remark 3.24
The verification problem

$$\mathcal{S}(\geq n_0) \models \forall \phi(i_1, \ldots, i_k)$$

can easily be transformed in a problem with disjoint indices. We illustrate this for $k = 3$:

$$\mathcal{S}(\geq n_0) \models \forall \phi(i_1, i_2, i_3)$$

is valid if and only if

$$\begin{aligned} & \mathcal{S}(\geq n_0) \models \forall_d \phi[i_1, i_1/i_2, i_3](i_1) \\ \text{and } & \mathcal{S}(\geq n_0) \models \forall_d \phi[i_1/i_2](i_1, i_3) \\ \text{and } & \mathcal{S}(\geq n_0) \models \forall_d \phi[i_1/i_3](i_1, i_2) \\ \text{and } & \mathcal{S}(\geq n_0) \models \forall_d \phi[i_2/i_3](i_1, i_2) \\ \text{and } & \mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, i_2) \end{aligned}$$

Therefore, we usually omit this problem and prove results only for the $\forall_d$ case. ♣

3.4 System Classification

We have already formally defined the general verification problem of interest. Now, we will first give an intuition about our point of view on our parameterized systems.

In a system with array variables x_j, normal variables y_j, and numeric variables z_j, one can interpret the array variables as *user process* variables, each process i using basically variables $x_j[i]$. Then, the normal and numeric variables are associated to a global *controller* process.

The system transitions and initial state predicates are unconstrained at this point. Now, we will impose several restrictions to the transition relation leading to different classes of parameterized systems.

These restrictions will be defined in such a way that the resulting transition relation fits to the interpretation of a system consisting of a controller and a set of user processes.

In order to find these restrictions it is useful to ask the following questions:

- What is the execution model of the system?
- What is the connection between controller and user processes?
- What variables are the user processes and the controller allowed to change?
- On which information is the enabledness of a transition based? What can a user process resp. the controller observe?

First, we have to take care that the relation defines the execution model of the system in a proper way. That means, the transition relation should ensure that all processes can make a step when we are interested in a synchronous model. If we want to model an asynchronous system with interleaving semantics, the relation should ensure that only one process makes a step.

The connection between user processes and controller gives rise to another form of synchronization, either one allows the controller to act synchronously with a user process, in case we have an interleaving model for the user processes, or with all of them in a synchronous model.

The next topic, which variables may be modified by a transition, also strongly influences what can be modeled: If, for example, one is interested in modeling a token, which can be given from one user process to another, one should allow an operation giving the token away. Such an operation is then a transition of process i which modifies its "own variables", which are the array variables at position i, as well as variables of other processes or normal variables. We will use a restricted model only allowing each component to modify its own variables.

The last point is another parameter yielding to a large variety of computation models. There are two different basic ways of dealing with the other user processes: Either, one quantifies universally over them, basically expressing that all of them stay in a certain set of local states, or one quantifies existentially, expressing that there is at least one process in a certain local state. One can also define several subclasses of these systems by, e.g., syntactically restricting user transitions to involve array variables only, thus not using controller information. This can be done for controller transitions as well.

The goal now is to restrict our general model in the ways described above to find interesting classes of parameterized systems. A first restriction dealing with the way transitions influence the state is the following.

Restriction 1
Only process i may modify the value of an array variable at position i.

This restriction corresponds to the popular single-writer multi-reader model, where each process P has variables which can be read by everyone, but may only be modified by itself. If we have array variables $x_1, \ldots, x_k$, then the variables writable by process i are $x_1[i], \ldots, x_k[i]$.

Restriction 2
In the transition relation predicate ρ and in the initial state predicate θ, no index is a constant.

We do not think that it is natural to refer to some fixed processes using a constant index in the transition relation, since doing so would assign a special "global" meaning to that process. In our opinion, information with a global meaning should be left to the controller, i.e., one should introduce normal variables which substitute the array variables with a constant index.

The initial state condition has to be limited further, at least for some constructions given later. Some reasons will become clearer in Section 4.1.1, see the Technical Note 4.14 for an example showing technical problems making one construction for a decidability result impossible. Of course, it is not clear whether there exists another way to show decidability even for a model with more freedom in the choice of the initial state condition.

Restriction 3
All initial state formulae have the form $\exists_N i_1, \ldots, i_k . \forall_N i : \phi(i, i_1, \ldots, i_k)$, where ϕ is a quantifier-free formula without any index term comparisons.

Unless something else is explicitly stated, we only investigate models without numeric variables.

Restriction 4
There are no numeric variables in $\mathcal{V}$.

We restrict ourselves to transition relations given as predicates in a generic shape defined below. The relation is constructed by use of a set of single transitions $\tau \in T$ given by predicates ρ_τ. The generic transition shape reflects Restriction 1, and also Restriction 2 is implied by the transition relation defined as in Definition 3.25, it is even more restrictive: The only indices allowed are i (the process doing the step) and one single quantified variable j.

Definition 3.25 (Transitions)
Assume that a set of transitions T is given, where each transition $\tau \in T$ is defined by a predicate $\rho_\tau(i)$ of the form[2]

$$\rho_\tau(i) = \forall_N \bar{j} : \phi(i) \wedge (\bar{j} \neq i \Rightarrow \psi(i, \bar{j})) \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y$$

[2]We use a guarded-command-like style to denote predicates, the formal semantics of

or

$$\rho_\tau(i) = \exists_N \bar{j} : \phi(i) \wedge \bar{j} \neq i \wedge \psi(i, \bar{j}) \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y \ .$$

Here, $\bar{j}$ is a list of index variables $j_1, \ldots, j_e$. For $Q \in \{\forall, \exists\}$, formula $Q_N \bar{j}$ abbreviates the e quantifications $Q_N j_1 . Q_N j_1 . \ldots . Q_N j_e$. The formula $\bar{j} \neq i$ abbreviates $\bigwedge_{1 \leq l \leq e} j_l \neq i$.

Such a predicate $\rho_\tau(i)$ describes a step of a process i. The former predicate is called an $\forall$-transition, the latter an $\exists$-transition. In this predicate,

- $\phi(i)$ is a quantifier-free guard about the local state of process i and about the normal variables (so we do not allow to add constants to i; all variables of $\mathcal{V}$ may occur, but arrays are only indexed with i),

- $\psi(i, \bar{j})$ is a quantifier-free guard without index term comparisons having at most $i, \bar{j}$ as free index variables (again, no addition of constants is allowed),

- and $\bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y$ describes the effect of the transition, i.e., the new values of the arrays at position i and the new values of the normal variables.

  Here, $\bar{c}_x$ and $\bar{c}_y$ are vectors of constant values. Each of these assignments can be seen as a set of simultaneous variable assignments. We use the pseudo statement *skip* to denote the empty assignment set.

  The assignment $\bar{x}[i] := \bar{c}$ means that the constant values in the vector $\bar{c}$ are simultaneously assigned to the vector $\bar{x}$ of some array variables at position i.

In a second step, we split this transition set into two sets T_U and T_C according to user and control components:

- T_U: These are the user process transitions. For each transition we require that the assignment part of the transition does not involve the normal variables $\bar{y}$.

- T_C: These are the control process transitions which do not change any $\bar{x}$ variables. Moreover, we require that the predicates involved are closed. Hence, i is not a free index variable and we replace the sub-formulae $\bar{j} \neq i$ by tt.

these transitions will be defined later. Although the special parameter variable N occurs in some sense freely in ρ_τ, we do not denote this explicitly, because quantification is restricted by N in every predicate we define.

The transitions relation ρ is constructed in the following way:

$$\rho = \left(\bigvee_{\tau \in T_C} \rho_\tau \right) \quad q \quad \left(Qi. \bigvee_{\tau \in T_U} \rho_\tau(i) \right) \quad ,$$

where $q \in \{\vee, \wedge\}$ and $Q \in \{\exists, \forall\}$. □

The quantor Q of a transition relation ρ determines the *execution mode* of the user processes: Whether the system executes in an interleaving manner, where only one process takes a transition, or whether all processes have to make a step simultaneously. The boolean connective q does the same for the controller connected to the user processes, we call this the *control mode*.

The image one should have in mind is

$$C \tilde{\|} (U_1 \| \ldots \| U_n) \quad ,$$

a controller running in parallel with a system of parallel user processes, where both parallel operators range independently over synchronous and asynchronous parallel composition.

The choice which of the two transition types are used determines in which way information of other processes may be accessed for transition enabling conditions. Either one can restrict all processes to be in certain states (using the $\forall$ quantifier), or one can enforce that there is at least one other process in a certain set of states (using the $\exists$ quantifier). We call these restrictions on the information of other components the *observability*, defining what a component is able to observe. Syntactic restrictions may further reduce the observability.

Before we give an overview over the classes defined in that way, we show which transition predicates are defined by transitions given in the guarded command style.

Definition 3.26
Assume that each transition $\tau \in T_C \cup T_U$ is given by a predicate

$$\rho_\tau(i) = g(i, \bar{\jmath}) \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y$$

where g is the whole first part of the transition predicate up to the assignment part. Define for each set of variables $V \subseteq \mathcal{V}$ the predicate

$$UC(V) \stackrel{\text{def}}{=} \bigwedge_{v \in V} v' = v \quad ,$$

expressing that no variable from V is modified. For each variable set $V \subseteq \mathcal{V}_X$

$$UC(V, i) \stackrel{\text{def}}{=} \bigwedge_{v \in V} v'[i] = v[i]$$

expresses that none of process i'th local V variables change their values, and

$$UC(V, \neq i) \stackrel{\text{def}}{=} \forall_N l : l \neq i \Rightarrow \bigwedge_{v \in V} v'[l] = v[l]$$

describes that all the other processes remain in the same local state with respect to the V variables.

Note that $\bar{x}[i] := \bar{c}_x$ is a set of assignments $x[i] := c$, where each x in the list $\bar{x}$ is an array variable, and $\bar{y} := \bar{c}_y$ is a set of assignments for normal variables.

By Definition 3.25, either the first assignment set is empty or the latter is. Each control transition can be translated to the predicate

$$\begin{aligned}\hat{\rho}_\tau(\mathcal{V}, \mathcal{V}') \stackrel{\text{def}}{=}\ & g(\bar{j}) \wedge UC(\mathcal{V}_Y \setminus \bar{y}) \\ & \wedge \textstyle\bigwedge_{y:=c \in \bar{y}:=\bar{c}} y' = c\end{aligned}$$

Note that this predicate on its own does not require array variables to be unmodified. Since such a predicate is to be used for several execution modes, conditions for these variables are introduced later.

Similarly, a user process transition corresponds to

$$\begin{aligned}\hat{\rho}_\tau(i, \mathcal{V}, \mathcal{V}') \stackrel{\text{def}}{=}\ & g(i, \bar{j}) \wedge UC(\mathcal{V}_X \setminus \bar{x}, i) \\ & \wedge \textstyle\bigwedge_{x[i]:=c \in \bar{x}[i]:=\bar{c}} x'[i] = c\end{aligned}$$

The transition relation is built up by these predicates as follows. Recall the transition relation generic shape

$$\rho = \left(\bigvee_{\tau \in T_C} \rho_\tau \right) \quad q \quad \left(Qi. \bigvee_{\tau \in T_U} \rho_\tau(i) \right) \quad ,$$

where $q \in \{\vee, \wedge\}$ and $Q \in \{\exists, \forall\}$.

The corresponding transition relation $\hat{\rho}$ built up by the translated predicates is

$$\hat{\rho} = \left(\bigvee_{\tau \in T_C} \hat{\rho}_\tau \wedge UC(V_X) \right) \vee \left(UC(V_Y) \wedge \hat{Q}(i) \wedge \bigvee_{\tau \in T_U} \hat{\rho}_\tau(i) \right) \quad ,$$

in case $q = \vee$, and

$$\hat{\rho} = \left(\bigvee_{\tau \in T_C} \hat{\rho}_\tau \right) \wedge \left(\hat{Q}(i) \wedge \bigvee_{\tau \in T_U} \hat{\rho}_\tau(i) \right) \quad ,$$

otherwise. The process quantification $\hat{Q}$ abbreviates

$$\hat{Q}(i) \stackrel{\text{def}}{=} \begin{cases} \forall i.tt & \text{if } Q = \forall \\ \exists i.UC(\mathcal{V}_Y, \neq i) & \text{if } Q = \exists \end{cases}$$

□

We will mainly investigate two independent parameters. The first one is the execution mode, e.g., synchronous or asynchronous execution, both for the user processes and the connected controller. The second one is the *observability* of controller and the user processes, e.g., one could restrict the system such that the controller is in fact only an observer which has no possibility to influence the user processes.

3.4.1 Basic System Classes

Each of the two execution modes for processes (synchronous resp. asynchronous) can be combined with the restriction to only use $\forall$-transitions resp. only use $\exists$-transitions. This gives us four basic system classes having the following transition relations, ignoring the control part for the moment. These classes are listed in Table 3.1. The user process transitions begin with

Table 3.1: Basic system classes

Σ_0:	$\exists_N i. \bigvee_{\tau \in T_U} \exists_N \bar{j} : \phi_\tau(i) \wedge \bar{j} \neq i \wedge \psi_\tau(i, \bar{j}) \rightarrow \bar{x}[i] := \bar{c}_\tau$
Σ_1:	$\exists_N i. \bigvee_{\tau \in T_U} \forall_N \bar{j} : \phi_\tau(i) \wedge (\bar{j} \neq i \Rightarrow \psi_\tau(i, \bar{j})) \rightarrow \bar{x}[i] := \bar{c}_\tau$
Π_0:	$\forall_N i. \bigvee_{\tau \in T_U} \forall_N \bar{j} : \phi_\tau(i) \wedge (\bar{j} \neq i \Rightarrow \psi_\tau(i, \bar{j})) \rightarrow \bar{x}[i] := \bar{c}_\tau$
Π_1:	$\forall_N i. \bigvee_{\tau \in T_U} \exists_N \bar{j} : \phi_\tau(i) \wedge \bar{j} \neq i \wedge \psi_\tau(i, \bar{j}) \rightarrow \bar{x}[i] := \bar{c}_\tau$

the second quantor in this description.

The *controller transitions* in each class are in principle the same, but we require them to be closed, such that there is no free variable i, and we substitute the $\bar{j} \neq i$ sub-formula by tt. Consequently, these transitions refer only to global variables and to user processes in the usual quantified manner. Moreover, control transitions modify global variables, hence, the assignment part has the form $\bar{y} := \bar{c}_\tau$ instead of $\bar{x}[i] := \bar{c}_\tau$.

As example we only give a control transition of the Σ_0 and Π_1 model:

$$\exists_N \bar{j} : \phi_\tau \wedge \psi_\tau(\bar{j}) \rightarrow \bar{y} := \bar{c}_\tau$$

With $\Sigma_0 \cup \Sigma_1$ (resp. $\Pi_0 \cup \Pi_1$) we denote the class of systems where each transition is either a $\exists$-transition or a $\forall$-transition.

Each of these classes can be combined with both *control modes*, either synchronous or asynchronous.

3.4.2 Observability

The last parameter which we vary are syntactic constraints on the transitions reducing the observability of controller resp. user processes. May the enabling condition of a transition depend on both user and controller variables or only on one of them? A very tight coupling, for example, would allow both user and controller processes to depend on both user and controller variables, whereas a weaker coupling could allow control transitions only to depend on its own variables.

We distinguish the following classes:

Mutual observability: A transition depends on both $\mathcal{V}_X$ and $\mathcal{V}_Y$. The transitions are exactly the ones given in Section 3.4.1. In control theory, this type of control is also called *closed loop control.*

Non-reactive control: The controller transitions does not refer to array variables. Hence, the control only uses its own variables. This type of control is also called *open loop control* in control theory.

Observer: The controller has no influence on the user processes, but is able to observe them.

Independent: Controller transitions are only referring to global variables, and user transitions only to array variables.

No Control: There are no global variables, and no control transitions.

These observability constraints are summarized in Table 3.2. In this table, an x refers to the array variables $\mathcal{V}_X$ and y refers to $\mathcal{V}_Y$ variables. The line $x, y \to x'$ means that the new values of $\mathcal{V}_X$ variables may depend on both $\mathcal{V}_X$ and $\mathcal{V}_Y$ variables.

3.4.3 Further Variations

There are several further syntactic restrictions one can think of. The first one slightly simplifies the classes we define.

Table 3.2: Observability constraints

Observability	Transition form
mutual observability	$x, y \to x'$ $x, y \to y'$
non-reactive control	$x, y \to x'$ $y \to y'$
observer	$x \to x'$ $x, y \to y'$
independent	$x \to x'$ $y \to y'$
no control	$x \to x'$

Definition 3.27 (Weak system class)
For each system class $\mathcal{C}$, removing the $\bar{j} \neq i$ part (one can also replace it by tt) leads to a new system class, called *weak* $\mathcal{C}$. □

Hence, a process of a weak system can only quantify over all processes including itself. Intuitively, such a process is not able to observe processes in the same state as its own. For the $\Pi_0 \cup \Pi_1$ class, adding this constraint makes the undecidable class decidable, see Section 4.6 and Section 4.7.

Remark 3.28
If only one j variable is quantified, this subclass can also be introduced by restricting transitions to guards of the form

$$\phi(i) \wedge \psi(i,i) \wedge (j \neq i \Rightarrow \psi(i,j)) \ .$$

We will use this observation in Chapter 4. ♣

We list a few more possibilities to vary system classes here:

1. All transitions refer only to control variables, hence there are no quantifications over indices in the transition predicates.

2. One can loose the constraints on the index terms, allowing, e.g., $x[i+1] := tt$ in the transition's assignment part.

3. Transition guards may use the successor function (addition of 1) and the predecessor function (subtraction of 1) in index terms. Thus, every process can check the state of neighbored processes.

4. Index arithmetic and index term comparisons may be used in the transitions.

5. Use different forms of observability as given in Table 3.2, e.g., memoryless control with user transitions of the form $x, y \to x'$ and control transitions $x \to y'$.

3.5 One vs. Multiple Parameters and Process Classes

We will argue in this section that although we restrict ourselves to one parameter N, and hence, one process class for the user processes, many system classes with multiple parameters known from the literature can be modeled in our framework.

We start by shortly introducing the system classes defined in [EK00] and showing how to represent them in our framework.

The systems in [EK00] are built from several process classes $U_1, \ldots, U_k$, where each U_i is a simple transition system template consisting of a set of states and a set of transitions between these states guarded by some expressions over the states reached in other processes. An instance of the system

$$U_1^{n_1} \| \ldots \| U_k^{n_k} \ ,$$

for some natural numbers $n_1, \ldots, n_k \geq 1$, is composed of n_i copies of the template process U_i for $1 \leq i \leq k$.

The guards are restricted to one of the following cases:

- Disjunctive guards, where a process i in the class U_l may have transition guards of the form

$$\bigvee_{r \neq i} (a_l^r + \cdots + b_l^r) \vee \bigvee_{j \neq l} \bigvee_{k \in [1 \ldots n_j]} (a_j^k + \cdots + b_j^k)$$

  where $a_l^r + \cdots + b_l^r$ denotes that process r in class U_l may be in one of the local states $a, \ldots, b$.

- Conjunctive guards with initial state, where process i in class U_l is allowed to have transition guards

$$\bigwedge_{r \neq i} (i_l^r + a_l^r + \cdots + b_l^r) \vee \bigwedge_{j \neq l} \bigwedge_{k \in [1 \ldots n_j]} (i_j^k + a_j^k + \cdots + b_j^k)$$

  where i_l^r is the initial state of process r in class U_l.

Properties of the following form are investigated:

- $\bigwedge_{i_l} \mathsf{A}\phi(i_l)$ and $\bigwedge_{i_l} \mathsf{E}\phi(i_l)$, where i_l ranges over U_l processes, and $\phi(i_l)$ is an LTL formula. Here, A and E are *path quantifiers*, expressing that every path (respectively one of the paths) fulfills the LTL property $\phi(i_l)$.

- $\bigwedge_{i_l \neq j_l} \mathsf{A}\phi(i_l, j_l)$ and $\bigwedge_{i_l \neq j_l} \mathsf{E}\phi(i_l, j_l)$, where i_l, j_l range over pairs of distinct U_l processes.

- $\bigwedge_{i_l, j_m} \mathsf{A}\phi(i_l, j_m)$ and $\bigwedge_{i_l, j_m} \mathsf{E}\phi(i_l, j_m)$, where i_l ranges over U_l and j_m ranges over U_m.

Assume that we have k classes U_j, for $1 \leq j \leq k$, each of them having states L_j, an initial state i_j, and a transition relation $T_j \subseteq L_j \times G \times L_j$, where G denotes the set of all guards. We assume that $L_i \cap L_j = \emptyset$ for $i \neq j$.

Let $\mathcal{V} = \{s, c\}$ with type$(l) =$ array $[0 \ldots N]$ of $L_1 \cup \ldots \cup L_k$ and type$(c) =$ array $[0 \ldots N]$ of $\{1, \ldots, k\}$ (the latter is used to store the process' class).

Intuitively, each process chooses initially its class and starts in the corresponding state initial state. θ only ensures that there is at least one process in every class and that each process starts in valid initial state.

In fact, this modeling is slightly different from the original one, since the order of the processes is completely random here, e.g., it is possible that in a certain trace of an instance, process 1 is in class U_2, process 2 in class U_1, and process 3 in class U_2 again. But since the arrangement (or topology) of the processes has no influence in the system behavior, this works quite well.

Note that the distribution of the processes among the process classes usually changes from trace to trace, even for a fixed instance of the system.

The initial state predicate is defined by

$$\begin{aligned}\theta \stackrel{\text{def}}{=} \exists_N i_1, \ldots, i_k . \forall_N i : & c[i] \in \{1, \ldots, k\} \wedge \textstyle\bigwedge_{j=1}^{k} c[i_j] = j \\ & \wedge \textstyle\bigwedge_{j=1}^{k} (s[i] = i_j \Leftrightarrow c[i] = j) \ .\end{aligned}$$

For each transition $\tau = (l, g, l')$ of a class U_l with a disjunctive guard

$$\bigvee_{r \neq i} (a_l^r + \cdots + b_l^r) \vee \bigvee_{j \neq l} \bigvee_{k \in [1 \ldots n_j]} (a_j^k + \cdots + b_j^k)$$

we define

$$\begin{aligned}\rho_\tau(i) \stackrel{\text{def}}{=} \exists_N j . s[i] = l \wedge ((c[j] = l \wedge s[j] \in \{a_l, \ldots, b_l\}) \\ \vee \textstyle\bigvee_{j \neq l} (c[j] = j \wedge s[j] \in \{a_j, \ldots, b_j\}))\end{aligned}$$

(it is straightforward to express formula parts like $s[j] \in \{a_j, \ldots, b_j\}$ as a predicate).

If the transition has a conjunctive guard

$$\bigwedge_{r \neq i}(i_l^r + a_l^r + \cdots + b_l^r) \vee \bigwedge_{j \neq l} \bigwedge_{k \in [1 \ldots n_j]} (i_j^k + a_j^k + \cdots + b_j^k)$$

define

$$\begin{aligned}\rho_\tau(i) \stackrel{\text{def}}{=} \forall_N j.s[i] = l \wedge ((c[j] = l \Rightarrow s[j] \in \{i_l, a_l, \ldots, b_l\}) \\ \wedge \textstyle\bigwedge_{j \neq l}(c[j] = j \Rightarrow s[j] \in \{i_j, a_j, \ldots, b_j\}))\ .\end{aligned}$$

It remains to show that also the properties can be expressed in our framework. Since we restrict ourselves to LTL, it is not possible to express E properties, so we only investigate the A properties. The first type is a special case of the second, so we only show expressibility of the latter two types.

- Properties for two distinct processes in one class U_l, given by the formula $\bigwedge_{i_l \neq j_l} \mathsf{A}\phi(i_l, j_l)$, can be expressed by

$$\forall_N i_1, i_2 : i_1 \neq i_2 \Rightarrow (c[i_1] = l \wedge c[i_2] = l \Rightarrow \phi(i_1, i_2))\ .$$

- For properties referring to two classes U_l, U_m, $\bigwedge_{i_l, j_m} \mathsf{A}\phi(i_l, j_m)$, the following formula can be chosen:

$$\forall_N i_1, i_2 : i_1 \neq i_2 \Rightarrow (c[i_1] = l \wedge c[i_2] = m \Rightarrow \phi(i_1, i_2))\ .$$

The next two theorems show the close relationship of parameterized systems with multiple parameters to the corresponding one-parameter systems as given above.

Theorem 3.29
For all $n_1, \ldots, n_k \geq 1$,

$$\mathcal{S}'(n_1 + \ldots + n_k) \models \phi$$

implies

$$\mathcal{S}(n_1, \ldots, n_k) \models \phi\ .$$

Proof: Assume that $\mathcal{S}'(n_1 + \ldots + n_k) \models \phi$ holds, and assume that $\alpha \in [\![\mathcal{S}(n_1, \ldots, n_k)]\!]$. Then, there exists a corresponding computation of $\mathcal{S}'(n_1 + \ldots + n_k)$ which starts in a state where the first n_1 processes "choose" the class U_1, such that $c[i] = 1$ for these processes, the next n_2 processes choose class 2, and so on. Each α step can be mimicked by a corresponding $\mathcal{S}'$ step. This results in a computation $\alpha' \in \mathcal{S}'(n_1 + \ldots + n_k)$. Hence, since $\alpha' \models \phi$, also $\alpha \models \phi$ (we omit the property translation here). ∎

Theorem 3.30
For all $n \geq k$:

$$\forall n_1, \ldots, n_k \geq 1 : (\sum_{i=1}^{k} n_i) = n \Rightarrow \mathcal{S}(n_1, \ldots, n_k) \models \phi$$

implies

$$\mathcal{S}'(n) \models \phi \ .$$

Proof: Similar to the proof of Theorem 3.29. Note that in each initial state of $\mathcal{S}'(n)$, each process "chooses" exactly one process class, and for every distribution of processes among the classes, there exists a corresponding initial state. ■

3.6 Technical Notions

This section introduces some technical notions that are frequently used in the following chapters.

Definition 3.31 (Local and Global State)
A *global state* of $\mathcal{S}(n)$ is a well-typed function $\sigma_G : \mathcal{V}_Y \to \mathbb{V}$, where $\mathcal{V}_Y$ is the set of all normal variables of $\mathcal{S}$. We also use the name *control state*, and call these variables *control variables* or *global variables*.

A *local state* is a well-typed function $\sigma_L : \mathcal{V}_X \to \mathbb{V}$, where $\mathcal{V}_X$ is the set of array variables of $\mathcal{S}$. In this context, σ_L is called well-typed, if $\sigma_L(x) \in \text{type}(x)$ for all $x \in \mathcal{V}_X$.

The set of all global states is denoted by Σ_G, the set of all local states by Σ_L.

The *size* of a system $|\mathcal{S}|$ is defined as the product of the cardinality of the local and global state space, i.e., $|\mathcal{S}| \stackrel{\text{def}}{=} |\Sigma_G| \cdot |\Sigma_L|$. □

The names *local* and *global* correspond to the point of view explained in Section 3.4: A parameterized system consists of a controller process controlling the normal variables, and some user processes controlling the array variables. Therefore, all the entries of the arrays in a position i correspond to the state of user process i, and can be seen as a *local state* of process i.

Of course, this is just an intuition, and one is not forced to model only such systems. Whatever fits in our definition and to the restrictions we use, is a parameterized system (in a certain class).

Remark 3.32
For each parameterized system $\mathcal{S}$,

- $|\Sigma_G|$ and $|\Sigma_L|$ are finite,
- its size $|\mathcal{S}|$ is finite.

♣

Definition 3.33 (State projection)
For each $0 \leq j < n$ define the *state projection on process* j as the function $\pi_j : \Sigma \to \Sigma_L$ such that

$$\pi_j(\sigma)(b) = \sigma(b)(j) \text{ for all } b \in \mathcal{V}_L \text{ and } \sigma \in \Sigma \ .$$

Moreover, for each $\sigma \in \Sigma$, the *state projection on the global component* π_G is defined as

$$\pi_G(\sigma) \stackrel{\text{def}}{=} \sigma|_{\mathcal{V}_G} \ .$$

For a list of natural numbers $i_1, \ldots, i_l$ define $\pi_{G,i_1,\ldots,i_l} : \Sigma \to \Sigma_G \times (\Sigma_L)^l$ by

$$\pi_{G,i_1,\ldots,i_l}(\sigma) \stackrel{\text{def}}{=} (\pi_G(\sigma), \pi_{i_1}(\sigma), \ldots, \pi_{i_l}(\sigma))$$

for all $\sigma \in \Sigma$. □

This definition is used to simultaneously take local components from a state for a couple of processes.

Definition 3.34 (Induced local and global state)
Each state $\sigma \in \Sigma_n$ *induces a global state* (namely $\pi_G(\sigma)$) and a set of *local states* as follows. A *local state* $\sigma_L : \mathcal{V}_L \to \mathbb{V}$ *is induced* by state σ and index $j \in [n]$ if and only if $\pi_j(\sigma) = \sigma_L$.

An induced global state (resp. an induced local state) of a state σ is also called the *global component* (resp. a *local component*) of σ.

For a set of indices $I \subseteq [n]$, we define

$$\text{Localstates}(\sigma, I) \stackrel{\text{def}}{=} \{\pi_j(\sigma) \mid j \in I\} \ .$$

The case when we want to omit exactly one process j is denoted by

$$\text{Localstates}(\sigma, \neq j) \stackrel{\text{def}}{=} \text{Localstates}(\sigma, [n] \setminus \{j\}) \ ,$$

and the set of all local states induced by a state σ is

$$\text{Localstates}(\sigma) \stackrel{\text{def}}{=} \text{Localstates}(\sigma, [n]) \ .$$

Similarly, Globalstates(σ) is the global state induced by σ.

These functions are also expanded on sets and sequences of states. For a set of states $S \subseteq \Sigma$ define

$$\text{Localstates}(S, I) \stackrel{\text{def}}{=} \bigcup_{\sigma \in S} \text{Localstates}(\sigma, I)$$

and

$$\text{Globalstates}(S) \stackrel{\text{def}}{=} \{\ \text{Globalstates}(\sigma) \mid \sigma \in S\ \}\ .$$

For a sequence of states $\alpha \in \Sigma^{\underline{\omega}}$ we define

$$\text{Localstates}(\alpha, I) \stackrel{\text{def}}{=} \text{Localstates}(\{\ \alpha(i) \mid i < |\alpha|\ \}, I)\ .$$

The latter gives all local states occurring in the sequence, *not* the sequence of the induced local states as one might expect. The function Globalstates is expanded analogously. □

Definition 3.35 (Induced local and global trace)
Let $\alpha \in \Sigma^{\underline{\omega}}$ be a sequence of states of a a system instance. The projections π_j, π_G, and $\pi_{G,i_1,\ldots,i_l}$ are expanded to sequences of states such that for all $k < |\alpha|$:

- $\pi_j(\alpha)(k) \stackrel{\text{def}}{=} \pi_j(\alpha(k))$,
- $\pi_G(\alpha)(k) \stackrel{\text{def}}{=} \pi_G(\alpha(k))$, and
- $\pi_{G,i_1,\ldots,i_l}(\alpha)(k) \stackrel{\text{def}}{=} (\pi_G(\alpha(k)), \pi_{i_1}(\alpha(k)), \ldots, \pi_{i_l}(\alpha(k)))$.

The sequences $\pi_j(\alpha)$ are called the *induced local traces*, and analogously, the sequence $\pi_G(\alpha)$ is called the *induced global trace*. □

In the following sections we sometimes need the state variant, which is a state modified only in the global or in some of the local components.

Definition 3.36 (State variant)
Let $\sigma \in \Sigma_n$ be a state, and σ_G be a global state (assigning values only to normal variables), σ_L a local state, and $i < n$. The state variants $(\sigma : G \mapsto \sigma_G)$ and $(\sigma : i \mapsto \sigma_L)$ are defined by:

$$(\sigma : G \mapsto \sigma_G)(v) \stackrel{\text{def}}{=} \begin{cases} \sigma_G(v) & \text{if } v \in \mathcal{V}_Y \\ \sigma(v) & \text{if } v \notin \mathcal{V}_Y \end{cases}$$

$$(\sigma : i \mapsto \sigma_L)(v) \stackrel{\text{def}}{=} \begin{cases} (\sigma(v) : i \mapsto \sigma_L(v)) & \text{if } v \in \mathcal{V}_X \\ \sigma(v) & \text{if } v \notin \mathcal{V}_X \end{cases}$$

where $(\sigma(v) : i \mapsto \sigma_L(v))$ is the function variant as defined in Definition 2.5 (recall that $\sigma(x)$ is a function $[n] \to \mathbb{V}$ for an array variable $x \in \mathcal{V}_X$).

The state variant which is modified in both the global and one local component is defined by

$$(\sigma : G, i \mapsto \sigma_G, \sigma_L) \stackrel{\text{def}}{=} ((\sigma : G \mapsto \sigma_G) : i \mapsto \sigma_L) \ .$$

□

It is easy to see that in the last definition, the order of application of both variants has no influence in the outcome.

Example 3.37
The state σ given in Example 3.6 induces the following local and global states:

$$\text{Localstates}(\sigma) = \{(l \mapsto 1, c \mapsto \mathit{ff}), (l \mapsto 2, c \mapsto \mathit{ff}), (l \mapsto 3, c \mapsto \mathit{tt})$$

induced by index 1, 2, and 3, respectively. There is only one global state induced, namely $\text{Globalstates}(\sigma) = \pi_G(\sigma) = (s \mapsto 0)$. ♣

Next we define operators for appending local and global state components to a finite state sequence. They will be used to construct system traces by successively adding user process steps or controller steps.

Definition 3.38
For $k \geq 1$, define $\text{add}_{\text{L}} : \Sigma^{\mathcal{F}} \times \mathbb{N}^k \times (\Sigma_L)^k \to \Sigma^{\mathcal{F}}$ by

$$\text{add}_{\text{L}}(\alpha, (i_1, \ldots, i_k), (\sigma_1, \ldots, \sigma_k)) \stackrel{\text{def}}{=} \beta$$

such that

- $\beta|_h = \alpha$,
- $|\beta| = h + 1$,
- $\pi_G(\beta(h)) = \pi_G(\beta(h-1))$,
- $\pi_j(\beta(h)) = \pi_j(\beta(h-1))$ for $j \notin \{i_1, \ldots, i_k\}$, and
- $\pi_{i_j}(\beta(h)) = \sigma_j$ for $1 \leq j \leq k$,

where $h = |\alpha|$. In case $k = 1$ we write $\text{add}_{\text{L}}(\alpha, i, \sigma)$. □

Definition 3.39
Define $\mathrm{add_G} : \Sigma^{\mathcal{F}} \times \Sigma_G \to \Sigma^{\mathcal{F}}$ by

$$\mathrm{add_G}(\alpha, i, \sigma) \stackrel{\mathrm{def}}{=} \beta$$

such that

- $\beta|_h = \alpha$,
- $|\beta| = h + 1$,
- $\pi_G(\beta(h)) = \sigma$, and
- $\pi_j(\beta(h)) = \pi_j(\beta(h-1))$ for all j,

with $h = |\alpha|$. □

3.7 General System Properties

We prove some general properties common to all system classes in this section. They are implied by the restrictions imposed in Section 3.4. We start with the observation that validity of the initial state predicate only depends on the global state and the set of local states. First, we give a result for more restricted initial state predicates.

Lemma 3.40
Let $\mathcal{S}$ be a parameterized system having an initial state condition θ without existential quantification that fulfills Restrictions 2 and 3. For all $n, \bar{n} \in \mathbb{N}$ and $\sigma \in \Sigma_n, \bar{\sigma} \in \Sigma_{\bar{n}}$ with

- Localstates($\bar{\sigma}$) $\subseteq$ Localstates(σ) and
- $\pi_G(\sigma) = \pi_G(\bar{\sigma})$,

the following holds:

$$\sigma \models \theta \text{ implies } \bar{\sigma} \models \theta \ .$$

Proof: From the assumption we know that the initial state condition has the form $\theta = \forall_N i : \phi(i)$, where ϕ is quantifier-free. We assume that $n, \bar{n}, \sigma, \bar{\sigma}$ are given as above such that $\sigma \models \theta$.

We have to show for each $\bar{\jmath} < \bar{n}$, $\bar{\sigma} \models \phi[\bar{\jmath}/i]$. So let us assume that $\bar{\jmath} < \bar{n}$ is given.

Then, the first assumption implies that there exists a $j < n$ such that $\pi_j(\sigma) = \pi_{\bar{j}}(\bar{\sigma})$.

Since i is the only index variable in ϕ, the local state of process j in state σ equals that of process $\bar{j}$ in $\bar{\sigma}$, the global state components are identical, and ϕ is quantifier free, all sub-expressions from ϕ evaluate to the same values in $\bar{\sigma}$ for index $\bar{j}$ as they do in σ for index j (simple structural induction proof).

Consequently, $\bar{\sigma} \models \phi[\bar{j}/i]$, which proves the lemma. ■

A similar result can be proved for arbitrary initial state predicates.

Lemma 3.41
Let $\mathcal{S}$ be a parameterized system, let $\theta = \exists_N i_1, \ldots, i_k . \forall_N i : \phi_\theta(i, i_1, \ldots, i_k)$ be the initial state condition that fulfills Restrictions 2 and 3. Let $n, \bar{n} \in \mathbb{N}$, $\sigma \in \Sigma_n, \bar{\sigma} \in \Sigma_{\bar{n}}$, and $n_1, \ldots, n_k < n$. If

- $\sigma \models \forall_N i : \phi_\theta(i, n_1, \ldots, n_k)$,
- Localstates$(\bar{\sigma}) \subseteq$ Localstates(σ),
- $\pi_G(\sigma) = \pi_G(\bar{\sigma})$, and
- for all $1 \leq i \leq k$, $\pi_{n_i}(\sigma) \in$ Localstates$(\bar{\sigma})$,

then

$$\bar{\sigma} \models \theta \ .$$

Proof: By the fourth requirement above, there exist n'_j such that $\pi_{n'_j}(\bar{\sigma}) = \pi_{n_j}(\sigma)$. Now, let $j' < \bar{n}$. Then, by the second requirement, there exists $j < n$ such that $\pi_{j'}(\bar{\sigma}) = \pi_j(\sigma)$. Since $\sigma \models \phi(j, n_1, \ldots, n_k)$ is valid, we conclude that also $\bar{\sigma} \models \phi(j', n'_1, \ldots, n'_k)$. This can be shown by a simple structural induction proof: since corresponding local states are identical, all sub-terms are evaluated to the same values.

Consequently, $\bar{\sigma} \models \theta$. ■

Corollary 3.42
Let $\mathcal{S}$ be a parameterized system with initial state condition θ that fulfills Restrictions 2 and 3. For all $n, \bar{n} \in \mathbb{N}$ and $\sigma \in \Sigma_n, \bar{\sigma} \in \Sigma_{\bar{n}}$ with

- Localstates$(\bar{\sigma}) =$ Localstates(σ) and
- $\pi_G(\sigma) = \pi_G(\bar{\sigma})$,

the following holds:

$$\sigma \models \theta \text{ implies } \bar{\sigma} \models \theta \ .$$

Proof: Lemma 3.41. ■

The next lemma characterizes the system instances for which θ is satisfiable, if there are any. These instances always have some behavior.

Lemma 3.43
Let $\mathcal{S}$ be a parameterized system with initial state condition θ that fulfills Restrictions 2 and 3.

(1) If there exists $\sigma \in \Sigma_n$ with $\sigma \models \theta$, then there exists $\sigma' \in \Sigma_{n+1}$ with $\sigma' \models \theta$.

(2) There exists $n \geq 1$ such that $\sigma \models \theta$ for some $\sigma \in \Sigma_n$ if and only if there exists $\sigma' \in \Sigma_{|\Sigma_L|}$ with $\sigma' \models \theta$.

(3) Let l be the numbers of existential quantifications in θ. Then there exists $n \geq 1$ such that $\sigma \models \theta$ for some $\sigma \in \Sigma_n$ if and only if there exists $\sigma' \in \Sigma_l$ with $\sigma' \models \theta$.

Proof: For (1), let us assume that $\sigma \models \theta$. One simply has to choose an initial state σ' such that $\pi_j(\sigma) = \pi_j(\sigma')$, the control components are identical, and then "copy" one local state, say that of process 0, for process n such that $\pi_n(\sigma') = \pi_0(\sigma)$. Corollary 3.42 then implies that $\sigma' \models \theta$.

Property (2) follows by observing that an initial state for every subset $M \subseteq \Sigma_L$, $M \neq \emptyset$, can be chosen. So if there is any system instance $\mathcal{S}(n)$ with $\sigma \in \Sigma_n$, $\sigma \models \theta$, then Localstates$(\sigma) \subseteq \Sigma_L$, and one can choose a corresponding state $\sigma' \in \Sigma_{|\Sigma_L|}$ such that Localstates(σ') = Localstates(σ), and such that the control components are identical. Again by Corollary 3.42, $\sigma' \models \theta$.

For property (3), assume that σ fulfills

$$\theta = \exists_N i_1, \ldots, i_l . \forall_N i : \phi(i, i_1, \ldots, i_l) \ ,$$

where ϕ is a quantifier-free formula without comparisons of index terms. The other implication holds trivially. Then, since $\sigma \models \theta$, one can find $n_1, \ldots, n_l$ such that $\sigma \models \phi(h, n_1, \ldots, n_l)$, for all $h < \sigma(N)$.

Choose $\sigma' \in \Sigma_l$ such that $\pi_j(\sigma') = \pi_{n_{j+1}}(\sigma)$ for $0 \leq j \leq l-1$, and $\pi_G(\sigma') = \pi_G(\sigma)$ (obviously, such a state exists).

Let $h' < \sigma'(N) = l$. Then, for $h \stackrel{\text{def}}{=} n_{h'+1}$, $\sigma \models \phi(h, n_1, \ldots, n_l)$, and $\pi_h(\sigma) = \pi_{h'}(\sigma')$. By a straightforward structural induction proof we conclude $\sigma' \models \phi(h', 0, \ldots, l-1)$ (intuitively, all sub-terms are evaluated to the same values, since corresponding local states are identical). ■

We note that it might be possible that θ is satisfiable for some state of a system instance *smaller* than l. In that case, some of the i_j variables have the same value.

Next we observe that the validity of an LTL formula $\phi(n_1, \ldots, n_k)$ only depends on the local traces of indices $n_1, \ldots, n_k$ and the values of the normal variables.

Lemma 3.44
For all LTL formulae $\phi(i_1, \ldots, i_k)$, $n, n' \in \mathbb{N}$, $\alpha \in \Sigma^{\omega}_{\mathcal{V},n}$, $\alpha' \in \Sigma^{\omega}_{\mathcal{V},n'}$, $n_1, \ldots, n_k < n$, and $n'_1, \ldots, n'_k < n'$ we have that

- $\alpha \models \phi(n_1, \ldots, n_k)$,
- $\pi_{n_j}(\alpha) = \pi_{n'_j}(\alpha')$ for $j = 1, \ldots, k$, and
- $\pi_G(\alpha) = \pi_G(\alpha')$

implies

$$\alpha' \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: By structural induction. Intuitively, for each state in the sequence α, each sub-term of ϕ is evaluated to the same value as the corresponding sub-term evaluated in the corresponding state in α', since the only index terms occurring in ϕ are $i_1, \ldots, i_k$. ■

Corollary 3.45
For all LTL formulae $\phi(i_1, \ldots, i_k)$, $n, n' \in \mathbb{N}$, $\alpha \in \Sigma^{\omega}_{\mathcal{V},n}$, $\alpha' \in \Sigma^{\omega}_{\mathcal{V},n'}$, $n_1, \ldots, n_k < n$, and $n'_1, \ldots, n'_k < n'$ we have that

$$\pi_{n_j}(\alpha) = \pi_{n'_j}(\alpha') \text{ for } j = 1, \ldots, k \text{ and } \pi_G(\alpha) = \pi_G(\alpha')$$

implies

$$\alpha \models \phi(n_1, \ldots, n_k) \Leftrightarrow \alpha' \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: Lemma 3.44. ■

The next lemma states that each process can "mimic" the behavior of any other process, so processes are interchangeable.

Lemma 3.46 (Process symmetry)
Let $\mathcal{C} \in \{\Sigma_0, \Sigma_1, \Sigma_0 \cup \Sigma_1, \Pi_0, \Pi_1, \Pi_0 \cup \Pi_1\}$ be a system class, $\mathcal{S}(n)$ be a $\mathcal{C}$ system instance (with asynchronous or synchronous control), and $l, l' < n$. If $\alpha \in [\![\mathcal{S}(n)]\!]$, and $\alpha \models \phi(n_1, \ldots, n_k)$, then there exists $\beta \in [\![\mathcal{S}(n)]\!]$ such that

- $\beta \models \phi(n'_1, \ldots, n'_k)$, where the n'_j originate from the n_j by swapping l and l' (in case l or l' are members of the list $n_1, \ldots, n_k$),
- $\pi_j(\alpha) = \pi_j(\beta)$ for $j \neq l$ and $j \neq l'$,
- $\pi_G(\alpha) = \pi_G(\beta)$,
- $\pi_l(\alpha) = \pi_{l'}(\beta)$, and $\pi_{l'}(\alpha) = \pi_l(\beta)$.

Proof: Straightforward. β is constructed from α in the obvious way, by swapping the local traces of l and l'. This is again a valid trace: whenever process l is "chosen" in a step, choose now l' and vice versa. Note that quantification over processes refers in the very same way to all processes. The initial state predicate holds for β due to Corollary 3.42. ■

This result can be used to show that verification of properties with universal quantification over processes can often be reduced to one particular variable instantiation. This symmetry reduction "up to permutation" is in the spirit of the work presented in [ES93, ES96, CJEF96, ET99].

Lemma 3.47 (Symmetry reduction)
Let $\mathcal{S}(n)$ be a $\mathcal{C}$ system instance, $n \geq k$. Then,

$$\mathcal{S}(n) \models \forall_d \phi(i_1, \ldots, i_k) \text{ if and only if } \mathcal{S}(n) \models \phi(0, \ldots, k-1) \ .$$

Proof:

$$\begin{array}{ll} & \mathcal{S}(n) \not\models \forall_d \phi(i_1, \ldots, i_k) \\ \text{iff} & \exists n_1, \ldots, n_k < n, n_i \neq n_j, \exists \alpha \in [\![\mathcal{S}(n)]\!] : \alpha \not\models \phi(n_1, \ldots, n_k) \\ \text{iff} & \exists \beta \in [\![\mathcal{S}(n)]\!] : \beta \not\models \phi(0, \ldots, k-1) \qquad \text{(Lemma 3.46)} \\ \text{iff} & \mathcal{S}(n) \not\models \phi(0, \ldots, k-1) \end{array}$$

■

Chapter 4

Decidability Results

In this chapter we investigate whether the verification problem for various system classes as defined in Chapter 3 is decidable. We present decidable as well undecidable subclasses, trying to explore the boundaries of undecidability. Some of the decidable classes correspond to classes known to be decidable in the literature; in this case we discuss the relationship of our work to the published results, and how already known results can be lifted to our setting.

The verification problem itself is parameterized by both the system class and the class of properties to establish. Therefore, we investigate the verification problem for several variants of basic system classes and different property classes.

We start with the Σ system classes which correspond to systems consisting of user processes that proceed asynchronously, and which are connected synchronously or asynchronously with some controller.

Starting with Section 4.4 we present results for systems consisting of synchronously proceeding user processes and some synchronously or asynchronously parallel controller.

We give an overview over all the results established in this chapter in Section 4.9.

4.1 Verification of Σ_0 Systems

We start this chapter with showing decidability of the asynchronous system classes (with respect to the user processes) Σ_0 with synchronous and asynchronous control. These classes have in common that only existential quantification over other processes is allowed in transition guards. Interestingly, the construction used for the synchronous case can also be used for the asynchronous case, showing decidability for the whole verification problem. Nevertheless, we give another direct construction for the asynchronous case first.

4.1.1 Decidability for Asynchronous Control

In this section we show that the verification problem for Σ_0 systems with asynchronous control is decidable for several slight restrictions of the general problem, namely if we restrict the properties to safety properties only, or if one restricts the system class to non-blocking or weak systems only.

The result is related to that of [EK00], where also systems consisting of a parameterized number of asynchronous processes with existential guards are investigated. We follow the same approach and show that the problem is reducible to the verification of system instances up to a certain bound. For a detailed discussion on the relationship see Technical Note 4.15.

We start by making a few observations about all Σ_0 systems.

Lemma 4.1
Assume that ρ is the transition relation of a Σ_0 system $\mathcal{S}$, with synchronous or asynchronous control, $n \geq 1$, and $(\sigma, \sigma') \models \rho$. Then, the following holds:

- There exists $l < n$ such that $\sigma' = (\sigma : G, l \mapsto \pi_G(\sigma'), \pi_l(\sigma'))$.
- If $\pi_l(\sigma) \neq \pi_l(\sigma')$, then there exists $\tau_u \in T_U$, such that $(\sigma, \sigma') \models \rho_{\tau_u}(l)$.
- If $\pi_G(\sigma) \neq \pi_G(\sigma')$, then there exists $\tau_c \in T_C$, such that $(\sigma, \sigma') \models \rho_{\tau_c}$.

Proof: By definition of the Σ_0 class, especially Definitions 3.25 and 3.26 we make the following observations:

- In case of asynchronous control, only one of the parts for controller and user transitions of ρ must be satisfied. By definition, all variables not involved in such a transition are not altered, consequently, there is at most one of these parts that is satisfied.

- In case of synchronous control, both the user and the controller part of ρ must hold, but it is by definition required that variables not belonging to the active user process are unchanged.

So, at most global variables and array variables in one particular position l are modified. The lemma then follows by definition of local state and state variant. ■

In case we allow only asynchronous control, the result is even sharper: either the control make a step, or one of the processes, as one would expect.

Lemma 4.2
Let $\mathcal{S}(n)$ be an instance of a Σ_0 system with asynchronous control. Then, $(\sigma, \sigma') \models \rho$ implies that either

$$\sigma' = (\sigma : l \mapsto \pi_l(\sigma')) \ ,$$

for some $l < n$, or

$$\sigma' = (\sigma : G \mapsto \pi_G(\sigma')) \ .$$

Proof: Follows directly from the definition of the Σ_0 transitions. ■

For the rest of this section we assume that a parameterized system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ in the class Σ_0 with asynchronous control is given.

The next lemma states that the processes are indeed not closely connected, since the enabledness of a transition only depends on the set of local states induced by a state, but, for example, not

- on the number of processes in one particular local state,
- on the local state of a certain process (so they are "interchangeable"), or
- on the local states of "neighbored" processes.

Such dependencies are beyond the limits of this system class.

Lemma 4.3
For all $n, \tilde{n} \in \mathbb{N}$, $\sigma, \sigma' \in \Sigma_n, \tilde{\sigma} \in \Sigma_{\tilde{n}}, k < n$, and $\tilde{k} < \tilde{n}$ with

- $\text{Localstates}(\sigma, \neq k) \subseteq \text{Localstates}(\tilde{\sigma}, \neq \tilde{k})$,
- $\pi_G(\tilde{\sigma}) = \pi_G(\sigma) = \pi_G(\sigma')$,
- $\pi_k(\sigma) = \pi_{\tilde{k}}(\tilde{\sigma})$, and $\pi_k(\sigma) \neq \pi_k(\sigma')$,

the following holds:

$$(\sigma, \sigma') \models \rho \text{ implies } (\tilde{\sigma}, \tilde{\sigma}') \models \rho \ ,$$

for $\tilde{\sigma}' \stackrel{\text{def}}{=} (\tilde{\sigma} : \tilde{k} \mapsto \pi_k(\sigma'))$.

Proof: By Lemma 4.1 and 4.2, $\sigma' = (\sigma : k \mapsto \pi_k(\sigma'))$, and there exists a transition τ_u such that

$$(\sigma, \sigma') \models \rho_{\tau_u}(k) \ .$$

Our goal is to show that also $(\tilde{\sigma}, \tilde{\sigma}') \models \rho_{\tau_u}(\tilde{k})$. Basically, we have to show the following:

1. The local states of all processes different from $\tilde{k}$ are unchanged, which is trivial by the definition of $\tilde{\sigma}'$.
2. The transition $\rho_{\tau_u}(\tilde{k})$ is enabled.
3. The values given to the global variables and the arrays in position $\tilde{k}$ match with the values assigned in ρ_{τ_u} (we refer here to the "guarded command style" notation for transitions).

The third point is quite obvious, since $(\sigma, \sigma') \models \rho_{\tau_u}(k)$ holds, and the same values are assigned in both steps.

So only the enabledness remains. Since $\mathcal{S}$ is a Σ_0 system, we know that the transition guard has the form

$$\exists_N \bar{\jmath} : \phi(i) \wedge \bar{\jmath} \neq i \wedge \psi(i, \bar{\jmath}) \ .$$

Consequently, there must exist indices $\bar{l} < n$ such that $\sigma \models \phi(k) \wedge k \neq \bar{l} \wedge \psi(k, \bar{l})$, because $(\sigma, \sigma') \models \rho_{\tau_u}(k)$ (the guard does not refer to the post-state, so no σ' is needed here). From $k \neq \bar{l}$ and $\text{Localstates}(\sigma, \neq k) \subseteq \text{Localstates}(\tilde{\sigma}, \neq \tilde{k})$ we know there exist $\bar{m} < \tilde{n}, \bar{m} \neq \tilde{k}$ satisfying $\pi_{l_p}(\sigma) = \pi_{m_p}(\tilde{\sigma})$ for $1 \leq p \leq e$, where $\bar{l} = l_1, \ldots, l_e$ and $\bar{m} = m_1, \ldots, m_e$.

We observe that every sub-term of $\phi(k) \wedge k \neq \bar{l} \wedge \psi(k, \bar{l})$ in state σ is evaluated to exactly the same value as the corresponding sub-term of $\phi(\tilde{k}) \wedge \tilde{k} \neq \bar{m} \wedge \psi(\tilde{k}, \bar{m})$ evaluated in $\tilde{\sigma}$. This holds since the only indices used in these sub-terms are k and the $\bar{l}$ indices resp. $\tilde{k}$ and the $\bar{m}$ indices, and the corresponding local states are identical. This observation can be proved by a simple structural induction proof. Hence, $\sigma \models \phi(k) \wedge k \neq \bar{l} \wedge \psi(k, \bar{l})$ iff $\tilde{\sigma} \models \phi(\tilde{k}) \wedge \tilde{k} \neq \bar{m} \wedge \psi(\tilde{k}, \bar{m})$.

Therefore, we conclude that transition τ_u is enabled for process $\tilde{k}$ in state $\tilde{\sigma}$.

By definition of ρ we can conclude that $(\tilde{\sigma}, \tilde{\sigma}') \models \rho$. ■

We note that it may be the case that $(\sigma, \sigma') \models \rho$ but $\sigma = \sigma'$. In this case the lemma cannot be applied, since the prerequisites are not met.

Lemma 4.4
For all $n, \tilde{n} \in \mathbb{N}$, $\sigma, \sigma' \in \Sigma_n$, and $\tilde{\sigma} \in \Sigma_{\tilde{n}}$ with

- Localstates$(\sigma) \subseteq$ Localstates$(\tilde{\sigma})$,
- $\pi_G(\sigma) = \pi_G(\tilde{\sigma})$, and
- $\pi_G(\sigma) \neq \pi_G(\sigma')$,

the following holds:

$$(\sigma, \sigma') \models \rho \text{ implies } (\tilde{\sigma}, \tilde{\sigma}') \models \rho \ ,$$

for $\tilde{\sigma}' \stackrel{\text{def}}{=} (\tilde{\sigma} : G \mapsto \pi_G(\sigma'))$.

Proof: Similar to the proof of Lemma 4.3. ■

The next lemma is the key result for decidability. It can be used to show that for every safety property counterexample that exists for a large system instance, one can also find one for a system up to a certain bound. Therefore, for these properties, verification of finitely many system instances up to a certain bound already allows to conclude the correctness for the whole parameterized system.

Lemma 4.5 (Bounding lemma)
For all $n > |\Sigma_L| + k, n_1, \ldots n_k < n$, and LTL formulae $\phi(i_1, \ldots, i_k)$,

$$\alpha \in [\![\mathcal{S}(n)]\!] \text{ and } \alpha \models \phi(n_1, \ldots, n_k)$$

implies that there exists a *prefix* β of a trace in $[\![\mathcal{S}(|\Sigma_L| + k)]\!]$ and there exist $n'_1, \ldots n'_k < |\Sigma_L| + k$ such that

$$\beta \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: Let $n > |\Sigma_L| + k$, and assume that $\alpha \in [\![\mathcal{S}(n)]\!]$ such that $\alpha \models \phi(n_1, \ldots, n_k)$.

Without loss of generality, we assume that $n_i \neq n_j$, for $i \neq j$. Since we only look in our construction at individual local traces, only the set of these process identities is important. If $n_i = n_j$ for some $i \neq j$, then we do

the same construction only for this set of identities, and define later the n_i' corresponding to the n_i with some identical values.

Let $L \stackrel{\text{def}}{=} \text{Localstates}(\alpha)$ be the set of local states occurring in α.

Since L is finite, those states must appear on a finite prefix of α. Hence, choose some $r < |\alpha|$ such that all states in L appear in the prefix of α up to position r, i.e., for each $s \in L$ we find a position $j \leq r$ such that $\alpha(j)$ induces s as described in Definition 3.34.

Since $\mathcal{S}$ has asynchronous control, in each step of the trace, either one local component (of one index), or the global component of the state changes, but never both simultaneously, see Lemma 4.2.

So let $0 = c_0 < \ldots < c_g \leq r$ be the positions where the global state component of α changes; more precisely, the first occurrences of a global state in α, such that for all $0 \leq j < g$ we have:

- $\text{Globalstates}(\alpha(i)) = \text{Globalstates}(\alpha(c_j))$ for all $c_j \leq i < c_{j+1}$,
- $\text{Globalstates}(\alpha(c_j)) \neq \text{Globalstates}(\alpha(c_{j+1}))$, and
- $\text{Globalstates}(\alpha(i)) = \text{Globalstates}(\alpha(c_g))$ for all $c_g \leq i \leq r$.

Now sort the set L according to the first position in which each local state appears in α, beginning with the initial local states. So,

$$L = \{x_0, \ldots, x_l\}$$

such that x_m appears before (or, in case both are initial local states, simultaneously with) $x_{m'}$ in α, for all $m < m'$.

Then, for each $j \leq g$, there exists an l_j such that

$$M_j \stackrel{\text{def}}{=} \{x_0, \ldots, x_{l_j}\} = \text{Localstates}(\alpha|_{c_j}),$$

i.e., all local states appearing in α up to the j'th controller state are exactly $\{x_0, \ldots, x_{l_j}\}$. In the step $\alpha(c_j{-}1) \to \alpha(c_j)$, the global component is changed, hence, the set of local states remains the same. Note that M_0 contains all initial local states. M_1 is a set of all local states which can be reached with the initial global state, thus, without any controller activity. Note that $l_g < l$ is possible, such that $M_g \subsetneq L$. In this case, after the global state changed in position c_g, some new local states are reached in α.

For each local state $s \in L$, there exists a first position j_s such that $s \in \text{Localstates}(\alpha(j_s))$, i.e., one can find a process index i_s with $\pi_{i_s}(\alpha(j_s)) = s$. Let

$$\text{MinComp}(s) \stackrel{\text{def}}{=} \pi_{i_s}(\alpha|_{j_s+1})$$

be the local trace of process i_s ending in the first occurrence of s.

The idea is that we now construct a trace such that all local states from L are visited in the order given above. Once a (new) local state is reached, that process will stop moving and remains in that local state. Since all transitions are $\exists$-transitions, this can never block other processes or the controller. In the contrary, the more local states are reached the more transitions are enabled. Once all states from L are reached, the ϕ-processes together with the controller can do every step taken in α, without any activity of other processes. Enabledness of a transition of process i only depends on its local state and the current global state.

Note that it is not possible to order the local states by the number of *local* steps of that process needed to reach it. E.g., it could be the case that only one transition is used to reach a local state, but this transition could rely on a number of other local states reached before by other processes.

Note further that for an initial state, there may be many different processes reaching that state (with the same number of local steps, namely no steps). But for non-initial local states, there is exactly one process reaching that state first, since at every step, only one local state or the global state is modified.

Table 4.1 illustrates the construction of the run of $\mathcal{S}(|\Sigma_L| + k)$.

Note that there can be more than one initial local state like x_0 which is reached without a local step. Each arrow represents a sequence of transitions of that user process (or of that set of processes). The last arrows represent an interleaving of the control process and the local traces of processes n_1 to n_k. Here, x_{n_j} is the local computation of process n_j in α, and $x^m_{n_j}$ is the m'th local state in that local computation.

The constructed trace has the following properties:

- It starts with the local traces reaching all the local states M_1 reached in α during the initial global state. Afterwards, these processes stop, so all local M_1 states will be *available* for the rest of the run.

- The next processes, which shall reach the local states $L \setminus M_1$, proceed, reaching an intermediate state of their local trace, namely the last state reached in α with the actual global state.

- Then, the processes used for the property ϕ (called ϕ-processes) make parts of their local traces which are done in α during the initial global state. Since the resulting trace shall satisfy ϕ, it is essential that their steps are not appended one after the other, but simultaneously (this works since the local steps are already interleaved in α). So we project the subsequence of α on the ϕ-processes and append them together.

Table 4.1: Constructed run of $\mathcal{S}(|\Sigma_L| + k)$

y_0	$\underbrace{x_0}$	x_0'	$\ldots$	x_0''	$\ldots$	$x_{n_1}^0$	$\ldots$	$x_{n_k}^0$
	M_0	x_1	$\ldots$	x_0''	$\ldots$	$x_{n_1}^0$	$\ldots$	$x_{n_k}^0$
			$\ldots$	x_0''	$\ldots$	$x_{n_1}^0$	$\ldots$	$x_{n_k}^0$
y_0	x_0	x_1	$\ldots$	x_i	$\ldots$	$x_{n_1}^0$	$\ldots$	$x_{n_k}^0$
y_0		M_1			$\ldots$	$x_{n_1}^1$	$\ldots$	$x_{n_k}^1$
y_0					$\ldots$	$x_{n_1}^1$	$\ldots$	$x_{n_k}^1$
y_1					$\ldots$	$x_{n_1}^1$	$\ldots$	$x_{n_k}^1$
y_1					$,\ldots$	$x_{n_1}^1$	$\ldots$	$x_{n_k}^1$
y_1		M_2			$\ldots$	$x_{n_1}^2$	$\ldots$	$x_{n_k}^2$
y_1					$\ldots$	$x_{n_1}^2$	$\ldots$	$x_{n_k}^2$
y_2					$\ldots$	$x_{n_1}^2$	$\ldots$	$x_{n_k}^2$
y_g					M_g	$x_{n_1}^g$	$\ldots$	$x_{n_k}^g$
y_g					L	$x_{n_1}^g$	$\ldots$	$x_{n_k}^g$

- Now the essential part of α up to the first global step has been done, so next the control process makes a step.

- This construction is now repeated until all other sets M_j, for $j \leq g$, and global states y_j are reached. Then, the rest of the local states of L can be added. The ϕ-processes are then in an intermediate state of their local computation.

- Now, all local states of α are available *simultaneously*, which means that for all transitions taken in α, the enabledness is only depending on the global states and the process' own local state, but not on the local states of other processes. Then, only the ϕ-processes and the controller are taking transitions to fulfill the property ϕ.

The computation will start with the state σ_0 such that

- $\pi_G(\sigma_0) = \pi_G(\alpha(0))$,

- $\pi_j(\sigma_0) = \mathrm{MinComp}(x_j)(0)$ for $0 \leq j \leq l$ (the initial state of the local trace leading to local state x_j),

- $\pi_j(\sigma_0) = \mathrm{MinComp}(x_0)(0)$ for $l < j < |\Sigma_L|$ (initial states for processes not used in the construction in case $|L| < |\Sigma_L|$),

- $\pi_{m+j}(\sigma_0) = \pi_{n_j}(\alpha(0))$ for $1 \leq j \leq k$, where $m \stackrel{\text{def}}{=} |\Sigma_L| - 1$ (these are the ϕ-processes).

Starting with σ_0, we now construct the run. We say that a sequence β_j is constructed up to index j, if the last step changed the global component to the global state of $\alpha(c_j)$, and all local states x_m are reached for $m \leq l_j$ (and the processes with index $m > l_j$ have taken corresponding steps). For c_0, this only means that we have the trace consisting of the initial state σ_0.

So assume that β_j is constructed up to index $j < g$.

- First add the local steps leading to the new local states $M_{j+1} \setminus M_j = \{x_{l_j+1}, \ldots, x_{l_{j+1}}\}$ by

$$\mathrm{add_L}(\bar{\beta}, m, \mathrm{MinComp}(x_m)[c_j + 1 \ldots c_{j+1}])$$

for all $l_j < m \leq l_{j+1}$ (add them successively in increasing order). Here, $\bar{\beta} = \beta_j$ for the first index, and the outcome of the previous index for the rest. Let β' be the result.

- The processes which have not reached "their" local state to stop in, follow a part of their local trace:

  $$\mathrm{add}_{\mathrm{L}}(\bar{\beta}, m, \mathrm{MinComp}(x_m)[c_j + 1 \ldots c_{j+1}])$$

  for all $l_{j+1} < m \leq l$. Similarly to the previous case, $\bar{\beta}$ starts with β', and for an index $m+1$, $\bar{\beta}$ is the result of the computation for index m. β'' is the result of this construction.

- Next, the local traces of the ϕ-processes can be added:

  $$\beta''' \stackrel{\mathrm{def}}{=} \mathrm{add}_{\mathrm{L}}(\beta'', (m+1, \ldots, m+k), \\ (\pi_{n_1}, \ldots, \pi_{n_k})(\alpha[c_j + 1 \ldots c_{j+1}]))$$

  where $m \stackrel{\mathrm{def}}{=} |\Sigma_L| - 1$.

- Last, the global state can change:

  $$\beta_{j+1} \stackrel{\mathrm{def}}{=} \mathrm{add}_{\mathrm{G}}(\beta''', \pi_G(\alpha(c_{j+1}))) \ .$$

  Define $d_{j+1} \stackrel{\mathrm{def}}{=} |\beta_{j+1}| - 1$. This is the position in the constructed trace corresponding to position c_{j+1} in α (the "first" occurrence of the new global state). We start with $d_0 \stackrel{\mathrm{def}}{=} 0$.

Then, β_{j+1} is constructed up to index $j+1$.

Now, let β_g be the trace constructed up to index g. Then, there may be some more local states not yet reached:

- For each $l_g < m \leq l$ append in increasing order the remaining local steps leading to the local states $L \setminus M_g$:

  $$\mathrm{add}_{\mathrm{L}}(\bar{\beta}, m, \mathrm{MinComp}(x_m)|^{c_g+1}) \ .$$

  Again, $\bar{\beta} = \beta_g$ for the first index $m = l_g + 1$, and $\bar{\beta}$ is the previous result for $m > l_g + 1$.

Let $\tilde{\beta}$ be the resulting trace, with final state σ. Let β be the trace starting like $\tilde{\beta}$, i.e.,

$$\beta|_h = \tilde{\beta} \text{ for } h \stackrel{\mathrm{def}}{=} |\tilde{\beta}| \ ,$$

such that for all $j \geq 0$,

- $\pi_m(\beta(h+j)) = \pi_m(\sigma)$, for each $0 \leq m < |\Sigma_L|$,

- $\pi_G(\beta(h+j)) = \pi_G(\alpha(c_g+1+j))$, and
- $\pi_{|\Sigma_L|-1+m}(\beta(h+j)) = \pi_{n_m}(\alpha(c_g+1+j))$, for $1 \leq m \leq k$.

Then, β is a stuttering of a trace of $\mathcal{S}(|\Sigma_L|+k)$ which fulfills $\phi(n'_1,\ldots,n'_k)$ with $n'_j \stackrel{\text{def}}{=} |\Sigma_L|-1+j$.

We have to show that

1. removing the stuttering from β results in a (prefix of a) trace of $\mathcal{S}(|\Sigma_L|+k)$, and
2. $\beta \models \phi(n'_1,\ldots,n'_k)$.

First we proof some helpful properties about the construction. ■

Lemma 4.6
For the sequence β constructed in the previous proof, the following properties hold:

$$\text{For all } j \geq 0 : \text{Localstates}(\beta(j)) \subseteq \text{Localstates}(\beta(j+1)) \tag{4.1}$$
$$\text{For all } j \leq g : \text{Localstates}(\beta|_{d_j}) \subseteq M_j \tag{4.2}$$
$$\text{Localstates}(\tilde{\beta}) = L \tag{4.3}$$

Proof: We prove property (4.1) by induction on j.

Base case: Localstates($\beta(0)$) is the set of all initial local states occurring in α. For each such local state s, there is one process $h \leq l_0$ to which no more local steps are added, so that this process remains by construction in state s in the whole sequence β. Consequently, Localstates($\beta(0)$) $\subseteq$ Localstates($\beta(1)$).

Inductive step: The only interesting case is that one of the local states is modified in that step, so assume that $\pi_h(\beta(j)) \neq \pi_h(\beta(j+1))$. The question is whether $s \stackrel{\text{def}}{=} \pi_h(\beta(j)) \in \text{Localstates}(\beta(j+1))$. We observe that s is a local state appearing in α, consequently, $s \in L$.

- If $h < |\Sigma_L|$, then s is an intermediate state appearing "on the way" to local state x_h. Consequently, MinComp(s) is shorter than MinComp(x_h), so there is a process $h' < h$ with $x_{h'} = s$. Since local states are added in the order of their minimal computation length, the steps of h' are already added. Consequently, process h' has already reached s (since h reached s with that sequence of global states, h' is also able to do that, maybe even earlier) and stopped in that state.

- If $h \geq |\Sigma_L|$, then s is reachable with the sequence of global states in $\beta|_j$. Since ϕ-steps are added last in the construction of β, the whole set of reachable local states with that global step sequence is already reached before. Hence, $s \in \text{Localstates}(\beta(j+1))$.

The next property is (4.2). Previous to the addition of a global step from $d_j - 1$ to d_j, local steps are added leading to all local states reachable with the sequence of global states seen so far. So, $M_j \subseteq \text{Localstates}(\beta|_{d_j})$.

On the other hand, if there is a local state $s \in \text{Localstates}(\beta|_{d_j})$, there obviously exists a local computation in α leading to s with that global state sequence, since the local trace added in each index is a local trace in α. So by construction, there is one process reaching s first and stopping then.

Property (4.3) follows from the previous property and with the observation that after M_g is reached in the prefix of $\tilde{\beta}$, first the missing local states $L \setminus M_g$ are appended. ■

Proof: Now we finish the proof of Lemma 4.5. As already said, we have to show that

1. removing the stuttering from β results in a (prefix of a) trace of $\mathcal{S}(|\Sigma_L| + k)$, and

2. this trace fulfills ϕ.

We start with 1. From the construction it is obvious that

$$\text{Localstates}(\alpha(0)) = \text{Localstates}(\sigma_0) = \text{Localstates}(\beta(0)) \ .$$

Moreover, $\pi_G(\alpha(0)) = \pi_G(\sigma_0) = \pi_G(\beta(0))$. Consequently, by Corollary 3.42, $\alpha(0) \models \theta$ implies $\beta(0) \models \theta$.

For the second condition let us assume that $0 \leq j < |\beta| - 1$, and that $\beta(j) \neq \beta(j+1)$ (otherwise this "step" is erased when stuttering is removed).

First consider the case $j + 1 \leq |\tilde{\beta}|$. Then, the step was added during the first part of the construction. By construction, $\beta(j+1)$ differs from $\beta(j)$ in either the global state, or in the local state of exactly one process l.

- $\pi_G(\beta(j)) \neq \pi_G(\beta(j+1))$: By construction of β there exists a smallest h with $\pi_G(\alpha(c_h - 1)) = \pi_G(\beta(j))$ and $\pi_G(\alpha(c_h)) = \pi_G(\beta(j+1))$. Before this step (or a subsequent global step with these global states) is added in the construction, the local steps are appended. Hence, $\text{Localstates}(\alpha(c_h - 1)) \subseteq \text{Localstates}(\alpha|_{c_h}) = M_h \subseteq \text{Localstates}(\beta(j))$.

Hence, Lemma 4.4 gives us

$$(\beta(j), (\beta(j) : G \mapsto \pi_G(\alpha(c_h)))) \models \rho \ .$$

This completes the case, because $(\beta(j) : G \mapsto \pi_G(\alpha(c_h))) = (\beta(j) : G \mapsto \pi_G(\beta(j+1))) = \beta(j+1)$.

- $\pi_l(\beta(j)) \neq \pi_l(\beta(j+1))$: Let h be the first occurrence of a local state change $\pi_l(\beta(j)) \rightarrow \pi_l(\beta(j+1))$ in α with global state $\pi_G(\beta(j))$, i.e., $\pi_m(\alpha(h)) = \pi_l(\beta(j))$, $\pi_m(\alpha(h+1)) = \pi_l(\beta(j+1))$, and $\pi_G(\beta(j)) = \pi_G(\alpha(h))$, for some m. There is such a step, because in the construction of β, local steps are added only when such a step appears in α with the same global state.

  To use Lemma 4.3 we first have to establish Localstates$(\alpha(h), \neq m) \subseteq$ Localstates$(\beta(j), \neq l)$. So assume that $s \in$ Localstates$(\alpha(h), \neq m)$.

  - If $l < |\Sigma_L|$, then this step is an intermediate step of the local trace to local state x_l. Since this process does not stop in s, and with the global state sequence seen so far, s is reachable, there must exist $l' < l$ such that $x_{l'} = s$. Since s is reachable with these steps, process l' has already stopped in s (l' steps are added before the steps of process l, and the local trace to $x_{l'}$ is obviously shorter than that to x_l). Consequently, $s \in$ Localstates$(\beta(j), \neq l)$.
  - If $l \geq |\Sigma_L|$, then this is a step of a ϕ-process. Take the corresponding step in α. The construction ensures that prior to a ϕ-process step, every local state appearing in α up to the next global step is already reached (this is one of the $M_{j'}$, for a particular j'), namely by a process with index $l' < |\Sigma_L|$. Since s obviously belongs to this set, $s \in$ Localstates$(\beta(j), \neq l)$. Since all $n_m \neq n_{m'}$ for $m \neq m'$, in each step only one local state is modified.

  So consequently Localstates$(\alpha(h), \neq m) \subseteq$ Localstates$(\beta(j), \neq l)$ holds. Since α is a trace of $\mathcal{S}(n)$, $(\alpha(h), \alpha(h+1)) \models \rho$ holds. Lemma 4.3 then states, that

  $$(\beta(j), (\beta(j) : l \mapsto \pi_m(\alpha(h+1)))) \models \rho \ .$$

  Observing $(\beta(j) : l \mapsto \pi_m(\alpha(h+1))) = (\beta(j) : l \mapsto \pi_l(\beta(h+1))) = \beta(j+1)$ completes this case.

Now we have to consider the case $j+1 > |\tilde{\beta}|$. Then, Localstates$(\beta(j), I) = L$, for $I = \{0, \ldots, |\Sigma_L| - 1\}$, and this is either a controller step or a local step of a ϕ-process $l \geq |\Sigma_L|$.

Since the global state sequence and the local traces of the ϕ-processes are simply projections of corresponding α traces, the global state always "matches" to the local states.

So, for the corresponding position h in α, where the controller (resp. a process m) makes this step, obviously the conditions for Lemma 4.4 (resp. Lemma 4.3) are fulfilled, so again $(\beta(j), \beta(j+1)) \models \rho$ holds.

This implies that β is at least a prefix of a $\mathcal{S}(|\Sigma_L| + k)$ trace. Note that if α is infinite, then also β is, but the stuttering removal could result in a finite trace.

On the other hand, if α is finite, then it is a *deadlocked* trace. In the given construction, there are usually "more" local states available in the final state of β than in the final state of α, so it is not clear that also β is deadlocked! In the contrary, in many cases β can be prolonged.

But if we restrict to safety properties only, which we do in this case, a finite prefix is enough to invalidate the property. This is used in the following theorem stating decidability of the verification problem.

The second thing to show is that the (prefix) trace resulting from removing stuttering steps indeed fulfills our property ϕ. Recall that $n'_j \stackrel{\text{def}}{=} |\Sigma_L| - 1 + j$, for $1 \leq j \leq k$, which are the indices of the ϕ-processes. As mentioned in the beginning, if there are some $n_i = n_j$ for $i \neq j$, then we use each process in the given construction only once, and define the n'_i correspondingly.

Lemma 3.44 and Lemma 3.10 then imply that the trace resulting from removing all stuttering steps fulfills ϕ: $\natural\beta \models \phi(n'_1, \ldots, n'_k)$ (in fact slight variations of these lemmas are needed, but the general idea is the same). ■

Note that the existence of a *prefix* fulfilling ϕ does not imply in general the existence of a full trace that fulfills ϕ.

The following lemma, combined with Lemma 4.5, can be used to reduce the verification of safety properties to the verification of exactly one instance, as it shows that safety property counterexamples of small system instances are also present in every larger instance.

Lemma 4.7
For all $n \geq 1, n_1, \ldots n_k < n$, and LTL formulae $\phi(i_1, \ldots, i_k)$,

$$\alpha \in [\![\mathcal{S}(n)]\!] \text{ and } \alpha \models \phi(n_1, \ldots, n_k)$$

implies that there exists a *prefix* β of a trace in $[\![\mathcal{S}(n+1)]\!]$ such such that

$$\beta \models \phi(n_1, \ldots, n_k) \ .$$

The property described in Lemma 4.7 can be seen as *monotonicity* of the parameterized system: if one system has some behavior, all larger instances share it.

Proof: Given α, one can choose an initial local state of one process, say process 0, that is, $\sigma_L \stackrel{\text{def}}{=} \pi_0(\alpha(0))$. Then, one can simply "extend" each state in α by this local state, which corresponds to a trace where the new process $n+1$ stutters in its initial state. So choose β such that for all $j \geq 0$,

- $\pi_G(\beta(j)) = \pi_G(\alpha(j))$,
- $\pi_l(\beta(j)) = \pi_l(\alpha(j))$ for each $l < n$, and
- $\pi_n(\beta(j)) = \sigma_L$.

Corollary 3.42 states that $\beta(0) \models \theta$. It is obvious that the new process cannot block any transition taken in α, so this is a valid prefix of a trace. As before, one cannot be sure that this trace is maximal. The result follows by Lemma 3.44. ■

Note that it is not possible to extend this result to full traces in $[\![\mathcal{S}(n+1)]\!]$, as the next example illustrates.

Example 4.8
There exists a system $\mathcal{S}$, $n \geq 1$, an LTL property $\phi(i_1)$ and a trace $\alpha \in \mathcal{S}(n)$, such that $\alpha \models \phi(0)$, but for all traces $\beta \in [\![\mathcal{S}(n+1)]\!]$, $\beta \not\models \phi(0)$.

Take the simple system $\mathcal{S} = (N, \{s\}, \rho, \theta)$, where s is an array over $\{1, 2\}$, $\theta \stackrel{\text{def}}{=} \forall i.s[i] = 1$, and ρ is built up by the transition

$$\rho_\tau(i) \stackrel{\text{def}}{=} \exists_N j.s[i] = 1 \wedge j \neq i \wedge (s[j] = 1 \vee s[j] = 2) \rightarrow s[i] := 2 \ .$$

Each process of an instance of this system can graphically be represented by Figure 4.1, where the number in each node gives the value of $s[i]$.

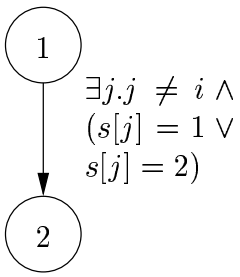

Figure 4.1: Non-monotone system

Choose the LTL property $\phi(i) \stackrel{\text{def}}{=} \Box s[i] = 1$.

For $n = 1$, there is a deadlock trace, namely the only trace of $\mathcal{S}(1)$, which consists of the initial state. Consequently, $\phi(0)$, i.e., $\Box s[0] = 1$, is a valid property of the system instance $\mathcal{S}(1)$. For $n = 2$, there are two possible traces, and in both of them, both processes eventually reach the local state $s[i] = 2$. So, for every trace $\beta \in [\![\mathcal{S}(2)]\!]$, $\beta \not\models \phi(0)$ holds.

Note that exactly as Lemma 4.7 states, there exist a prefix for every trace that fulfills $\phi(0)$, namely the prefix consisting only of the initial state. ♣

Using Lemma 4.5 and Lemma 4.7, we can state the decidability result for safety properties.

Theorem 4.9
The verification problem for Σ_0 systems with asynchronous control is decidable for safety properties.

Proof: Assume that $\mathcal{S}$ is a parameterized system with asynchronous control. Define $b \stackrel{\text{def}}{=} |\Sigma_L| + k$. Assume we have to show

$$\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k) \ .$$

Then, prove whether

$$\mathcal{S}(b) \models \phi(0, \ldots, k-1) \ ,$$

which is decidable and by Lemma 3.47 equivalent to

$$\mathcal{S}(b) \models \forall_d \phi(i_1, \ldots, i_k) \ .$$

If $\mathcal{S}(b)$ does not fulfill this property, give "false" as answer. Otherwise, return the answer "valid".

In case $n_0 > b$ and there is a counterexample for ϕ and system instance $\mathcal{S}(b)$, Lemma 4.7 states that there exists a prefix β of a $\mathcal{S}(n_0)$ trace that is also not fulfilling ϕ. If β is infinite, then this is a counterexample for $\mathcal{S}(n_0)$, hence, the answer "false" is correct.

Otherwise, if β is finite, we use the fact that ϕ is a safety property. Therefore, we can derive from Corollary 3.17 that no matter how this sequence is extended to a full trace, the extension will also be a counterexample for ϕ. So in any case, a negative answer is correct.

If $n_0 \leq b$, then a negative answer is obviously correct.

To prove that all positive answers are also correct, assume the answer is "valid" and there exists an $n \neq b$ such that $\mathcal{S}(n) \not\models \forall_d \phi(i_1, \ldots, i_k)$. Then

there exists $\alpha \in [\![\mathcal{S}(n)]\!]$ and pairwisely distinct $n_1, \ldots, n_k < n$ such that $\alpha \not\models \phi(n_1, \ldots, n_k)$, hence, $\alpha \models \neg\phi(n_1, \ldots, n_k)$

If $n > b$, then the bounding lemma says there exist (pairwisely distinct) $\bar{n}_1, \ldots, \bar{n}_k < b$ and a prefix β of a trace in $[\![\mathcal{S}(b)]\!]$ such that $\beta \models \neg\phi(\bar{n}_1, \ldots, \bar{n}_k)$.

Otherwise, if $n < b$, the existence of a corresponding trace prefix can be derived from Lemma 4.7.

If β is a maximal trace, we can conclude $\mathcal{S}(b) \not\models \forall_d \phi(i_1, \ldots, i_k)$. Contradiction!

Otherwise, if β is only a finite prefix of a trace we again make use of the fact that ϕ is a safety property, and derive from Corollary 3.17 that there exists a trace of $\mathcal{S}(b)$ that is a counterexample for $\phi(\bar{n}_1, \ldots, \bar{n}_k)$. Consequently, $\mathcal{S}(b) \not\models \forall_d \phi(i_1, \ldots, i_k)$. Contradiction!

In this proof, the value n_0 is not really important, as each possible counterexample for any size of the system can be found in system $\mathcal{S}(b)$, and each counterexample of $\mathcal{S}(b)$ can be lifted to a counterexample of $\mathcal{S}(n)$. Consequently, $\mathcal{S}(n)$ fulfills the property if and only if $\mathcal{S}(b)$ does.

If $n_0 < k$, then Remark 3.23 states that system instances $\mathcal{S}(n)$ with $n < k$ trivially fulfill the verification property. ■

We note that if each system instance $\mathcal{S}(n)$ is guaranteed never to block, then the result of Theorem 4.9 holds in fact for all verification properties.

Corollary 4.10
The verification problem for *non-blocking* Σ_0 systems with asynchronous control is decidable.

Proof: For non-blocking systems, every trace α of every instance is guaranteed to be infinite. Then, the construction of Lemma 4.7 gives always infinitely traces Consequently, every counterexample of small system instances can be lifted to larger systems.

Moreover, the sequence constructed in Lemma 4.5 is always infinite. If α is a counterexample for any LTL\X property, also this constructed trace is a counterexample for it.

There is only one minor problem: if the controller and the ϕ-processes eventually stop working, such that in α, from a particular position j onwards, only the other processes are active, then by construction, β has only stutter steps from the corresponding position in β onwards.

In this case, it could be the case that the stutter steps does not correspond to steps of the system, such that we have in fact only a finite trace of the system which could be extended by steps of the first processes.

To get a valid infinite trace, one can add one more process which makes infinitely many steps in α to the construction. Such a process can be added in a similar manner as the other processes, making a part of its trace for each global state change in the beginning, and adding the remaining steps together with the steps of the ϕ processes. Hence, a corresponding counterexample can be found at least in system $\mathcal{S}(|\Sigma_L|+k+1)$. Consequently, in this case it would suffice to verify all system instances up to this instance, so the bound is increased by one with respect to the safety case.

If there is no such process, then there must be an "idle" transition which is taken infinitely often, but which does not change the state. In this case, one has to choose one the first $|\Sigma_L|$ processes which ends up in a state, where a process in α makes infinitely many "self-loops", which do not change the local state. This process must be copied, so we introduce a new process that also reaches the state, similar as the original one, and which stops then. Note that the addition is necessary, because the self-loop could rely on the fact that there is another process in the same local state! Adding this process allows to infinitely stutter from a certain point onwards, which leads to an infinite computation.

Let $b \stackrel{\text{def}}{=} |\Sigma_L| + k + 1$. Similarly as in the proof of Theorem 4.9, each (infinite) counterexample found in a system smaller than $\mathcal{S}(b)$ can also be found in $\mathcal{S}(b)$, and also each counterexample of a larger system instance can be reduced to a trace of $\mathcal{S}(b)$. Therefore, it suffices to verify $\mathcal{S}(b) \models \phi(0, \ldots, k-1)$.

On the other hand, if $n_0 > b$ and $\mathcal{S}(b)$ has a counterexample, this trace can be lifted to a counterexample of $\mathcal{S}(n_0)$. ■

It is possible to extend the given bounding lemma construction in such a way that for deadlocked computations, also deadlocked computations are constructed. This result is only given for *weak* Σ_0 systems with asynchronous control, see Definition 3.27.

Lemma 4.11 (Deadlock bounding lemma)
Let $\mathcal{S}$ be a *weak* Σ_0 system with asynchronous control. For all $n > |\Sigma_L| + k, n_1, \ldots n_k < n$, and LTL properties $\phi(i_1, \ldots, i_k)$:

$$\alpha \in [\![\mathcal{S}(n)]\!], \alpha \models \phi(n_1, \ldots, n_k) \text{ and } \alpha \text{ is finite}$$

implies that there exists a trace $\beta \in [\![\mathcal{S}(|\Sigma_L|+k)]\!]$ and there exist $n'_1, \ldots n'_k < |\Sigma_L| + k$ such that

$$\beta \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: Let $n > |\Sigma_L| + k$, and assume that $\alpha \in [\![\mathcal{S}(n)]\!]$, $|\alpha| < \mathbb{N}$, and $\alpha \models \phi(n_1, \ldots, n_k)$.

Without loss of generality, we assume that $n_i \neq n_j$, for $i \neq j$.

We reuse some of the definitions given in the proof of Lemma 4.5, where we *define* g as the position of the last control step in α (this is possible since α is finite), and $L \stackrel{\text{def}}{=} \text{Localstates}(\alpha)$:

- The ordered set of local states $L = \{x_0, \ldots, x_l\}$ with indices l_j.
- The local trace to s, MinComp(s), for each local state s.
- The positions c_j, for $0 \leq j \leq g$, of the control steps.

The idea is similar to that illustrated in Table 4.1, but we now have to activate the first processes at some point such that they eventually leave their local state, making a deadlock possible.

Define $L_E \stackrel{\text{def}}{=} L \setminus \text{Localstates}(\alpha(|\alpha| - 1))$. These are the local states which must be left in our construction, because each of them could invalidate the maximality condition.

For each local state $s \in L_E$ there exists a maximal position h_s and a process index f_s, such that $s \in \text{Localstates}(\alpha(h_s))$ and $\pi_{f_s}(\alpha(h_s)) = s$. Then, we define

$$\text{ExitComp}(s) \stackrel{\text{def}}{=} \pi_{f_s}(\alpha|^{h_s+1}) \ ,$$

which is a local trace leaving local state s and reaching some local state in $L \setminus L_E$, which is a "good" one, since that local state is already available in the final state of α.

The set L_E can be ordered by the position of their last appearance in α, so we can find indices $e_1, \ldots, e_{|L_E|}$ such that

$$L_E = \{x_{e_1}, \ldots x_{e_{|L_E|}}\} \ ,$$

and for $l < l'$, the last position in which x_{e_l} appears is smaller than the position in which $x_{e'_l}$ appears. In the following we abbreviate $h_{x_{e_l}}$ with h_l, and $f_{x_{e_l}}$ with f_l.

The trace β is constructed similar as before, but for L_E processes, we have to add also the exit computation. We start with the same initial state. Now assume that β_j is constructed up to index $j < g$.

- First add the local steps leading to the new local states $M_{j+1} \setminus M_j = \{x_{l_j+1}, \ldots, x_{l_{j+1}}\}$ by

$$\text{add}_{\text{L}}(\bar{\beta}, m, \text{MinComp}(x_m)[c_j + 1 \ldots c_{j+1}])$$

for all $l_j < m \leq l_{j+1}$ (add them successively in increasing order). Here, $\bar{\beta} = \beta_j$ for the first index, and the outcome of the previous index for the rest. Let β' be the result.

- The processes which are not able to reach their "final" state proceed:

$$\mathrm{add}_{\mathrm{L}}(\bar{\beta}, m, \mathrm{MinComp}(x_m)[c_j + 1 \ldots c_{j+1}])$$

for all $l_{j+1} < m \leq l$. $\bar{\beta}$ starts with β', and for an index $m+1$, it is the result of the computation for index m. Let β'' be the result of this construction.

- Next, the local traces of the ϕ-processes can be added:

$$\beta''' \stackrel{\mathrm{def}}{=} \mathrm{add}_{\mathrm{L}}(\beta'', (m+1, \ldots, m+k), \\ (\pi_{n_1}, \ldots, \pi_{n_k})(\alpha[c_j + 1 \ldots c_{j+1}]))$$

where $m \stackrel{\mathrm{def}}{=} |\Sigma_L| - 1$.

- Now, processes from L_E may leave their local states. We have already reached all local states reached in α up to position c_{j+1}. A local state should be left, if the exit computation starts before that position. For each $1 \leq m \leq |L_E|$, if $h_m \leq c_{j+1}$, add (in increasing order)

$$\mathrm{add}_{\mathrm{L}}(\bar{\beta}, e_m, \mathrm{ExitComp}(x_{e_m})[\max(0, c_j - h_m) \ldots c_{j+1} - h_m - 1])$$

where $\bar{\beta}$ is β''' for the first one, and the outcome of the previous m for the following. Note that position l in $\mathrm{ExitComp}(x_{e_m})$ corresponds to position $h_m + 1 + l$ in α. Let β'''' be the result of this construction.

- Exactly the same has to be done for the "unused" processes m with $l < m < |\Sigma_L|$. This is just redoing the addition steps for the corresponding process 0 which stops in the initial state x_0, in case $x_0 \in L_E$.

- Last, the global state can change:

$$\beta_{j+1} \stackrel{\mathrm{def}}{=} \mathrm{add}_{\mathrm{G}}(\beta'''', \pi_G(\alpha(c_{j+1}))) \ .$$

Define $d_{j+1} \stackrel{\mathrm{def}}{=} |\beta_{j+1}| - 1$. This is the position in the constructed trace corresponding to position c_{j+1} in α (the "first" occurrence of the new global state). We start with $d_0 \stackrel{\mathrm{def}}{=} 0$.

Then, β_{j+1} is constructed up to index $j+1$.

At the end, all non-global steps are added in the same way as before, leaving out only the global state change. Afterwards, all stuttering steps can be removed, resulting in a sequence β.

We explain why this a correct computation. Since now the set of local states shrinks from a certain position onward, it is a priori not obvious that each step taken in β is a valid computation step, i.e., the existence of a a corresponding transition that can be taken is not clear.

For each step added to β in the construction, there exist a corresponding step in α. By construction, whenever this step is added, all local states reached before in α are already available in β. A local state s is left prior to this addition only, if it does not appear for the rest of α, for some position j *before* the α-step which is added. This can be understand if one looks on the global state changes: whenever such a step is added, all local states seen in α up to this step are reached, all intermediate computations for ϕ-processes are added, and last, all local states never seen again in α after this global step, are left. So later we only add α steps appearing after this global step.

Consequently, the enabledness of a transition for this step cannot depend on s, and the step is valid. Note that for the weak system class, a lemma similar to Lemma 4.3 holds, where it is only required that the sets of local states of σ and $\tilde{\sigma}$ agree.

What remains is to prove that this trace is indeed maximal. But obviously, for each process in β, the final state of its local computation is also a final state for some process in α. This is obvious for processes which leave "their" local state, and also clear for all the others, since that local states are part of the set of local states of the final α state. So we can conclude that $\text{Localstates}(\text{last}(\alpha)) = \text{Localstates}(\text{last}(\beta))$. This implies that β is maximal, since there is no successor state for the final α state. If there would be any successor for β, a corresponding step could also be done for α. ■

Remark 4.12
For "full" Σ_0 systems, the constructed sequence need not to be maximal. The problem is, that one cannot ensure that all local states seen in the final state of α only once, are also appearing only once in β. If the final state σ_L is reached in α by only one process i, and there are two or more local states that are only visited by this process, then indeed, at least two processes will reach σ_L in β.

But this could enable more transitions, since the transition guard is able to refer to the "other processes only", using the existential quantification over $\bar{j}$ variables and the comparison $\bar{j} \neq i$, so having a local state twice makes a

difference! ♣

Using Lemma 4.11 we can state decidability for all verification properties for the weak system class.

Theorem 4.13
For *weak* Σ_0 systems with asynchronous control, the verification problem is decidable for all verification properties.

Proof: This result is only valid if we make the assumption that $n_0 \leq |\Sigma_L| + k$, which is no severe restriction.

Let $\mathcal{S}$ be a parameterized system with asynchronous control, and let $\phi(i_1, \ldots, i_k)$ be an LTL formula. Define $b \stackrel{\text{def}}{=} |\Sigma_L| + k$. Then, prove for each $n_0 \leq n \leq b+1$ (we assume $n_0 \leq b$), whether

$$\mathcal{S}(n) \models \forall_d \, \phi(i_1, \ldots, i_k) \ ,$$

which can be done by proving

$$\mathcal{S}(n) \models \phi(0, \ldots, k-1) \ ,$$

according to Lemma 3.47.

If the property does not hold in one case, give "false" as answer. Otherwise, return the answer "valid".

If the procedure returns "false", this is obviously correct, since $b \geq n_0$.

To prove that positive answers are correct, assume the answer is "valid" and there exists $n \neq b, n \geq n_0$ such that $\mathcal{S}(n) \not\models \forall_d \, \phi(i_1, \ldots, i_k)$. Hence, there exists $\alpha \in [\![\mathcal{S}(n)]\!]$ and pairwisely distinct $n_1, \ldots, n_k < n$ such that $\alpha \models \neg\phi(n_1, \ldots, n_k)$. Since all systems $\mathcal{S}(n)$ with $n_0 \leq n \leq b+1$ are checked, we conclude $n > b+1$.

If α is infinite, then there exists a corresponding counterexample for system instance $\mathcal{S}(b+1)$, similarly as in the proof of Corollary 4.10. This contradicts our assumption.

Otherwise, if α is deadlocked, Lemma 4.11 shows that there is a corresponding counterexample of $\mathcal{S}(b)$. Contradiction! ■

Note that it does not suffice to check just the instances $\mathcal{S}(b)$ and $\mathcal{S}(b+1)$, since counterexamples of smaller instances cannot be lifted to larger system instances. The result of Lemma 4.7 can only be used for safety properties.

We finish this section with two technical notes. The first one discusses how loosing the restrictions to the initial state conditions influence the decidability construction presented so far. The second note discusses the relationship of our decidability result to already known results.

Technical Note 4.14 (Initial state condition)
In this note we explain why a less restrictive initial state condition would make the construction above impossible. Assume that we are able to express that one particular process starts in a certain local state. This can be done, for instance, by addressing one specific process using a constant as index term

$$\phi(0) \wedge \forall i > 0.\psi(i,j) \ ,$$

or by quantification over index variables using index term comparisons

$$\exists i.\phi(i) \wedge \forall j.j \neq i \Rightarrow \psi(i,j) \ .$$

Figure 4.2 shows a simple system, which does not allow to use the above construction, if we choose $\phi(i) \stackrel{\text{def}}{=} s[i] = 1$ and $\psi(i,j) \stackrel{\text{def}}{=} s[j] = 0$. Only

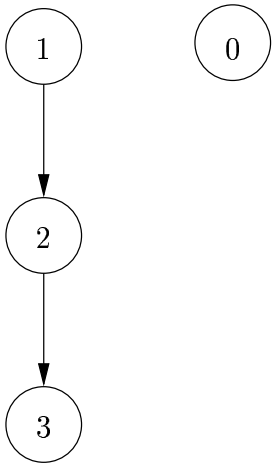

Figure 4.2: Problem with exclusively referenced indices in θ

one process is allowed to start in location 1, all the other have to start in location 0. Hence, it is not possible to reach all local states simultaneously, since location 2 and 3 are both only reachable for the one process starting in 1. Consequently, the construction given in Section 4.1.1 fails in this case.

We think that the proof may also be done if one allows constant indices or existential quantification where one requires all quantified processes to be pairwisely disjunct, but does not allow to *distinguish* them from other processes (by formulae $i > 0$ or $j \neq i$). This would complicate the construction a little bit. One can no longer sort the local traces simply by their length, because the "special" indices referred to by constants or quantified variables are enforced to start with the same local state as in α. A solution can be to

choose local traces for these indices with right local states, and change the order in which local traces are added correspondingly.

In the worst case, all constant index processes (existential quantified process indices) are only able to reach one local state out of Σ_L. Hence, we need more processes for our construction, namely $|\Sigma_L| + k + c - 1$, where c is the number of constant index terms (resp. number of existential quantifications) in θ. ♣

Technical Note 4.15
The disjunctive systems in [EK00] are closely related to our Σ_0 class. The main differences are:

- We also investigate systems with a controller connected to the parameterized number of user processes. In case of the bounding lemma, the rôle of the controller is very similar to the second process class in [EK00]. But for other results, the controller makes a real difference, because it is not possible to choose an instance large enough such that all controller states can be reached simultaneously, as it is the case for a second system class.

- We allow stronger transition guards by using predicates containing many process indices which are existentially quantified.

Our decidability result corrects a flaw present in the proof of the bounding and monotonicity lemma given in [EK00]. This proof does not consider a problem when new traces are constructed of deadlocked computations. We correct these results by proving the results only for safety properties, and giving a general result for a more restricted system classes, namely non-blocking systems and weak Σ_0 systems.

Moreover, our proof differs in that we split the trace by the global transitions and add local traces for different processes independently, ordered by the length of the local traces, while the other processes stutter in their state. Only the processes for the property are handled together. The construction used in [EK00] is based on choosing some trace such that each local trace leading to some local state is the same local trace as in the original sequence. In case there is one process such that its local trace reaches two or more local state first in the run, this local trace is copied and therefore the result is no interleaving, since some processes run synchronously. It is informally argued that the interleaving can be restored.

A further mistake is illustrated by the system given in Example 4.8. It refutes the Single-Cutoff Lemma presented in [EK00] which states that for

LTL properties, it is sufficient to prove only the one system of the cutoff bound.

Using the system of Example 4.8 we show that for general properties, counterexamples of smaller systems are not always found in the semantics of larger instances. Choose the property $\phi(i_1) \stackrel{\text{def}}{=} \Diamond s[i_1] = 2$. So we are interested in proving

$$\mathcal{S}(\geq 1) \models \forall_d \Diamond s[i_1] = 2 \ .$$

Similarly as in the example, one can argue that for every trace of the cutoff system $\mathcal{S}(b)$ (the cutoff is clearly larger than 1), each process eventually reaches the local state $s[i] = 2$, so the property is valid for the cutoff system.

On the other hand, for system instance $\mathcal{S}(1)$, this property is not valid, since the only trace is the one deadlocking in the initial state, thus, never reaching the local state $s[0] = 2$.

Consequently, for general LTL properties it is not possible to reduce the verification only to the system with size of the bound, as it is done for example in Theorem 4.9. ♣

4.1.2 Decidability for Synchronous Control

In this section we prove decidability of the verification problem of Σ_0 systems with synchronous control.

In fact, we can prove this result for an even stronger system class which we call Σ_0^+. For systems of this class we allow that one transition modifies *both* array and normal variables simultaneously. The controller part of the transition relation is then omitted.

Intuitively, such a system can be precisely simulated by a vector addition system with states, these are systems equipped with a set of counters over natural numbers that can be modified by each transition. This idea was also applied in [GS92] for a different class of systems where processes communicate using synchronization labels.

Model-checking of safety properties can be reduced to the problem of checking formulae of the form $\mathsf{E}\Diamond\phi$, for some state predicate ϕ, by using an automaton accepting the set of bad prefixes [KV99]. Following the work in [AČJT96], we show that this reachability problem for vector addition systems can be decided. For general properties, one can use the approach given in [GS92] to show decidability.

First we show the results for properties over controller variables only. The general verification problem can be reduced to the first one. We start by defining the system class of interest.

Definition 4.16 (Σ_0^+ systems)
A Σ_0^+ *system* is a parameterized system with a transition relation ρ built up by transitions of the form

$$\rho_\tau(i) \equiv \exists \bar{j} : \phi(i) \wedge \bar{j} \neq i \wedge \psi(i, \bar{j}) \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y$$

in the usual way. □

We remark that the system given in Example 3.2 belongs to this class. This new class subsumes all Σ_0 systems with synchronous and also asynchronous control.

Remark 4.17
Each Σ_0 system with asynchronous control is also a Σ_0^+ system. Obviously, each Σ_0 user transition is a special case of a Σ_0^+ transition which modifies only array variables. A control transition with guard $g \equiv \exists \bar{j} : \phi \wedge \psi(\bar{j})$ can be modeled by a set of Σ_0^+ transitions, one transition for each set $M \subseteq \bar{j}$ with guard

$$g_{i=M}(i) \equiv \exists \bar{j} \setminus M : \phi \wedge (\bar{j} \setminus M) \neq i \wedge \psi[i/j \mid j \in M] \ .$$

Here, $\bar{\jmath} \setminus M$ is the list of all variables of $\bar{\jmath}$ that are not in M. Now, a user transition has to simulate the control transition, hence, some index i has to take it. In the control transition, this index i may be used as one or more of the j indices. Therefore, for each possible case there is one transition. Intuitively, each M is the set of all j variables that are equal to this process i.

This set of transitions can also be substituted by two more complex transitions with guard

$$g(i) \equiv \exists \bar{\jmath} : \phi \wedge \bar{\jmath} \neq i \wedge (\bigvee_{M \subsetneq \bar{\jmath}} \psi[i/j \mid j \in M])$$

and

$$g(i) \equiv \phi \wedge \psi[i/j \mid j \in \bar{\jmath}] \ .$$

The latter transition is needed for system instances with only one index. In this case, the first transition cannot be taken since there is simply no other process available to instantiate the $\bar{\jmath}$ variables. ♣

Lemma 4.18
Each Σ_0 system with synchronous control can be simulated by a Σ_0^+ system.

Proof: Let us assume that a Σ_0 system with synchronous control $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ is given with transitions $T_C \cup T_U$.

For each pair of transitions $(\tau_c, \tau_u) \in T_C \times T_U$, given by

$$\rho_{\tau_c} = \exists \bar{\jmath}_c : \phi_c \wedge \psi_c(\bar{\jmath}_c) \rightarrow \bar{y} := \bar{c}_y$$

and

$$\rho_{\tau_u}(i) = \exists \bar{\jmath}_u : \phi_u(i) \wedge \bar{\jmath}_u \neq i \wedge \psi_u(i, \bar{\jmath}_u) \rightarrow \bar{x}[i] := \bar{c}_x \ ,$$

we define for each $M \subseteq \bar{\jmath}_c$ the transition

$$\begin{aligned} \rho_{\tau_c, \tau_u, i=M}(i) \stackrel{\text{def}}{=} & \exists \bar{\jmath}_c \setminus M, \bar{\jmath}_u : \phi_c \wedge \phi_u(i) \wedge (\bar{\jmath}_c \setminus M) \neq i \wedge \bar{\jmath}_u \neq i \\ & \wedge \psi_c[i/j \mid j \in M] \\ & \wedge \psi_u(i, \bar{\jmath}_u) \\ & \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y \ . \end{aligned}$$

Exactly as in Remark 4.17, the index i making the step can be involved in the enabledness of the control transition. Hence, one has to consider all

combination of equalities of controller j variables and i. Again, this set of transitions can be substituted by two transition as explained in Remark 4.17.

This defines a new transition relation $\bar{\rho}$. Then, $\bar{\mathcal{S}} \stackrel{\text{def}}{=} (N, \mathcal{V}, \bar{\rho}, \theta)$ is bisimilar to $\mathcal{S}$ and belongs to the class Σ_0^+.

Assume a trace of $\mathcal{S}(n)$ is given. For each step in the trace, there exist τ_c and τ_u which are taken. So, there must exist a corresponding i and corresponding values for $\bar{j}_c$. Then, one of the $\rho_{\tau_c,\tau_u,i=M}$ transitions can be taken by $\bar{\mathcal{S}}(n)$ in the same state, namely the one with M set to all $\bar{j}_c$ variables instantiated to the value of i. This transition leads to the same post-state. Hence, the same trace is in $[\![\bar{\mathcal{S}}(n)]\!]$.

That each step of $\bar{\mathcal{S}}(n)$ can be simulated by a $\mathcal{S}(n)$ step follows using similar arguments. ■

The Σ_0^+ class does not correspond well to the picture of a controller connected to a number of user processes. However, this is not important for us, since we are interested in finding syntactical restrictions to impose on the general computation model, such that the verification problem is decidable. There are anyhow no compelling reasons why we should only model systems consisting of one controller and several user processes.

Next, we reduce the verification problem of Σ_0^+ system to a problem for a certain class of counter machines, namely vector addition systems with states [Rei85], systems which are defined as follows.

Definition 4.19 (VASS)
A *vector addition system with states* (shortly *VASS*) $V = (Q, (x_i)_{1 \le i \le l}, T, Q_0)$ consists of

- a finite set of states Q,
- $l \ge 1$ counters $x_1, \ldots, x_l$,
- a finite transition relation $T \subseteq Q \times \mathbb{Z}^l \times Q$, and
- a set of initial states $Q_0 \subseteq Q$.

□

Each counter may be used to hold one non-negative value. In each transition $(q, \bar{z}, q') \in T$, with $\bar{z} = (z_1, \ldots, z_l)$, which we also denote by $q \xrightarrow{\bar{z}} q'$, counter j is increased or decreased according to z_j, as long as it remains positive or zero. Otherwise, the transition cannot be taken. This idea is formally defined in the following definition.

Definition 4.20 (Semantics of VASS)
A *configuration* of a VASS V is an element $(c, \bar{v}) \in Q \times [\mathcal{C} \to \mathbb{N}]$, where $\mathcal{C} = \{x_1, \ldots, x_l\}$ is the set of counters. The set of configurations is denoted by *Conf*. We also use the vector of counter values instead of a function mapping counters to values, where we assume that an order on the counters is given.

For two configurations $(q, \bar{v})$ and $(q', \bar{v}')$, we define the step relation by

$$(q, \bar{v}) \to (q', \bar{v}') \text{ iff } \exists (q, \bar{z}, q') \in T : \bar{v}' = \bar{v} + \bar{z} \ .$$

As usual, $\to^*$ denotes the reflexive transitive closure of $\to$. □

Given this system class, we are able to formulate a couple of problems related to VASS.

Definition 4.21
The *reachability problem* for VASS can be formulated as follows: Given a state $q \in Q$, is there a $q_0 \in Q_0$ and a $v \in [\mathcal{C} \to \mathbb{N}]$ such that $(q_0, 0^l) \to^* (q, \bar{v})$?

For a second problem we need the following partial order $\preceq \subseteq \mathit{Conf} \times \mathit{Conf}$ on configurations: for all configurations $c = (q, \bar{v})$ and $c' = (q', \bar{v}')$ we define $c \preceq c'$ if and only if $q = q'$ and $\bar{v} \leq \bar{v}'$ (pointwisely).

Given this order and given a configuration $c \in \mathit{Conf}$, we can state the *coverability problem*: Is there a $q_0 \in Q_0$ and a $c' \in \mathit{Conf}$ such that $(q_0, 0^l) \to^* c'$ and $c \preceq c'$? □

We observe that the reachability problem is a special instance of the coverability problem; the reachability of $q \in Q$ can be formulated as coverability of $(q, 0^l)$.

Before we show how to solve these problems, we first investigate the close relationship between Σ_0^+ systems and VASS. Therefore, a few observations are helpful.

Lemma 4.22
Let $\mathcal{S}$ be a Σ_0^+ system, and $\sigma, \sigma' \in \Sigma_n$. Then, $(\sigma, \sigma') \models \rho$ if and only if

1. there exists $l < n$ such that $\sigma' = (\sigma : G, l \mapsto \pi_G(\sigma'), \pi_l(\sigma'))$, and
2. there exists τ and $l < n$ such that $(\sigma, \sigma') \models \rho_\tau[l/i]$.

Proof: Follows directly from the definition of the transition relation predicates of this class. ■

The following lemma shows a connection between Σ_0^+ systems and VASS.

Lemma 4.23 (Correspondence Σ_0^+ systems and VASS)
For each Σ_0^+ system $\mathcal{S}$ there exists a VASS V, such that for each trace of any instance of the system there is a corresponding computation of the VASS, and for each trace of VASS that reaches a control state of a certain set, there is a corresponding trace of some instance $\mathcal{S}(n)$.

The trace r of $\mathcal{S}(n)$ corresponds to a trace r' of V, if $\pi_G(r) = proj_1(f(r'))$ for some function f that removes some intermediate configurations from r'. Here $proj_1$ is the projection on the first component; $\pi_G(r)$ is the sequence of control states appearing in r.

Proof: Let $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ be a Σ_0^+ system built up by transitions $\tau \in T$.

The idea is to construct a VASS, which has a counter for each possible local state a user process can have. The counter simply counts the number of processes in the corresponding local state. The control state of the VASS corresponds either to the control state of $\mathcal{S}(n)$ or some intermediate state, since $\mathcal{S}$ steps are simulated by a sequence of VASS steps. So assume that the local user states are enumerated $\{s_1, \ldots, s_{|\Sigma_L|}\}$.

Let $\tau \in T$ be a transition. If this transition is taken, potentially the control state and one of the local states is modified. Moreover, it can only be taken if the system is in specific control states as well as the process that makes the step, and if some more local states, needed to enable the transition guard, are occupied in that state.

We observe that for each transition τ there exists a function $\Sigma_G \times \Sigma_L \to 2^{\Sigma_L}$ mapping pairs (c, s) to $E_{c,s}$ such that τ is enabled in state σ if

- $\pi_G(\sigma) = c$,
- the process i making the step is in state s, and
- Localstates$(\sigma, \neq i) \in E_{c,s}$.

This function is identified with the set of its triples, it is called *enabling set of* τ and is denoted by En_τ. These sets are effectively computable for each transition.

If $M \in En_\tau(c, s)$ then there exist $n \geq 1$, $k < n$, and states $\sigma, \sigma' \in \Sigma_n$ such that $\pi_k(\sigma) = s$, $\pi_G(\sigma) = c$, Localstates$(\sigma, \neq k) = M$, and $(\sigma, \sigma') \models \rho_\tau[k/i]$. We denote $(\pi_G(\sigma'), \pi_k(\sigma'))$ with $\tau(c, s)$. Note that this "result" of the transition is deterministic and unequivocal for all such σ, σ'.

To simulate a transition τ, we define for each $M \in En_\tau(c, s)$ the following transitions.

$$c \xrightarrow{\bar{z}} (\tau(c, s), M) \ ,$$

where $\bar{z} \stackrel{\text{def}}{=} -\bar{z}_s - \bar{z}_M$ with

$$\bar{z}_s(i) = \begin{cases} 0 & \text{if } s_i \neq s \\ 1 & \text{if } s_i = s \end{cases}$$

and

$$\bar{z}_M(i) = \begin{cases} 0 & \text{if } s_i \notin M \\ 1 & \text{if } s_i \in M \end{cases}$$

This transition is used to check for the enabledness of the transition. Intuitively, if this transition can be taken, it is ensured that there is a process in local state s that takes transition τ and there are the M local states present to enable it.

The second transition resets these values for M and simulates the affect of taking the transition. For $M \subseteq \Sigma_L$, $c' \in \Sigma_G$, and $s' \in \Sigma_L$ we define

$$(c', s', M) \xrightarrow{\bar{z}} c'$$

with $\bar{z} \stackrel{\text{def}}{=} \bar{z}_M + \bar{z}_{s'}$.

What is missing is the initialization of the VASS. One can observe that the initial state predicate θ only depends on the "set" of local states, see Corollary 3.42. Consequently, there exists a finite set $I \subseteq \Sigma_G \times (2^{\Sigma_L} \setminus \emptyset)$ such that $\sigma \models \theta$ iff $(\text{Globalstates}(\sigma), \text{Localstates}(\sigma)) \in I$. This set is effectively computable.

The following set of transitions can be used to reach a configuration that corresponds to an initial state of $\mathcal{S}$ for some instance $n \geq 1$.

For each $(c, M) \in I$ we define a transition that adds all the M processes:

$$q_0 \xrightarrow{\bar{z}_M} (c, M)$$

A valid inital configuration has only processes with local state in M, but there may be more than one process for each of these local states. Therefore, the following transitions add non-deterministically more processes starting in a local state of M. For each $s \in M$ take the following transition:

$$(c, M) \xrightarrow{\bar{z}_s} (c, M)$$

The last transition leaves the initialization part of the VASS computation. The configuration corresponding to some initial state is now completed and $\mathcal{S}$ steps can be simulated:

$$(c, M) \xrightarrow{\bar{0}} c$$

We define the following VASS $V = (Q, (x_i)_{1 \leq i \leq l}, T_V, Q_0)$ given by

- $Q \stackrel{\text{def}}{=} \Sigma_G \cup (\Sigma_G \times \Sigma_L \times 2^{\Sigma_L}) \cup (\Sigma_G \times 2^{\Sigma_L}) \cup \{q_0\}$,
- $x_i \stackrel{\text{def}}{=} s_i$ and $l \stackrel{\text{def}}{=} |\Sigma_L|$,
- the set of all transitions T_V described above, and
- $Q_0 \stackrel{\text{def}}{=} \{q_0\}$.

This VASS is denoted by $\mathit{VASS}(\mathcal{S})$.

Define the following function α that relates states of a system instance $\mathcal{S}(n)$ with configurations of V. For $\sigma \in \Sigma_n$ define $\alpha(\sigma) \stackrel{\text{def}}{=} (\pi_G(\sigma), \bar{v})$ such that $\bar{v}(i) = |\{j \mid j < n \wedge \pi_j(\sigma) = s_i\}|$.

Then, for each run r of $\mathcal{S}(n)$, there exists a corresponding trace of V, namely r' such that first the configuration corresponding to $r(0)$ is created, and from this position c on, $\alpha(r(i)) = \alpha(r'(2i + c))$ and $\alpha(r'(2i + 1 + c))$ is the intermediate state resulting from taking the transition τ that $\mathcal{S}(n)$ takes in the step from $r(i)$ to $r(i + 1)$ (which always exists).

Similarly, let r be a trace of V that started in configuration $(q_0, \bar{0})$ and reaches a control state of Σ_G (this is the set described in the lemma). Let c be the position of the first control state of Σ_G, so $r(c) = (c, \bar{v})$ with $c \in \Sigma_G$. Then, there exists $\sigma \in \Sigma_n$ with $n > 0$ and $n \stackrel{\text{def}}{=} \sum_{1 \leq i \leq l} \bar{v}(i)$. We can conclude $\sigma \models \theta$.

Then, there exists a corresponding trace of $\mathcal{S}$, for each two V steps simulating one transition τ, one only has to choose one process in the right local state and let that process take transition τ. The function f mentioned in the lemma simply removes all intermediate configurations $(c, \bar{v})$ from the trace (such that $c \notin \Sigma_G$). ■

We note that this simulation is indeed not "precise" with respect to the user processes. This means, we cannot use the construction given above to reduce the verification problem for arbitrary properties to the verification of the corresponding VASS. The reason is that one loses the possibility to focus on one process and observe its sequence of local states by the transformation into a VASS.

Lemma 4.24
The VASS $V = \mathit{VASS}(\mathcal{S})$ has the following property.

- In any computation of V which reaches a non-intermediate configuration, all following non-intermediate configurations have the same *weight* (which is the sum over the values of all counters).

Proof: This lemma holds by construction of $VASS(\mathcal{S})$. ■

Definition 4.25 (Partial order)
Given a set A and a relation $\preceq \subseteq A \times A$. $\preceq$ is a *partial order* if and only if it has the following properties:

- reflexivity: $\forall a \in A : a \preceq a$
- transitivity: $\forall a, b, c \in A : a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$
- anti-symmetry: $\forall a, b \in A : a \preceq b \wedge b \preceq a \Rightarrow a = b$

□

Definition 4.26 (Well quasi-ordering)
A partial order $\preceq$ on A is called a *well quasi-ordering* if and only if for each infinite sequence $\alpha \in A^\omega$ there exist i, j such that $i < j$ and $\alpha(i) \preceq \alpha(j)$.
□

It is obvious that every well quasi-ordering is also well founded (which means that there do not exist any infinite decreasing sequences in A).

Lemma 4.27
The $\preceq$ order on configurations is a well-quasi ordering.

Proof: We prove by induction on the number on counters a stronger result: For all sequences α of counter values, there exists an infinite sub-sequence of α, which is monotonically increasing.

Base case: Let us assume that we have only one counter, and that α is a sequence of counter values. If there is one counter value appearing infinitely often in α, choose this sub-sequence, which is obviously monotonically increasing. If all values occur only finitely often, then there must be infinitely many values, so they must grow beyond every bound. So we conclude that there is a sub-sequence increasing monotonically.

Inductive step: Let us assume there are $n + 1$ counters, and that α is a sequence of counter values. Then, if we only consider the first n counters, then the projection of α is a sequence of n values, so by induction hypothesis, there exists a monotonically increasing sub-sequence $(\alpha(i_j))_{j \in \mathbb{N}}$. Now, using the argumentation of the basic case for the position $n+1$, we find a monotonically increasing sub-sequence for this position. This sub-sequence is then also a monotonically increasing sub-sequence for all $n + 1$ counters. ■

Lemma 4.28
The coverability problem for VASS is decidable.

Proof: Let $V = (Q, (x_i)_{1 \leq i \leq l}, T, Q_0)$ be a VASS, and c be a configuration of V. The idea is to make a backward analysis starting with the given configuration. If this analysis terminates and there is some initial configuration in the computed set, a covering configuration of c can be reached.

For transition $t = (q, \bar{z}, q') \in T$, $p \in Q$, and $C \subseteq Conf$ we define

$$Pre_t(p, \bar{v}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } p \neq q' \vee (\bar{v} - \bar{z}) \not\geq 0 \\ \{(q, \bar{v} - \bar{z})\} & \text{otherwise} \end{cases}$$

$$Pre_t(C) \stackrel{\text{def}}{=} Min(\bigcup_{c \in C} Pre_t(c))$$

$$Pre(C) \stackrel{\text{def}}{=} Min(\bigcup_{t \in T} Pre_t(C))$$

The *Min* operator chooses only the minimal elements of the given sets, so for a finite set $C \subset Conf$ we define $Min(C)$ as the set $C' \subseteq C$ such that

- for all $c \in C$ there exists $c' \in C'$ such that $c' \preceq c$, and
- for all $c, c' \in C'$, $c \preceq c'$ implies $c = c'$.

For each set $C \subseteq Conf$ let $[\![C]\!] \stackrel{\text{def}}{=} \{c' \mid \exists c \in C : c \preceq c'\}$ (this is the upward closure of C).

We note that $Pre(C)$ as well as $Min(C)$ is effectively computable for all finite $C \subseteq Conf$, and $[\![C]\!] = [\![Min(C)]\!]$.

The following equations give an algorithm for solving the coverability problem.

$$M_0 \stackrel{\text{def}}{=} \{c\}$$

$$M_{i+1} \stackrel{\text{def}}{=} Min(M_i \cup Pre(M_i))$$

This fixed point iteration converges: assume that $M_{i+1} \neq M_i$ for all i. Then, $[\![M_{i+1}]\!] \supsetneq [\![M_i]\!]$, so choose some $c_i \in [\![M_i]\!] \setminus [\![M_{i-1}]\!]$ (with $M_{-1} \stackrel{\text{def}}{=} \emptyset$). Since $\preceq$ is a well-quasi ordering, there exist $i < j$ such that $c_i \preceq c_j$. Consequently, $c_j \in [\![M_i]\!]$, contradiction! ∎

Theorem 4.29
The verification problem for Σ_0^+ systems is decidable for invariance properties.

Proof: Let $\mathcal{S}$ be a system with local states $\Sigma_L = \{s_1, \ldots, s_l\}$ and assume an invariance property $\Box\phi(i_1, \ldots, i_k)$. We have to prove $\mathcal{S}(\geq n_0) \models \forall_d \Box\phi(i_1, \ldots, i_k)$, which can be reduced to $\mathcal{S}(\geq n_0) \models \forall_d \Box\phi(0, \ldots, k-1)$.

First construct the VASS $V_\mathcal{S} = \mathit{VASS}(\mathcal{S})$ for $\mathcal{S}$. There is a finite set of "bad states"

$$B_{\neg\phi} \stackrel{\text{def}}{=} \{(g, a_0, \ldots, a_{k-1}) \mid (\sigma : i \mapsto a_i, G \mapsto g) \models \neg\phi(0, \ldots, k-1)\}$$

This set is effectively computable. For each $b = (g, a_0, \ldots, a_{k-1}) \in B_{\neg\phi}$ let $c_b = (g, \bar{v})$ with $\bar{v}(i) = |\{j \mid a_j = s_i\}|$ be the configuration corresponding to b.

In case $n_0 > 1$ one has to ensure that each configuration corresponding to an initial state that $V_\mathcal{S}$ reaches has "enough" processes. This can be done by counting the number of local states added in these first steps in the control state, modifying $V_\mathcal{S}$ slightly.

Now use the backward analysis as given in the proof of Lemma 4.28, let M be the result of this analysis. All one has to check is whether the initial configuration is a member of M, which can be done effectively. If this is the case for any of these "bad" configurations, then some instance of $\mathcal{S}$ can reach a state violating ϕ, so ϕ is no invariant for all instances of $\mathcal{S}$. Otherwise it is invariant. ■

For the more general verification problem we apply the automata-theoretic approach for LTL model-checking [VW86], see also Section 5.4. To prove that property ϕ holds, one builds first the automaton $\mathcal{A}_{\neg\phi}$ and then the synchronous product of the system and $\mathcal{A}_{\neg\phi}$, the result is an automaton that accepts the intersection of both languages. This language is empty iff the system satisfies ϕ.

Definition 4.30
Let $\mathcal{S}$ be a parameterized system. Let r be a computation of $\mathit{VASS}(\mathcal{S})$. We define $\mathit{contr}(r)$ as the sequence of all control states appearing in non-intermediate configurations of r. □

Lemma 4.31
Let ϕ be a control property and $\mathcal{S}$ a parameterized system. Every instance of $\mathcal{S}$ satisfies ϕ iff $\mathit{contr}(r)$ satisfies ϕ for each computation r of $\mathit{VASS}(\mathcal{S})$ which contains at least one non-intermediate configuration.

In this case we say $\mathit{VASS}(\mathcal{S})$ satisfies ϕ, denoted by $\mathit{VASS}(\mathcal{S}) \models \phi$.

Proof: By Lemma 4.23 we know that every trace of $\mathit{VASS}(\mathcal{S})$ (with some non-intermediate configuration) corresponds to a run of some instance, and vice versa. ■

Lemma 4.32
Let ϕ be a control property and $\mathcal{S}$ a parameterized system. $VASS(\mathcal{S}) \not\models \phi$ iff there exists a finite computation r of $VASS(\mathcal{S})$ such that

1. $r = \alpha\beta$ and
   - α starts in the initial configuration and β is a cycle, and
   - there exists a run of $\mathcal{A}_{\neg\phi}$ over $contr(\alpha)contr(\beta)$ such that the tail of the run over $contr(\beta)$ is a cycle containing a final state, or
2. r is a deadlocked computation and $contr(r) \not\models \phi$.

Proof: Let $v = VASS(\mathcal{S})$.

$V \not\models \phi$
iff $\neg\forall$ computations r of $V : contr(r) \models \phi$
iff $\exists$ computation r of $V : contr(r) \models \neg\phi$
iff $\exists$ finite deadlocked computation r of $V : contr(r) \models \neg\phi$
or $\exists$ infinite computation r of $V : contr(r) \models \neg\phi$
iff $\exists$ finite deadlocked computation r of $V : contr(r) \models \neg\phi$
or $\exists$ a finite computation r of V and
$\exists$ run $\alpha\beta$ of $\mathcal{A}_{\neg\phi}$ over $contr(r)$
such that β is a cycle containing a final state

We prove the last equivalence in detail (only the part for infinite computations). Assume there exists r such that $contr(r) \models \neg\phi$. Then there exists an accepting run $r_{\mathcal{A}}$ of $\mathcal{A}_{\neg\phi}$ over $contr(r)$.

Since the sets of states of V and $\mathcal{A}_{\neg\phi}$ are finite and the weights of r remain constant for non-intermediate configurations (by Lemma 4.24), there exists position i such that $r(i)$ is non-intermediate and there are infinitely many j with $r(i) = r(j)$ and $r_{\mathcal{A}}(i) = r_{\mathcal{A}}(j)$.

Since there are infinitely many accepting states in $r_{\mathcal{A}}$, one of these j is large enough so that there is an accepting state between position i and j. This shows the existence of a cycle $\alpha\beta$.

The other direction is straightforward since the cycle can be taken infinitely often by both the VASS and the property automaton. ■

By defining the synchronous product of a VASS and a property automaton we reduce this problem to a problem about VASS.

Definition 4.33 ($VASS(\mathcal{S}, \mathcal{A})$)
Let $\mathcal{A}$ be a Büchi automaton given by (Q, A, Δ, Q_0, F) describing a temporal (controller) property of $\mathcal{S}$ instances, with states Q, alphabet $A = 2^{\mathcal{P}}$ (where $\mathcal{P}$ are atomic propositions of the form $y = v$ for $y \in \mathcal{V}_Y$ and $v \in \text{type}(y)$), transitions $\Delta \subseteq Q \times A \times Q$, initial states $Q_0 \subseteq Q$, and accepting states $F \subseteq Q$.

Construct $V_{\mathcal{S}} = VASS(\mathcal{S}) = (Q_V, (x_i)_{1 \le i \le l}, T_V, Q_{V,0})$ as in the proof of Lemma 4.23 (taking n_0 into account).

For $s \in \Sigma_G$ let $prop(s) = \{y = v \mid s(y) = v\}$ be the set of all propositions corresponding to s. The automaton $\mathcal{A}$ uses subsets of these propositions as its alphabet.

We construct a VASS $V \stackrel{\text{def}}{=} (Q_V \times Q, (x_i)_{1 \le i \le l}, T, Q_{V,0} \times Q_0)$ that is intuitively the synchronous product of $V_{\mathcal{S}}$ and $\mathcal{A}$. This VASS is also denoted by $VASS(\mathcal{S}, \mathcal{A})$.

We say that a configuration is *proper* similar as in [GS92] if the control state of that configuration is in $\Sigma_G \times Q$. These configurations correspond directly to states of some instance $\mathcal{S}(n)$. Other configurations are called *intermediate*. A proper configuration such that the automaton state is accepting is called *accepting* configuration.

There are three types of interesting transitions: The first type are transitions to build up the initial state, ending with the one transition $(c, M) \xrightarrow{\bar{0}} c$ that reaches the configuration that corresponds to the "initial" state of some instance $\mathcal{S}(n)$ (the first three items below correspond to transitions of this type). The second one is the set of transitions from configurations that represent a $\mathcal{S}(n)$ state to intermediate configurations. The last set of transitions leave intermediate configurations, reaching proper configurations.

- For each $\mathcal{S}$-transition $q_0 \xrightarrow{\bar{z}_M} (c, M) \in T_V$ and $q_0^{\mathcal{A}} \in Q_0$ we choose $(q_0, q_0^{\mathcal{A}}) \xrightarrow{\bar{z}_M} ((c, M), q_0^{\mathcal{A}}) \in T$.

- For each $(c, M) \xrightarrow{\bar{z}_s} (c, M) \in T_V$ and $q_0^{\mathcal{A}} \in Q_0$ choose $((c, M), q_0^{\mathcal{A}}) \xrightarrow{\bar{z}_s} ((c, M), q_0^{\mathcal{A}}) \in T$.

- For each $(c, M) \xrightarrow{\bar{0}} c \in T_V$ and $q_0^{\mathcal{A}} \in Q_0$ choose $((c, M), q_0^{\mathcal{A}}) \xrightarrow{\bar{0}} (c, q) \in T$ if and only if $(q_0^{\mathcal{A}}, prop(c), q) \in \Delta$.

  Intuitively, the automaton $\mathcal{A}$ encoded in the control checks the initial control state of the corresponding system instance $\mathcal{S}(n)$. This transition reaches the first proper state.

- For each $(c', s', M) \xrightarrow{\bar{z}} c' \in T_V$ with $c' \in \Sigma_G$ we choose a transition $((c', s', M), q) \xrightarrow{\bar{z}} (c', q') \in T$ for each $q, q' \in Q$ such that

$(q, prop(c'), q') \in \Delta$.

The automaton $\mathcal{A}$ checks the new control state c' that is reached.

- For each $c \xrightarrow{\bar{z}} (c', s', M) \in T_V$ with $c' \in \Sigma_G$ we choose $(c, q) \xrightarrow{\bar{z}} ((c', s', M), q) \in T$ for each $q \in Q$.

  Intuitively, the automaton $\mathcal{A}$ stutters in this step, since the control reaches an intermediate state.

□

Lemma 4.34
Let $\mathcal{S}$ be a parameterized system and ϕ be a control property. $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$ has a

- finite computation of the form $\alpha\beta$ such that β is a cycle containing an accepting configuration iff there exists a finite computation of $VASS(\mathcal{S})$ satisfying the conditions of Lemma 4.32 (1.)
- finite deadlocked computation such that the sequence of control states of all proper configurations in it satisfies $\neg\phi$ iff there exists a finite deadlocked computation r of $VASS(\mathcal{S})$ such that $contr(r) \not\models \phi$.

Proof: This lemma holds by construction, $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$ is simply the synchronous product of $VASS(\mathcal{S})$ and $\mathcal{A}_{\neg\phi}$. ■

Lemma 4.35 ([GS92])
There exists a finite path of $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$ of the form $\alpha\beta$ such that

- α starts in the initial configuration and
- β is a cycle in which an accepting configuration appears,

if and only if there exists such a path of length

$$\mathcal{O}(2^{k_1 \cdot p \cdot \log(p) \cdot 2^{k_1 \cdot m \cdot \log(m)}}) \ ,$$

where k_1 is a constant, p is the number of states of $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$, and m is the number of local states $|\Sigma_L|$ of $\mathcal{S}$.

Lemma 4.36
Let $\mathcal{S}$ be a parameterized system and ϕ be a control property. It is decidable whether $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$ has a finite deadlocked computation such that its sequence of proper control states satisfies $\neg\phi$.

Proof: Let $V = VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$.

First we note that a control state sequence r of a deadlocked computation of V satisfies ϕ iff $r \cdot c^\omega$ satisfies ϕ, where c is the final proper control state of r. Hence, we try to identify configurations which correspond to deadlock states, such that its stutter extension satisfies the property.

We call a configuration a of V *stutter-accepting*, if there exists a trace of $\mathcal{A}_{\neg\phi}$ starting in the property automaton state of a which accepts c^ω, where c is the control state of a. For each control state c and automaton state q, it is clearly decidable whether (c, q) is stutter-accepting or not.

Define $En(c, s) \stackrel{\text{def}}{=} \bigcup_\tau En_\tau(c, s)$.

For each control state c, and local state s, check for each $M \in 2^{\Sigma_L} \setminus En(c, s)$, whether a stutter-accepting configuration covering (c, s, M) is reachable. A configuration x covers (c, s, M), if c is the control state of x, $x - \bar{z}_M - \bar{z}_s \geq 0$, and every counter in x is zero if the corresponding local state is not in $M \cup \{s\}$.

If any of these stutter-accepting configurations is reachable (this check can be done efficiently), then there exists a deadlocking computation of V which satisfies ϕ.

On the other hand, if some deadlocking computation exists, it must end in a configuration covering (c, s, M) for some $M \notin En(c, s)$. ■

Theorem 4.37
The verification problem for Σ_0^+ systems is decidable for controller properties.

Proof: Let $\mathcal{S}$ be a parameterized system and ϕ be a controller property. Proving $\mathcal{S}(n) \models \phi$ is by Lemma 4.31 equivalent to proving $VASS(\mathcal{S}) \models \phi$. The latter is by Lemma 4.32 equivalent to proving that no infinite computation of the form $\alpha\beta$ exists where β contains an accepting state, and that no deadlocked computation exists satisfying $\neg\phi$. By Lemma 4.34, this problem can be reduced to a problem on $VASS(\mathcal{S}, \mathcal{A}_{\neg\phi})$.

By Lemma 4.35 the existence of such an infinite computation can be decided.

By Lemma 4.36 the existence of such a deadlocked computation is also decidable, showing decidability of the verification problem. ■

Corollary 4.38
The verification problem for Σ_0^+ systems is decidable for safety properties of the controller.

Proof: This corollary is a consequence of Theorem 4.37. For verification of safety properties, one can also apply a second approach using an automaton

$\mathcal{A}$ over finite words that accepts all finite "bad prefixes" of counterexamples instead of the Büchi automaton $\mathcal{A}_{\neg\phi}$ [KV99].

Then, one can use the backward analysis as in the case of invariance properties before to decide whether some "bad" $VASS(\mathcal{S}, \mathcal{A})$ control state from the set $\{(q, a) \mid q \in \Sigma_G, a \in F\}$ is reachable. ■

User properties

We follow the idea presented in [GS92] and reduce properties involving user processes to controller properties.

The next transformation can be used to move one process inside the control. Intuitively, we add non-array copies of the array variables with the same type, the state of these variables corresponds to the local state of one process. Then, for each transition we add a couple of new transitions that take this "process inside the controller" into account.

Definition 4.39 (Transformation)
Let $\mathcal{S}$ be a Σ_0^+ system. Let $\tilde{x}$ be a new (control) variable of type type(x) for each array variable x. We abbreviate the list of index variables from $\bar{j}$ not in some set M with $\bar{j} \setminus M$. We shortly write $\psi[\tilde{x}/x[j]]$ instead of $\psi[\tilde{x}/x[j] \mid x \in \mathcal{V}_X]$, always meaning the whole set of variables $\mathcal{V}_X$.

Intuitively, one process p moves into the control, so the $\tilde{x}$ variables replace the array entries $x[p]$. One has to consider the cases when this process makes a step and when this process p is referred to by transitions of other processes.

For each transition τ of $\mathcal{S}$

$$\begin{aligned}\rho_\tau(i) \equiv \exists \bar{j} : \phi(i) \wedge \bar{j} \neq i \wedge \psi(i, \bar{j}) \\ \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y\end{aligned}$$

with $\bar{j} = j_1, \ldots, j_l$ and $l \geq 1$, we define a set of transitions of the transformed system. First, we take for $M \subseteq \bar{j}$, $M \neq \emptyset$

$$\begin{aligned}\rho_{\tau, i=p, M}(i) \equiv \exists \bar{j} \setminus M : \phi[\tilde{x}/x[i]] \wedge (\bar{j} \setminus M) \neq i \\ \wedge (\psi[\tilde{x}/x[i]])[i/j \mid j \in M] \\ \rightarrow \tilde{\bar{x}} := \bar{c}_x; \bar{y} := \bar{c}_y\end{aligned}$$

Intuitively, this transition simulates that process p takes transition τ. Instead of modifying an array entry, the new variables must be changed. Obviously, the simulating transition must be taken by some other index. Therefore, we choose one of the indices to which $\bar{j}$ variables are instantiated.

Moreover, we define one transition for each $M \subseteq \bar{j}$:

$$\begin{aligned}\rho_{\tau,p=M}(i) \equiv \exists \bar{j} \setminus M : \phi(i) \wedge (\bar{j} \setminus M) \neq i \\ \wedge \psi[\tilde{x}/x[j] \mid j \in M] \\ \rightarrow \bar{x}[i] := \bar{c}_x; \bar{y} := \bar{c}_y\end{aligned}$$

(for $M = \emptyset$ this is the original transition ρ_τ). These transitions correspond to the case that a process $i \neq p$ takes the transition, and some of the $\bar{j}$ variables, namely M, are instantiated with p.

With $tr(\mathcal{S})$ we denote the resulting system with extended set of variables and transition relation that is built up by transitions $\rho_{\tau,i=p}$ and $\rho_{\tau,p=M}$ for $M \subseteq \{j_1, \ldots, j_k\}$.

In a similar way, an LTL property $\phi(i_1, \ldots, i_k)$ with $k \geq 1$ is modified:

$$tr(\phi) \stackrel{\text{def}}{=} \phi[\tilde{x}/x[i_1] \mid x \in \mathcal{V}_X] \ .$$

This formula has free variables $i_2, \ldots, i_k$. □

The next lemma states that this transformed system essentially is the old one.

Lemma 4.40
For every Σ_0^+ system $\mathcal{S}$ and LTL property $\phi(i_1, \ldots, i_k)$, and $n > k \geq 1$, $\mathcal{S}(n) \models \phi(0, \ldots, k-1)$ iff $tr(\mathcal{S})(n-1) \models tr(\phi)(0, \ldots, k-2)$.

Proof: Let us assume r is a trace of $\mathcal{S}(n)$. Let r' be the sequence of $tr(\mathcal{S})(n-1)$ states such that for each h, $\pi_j(r'(h)) = \pi_{j+1}(r(h))$, the control states projected on the $\mathcal{V}_Y$ variables are identical, and the $\tilde{x}$ variables in each state $r'(h)$ have the state $\pi_0(r(h))$.

For each step in r there exists τ and index $l < n$ such that $(r(h), r(h+1)) \models \rho_\tau(l)$. Transition τ existentially quantifies $\bar{j} = j_1, \ldots, j_e$. Hence, there exist corresponding values $c_1, \ldots, c_e$ that satisfy the transition guard.

If $l = 0$, then $tr(\mathcal{S})(n-1)$ can take transition $\rho_{\tau,i=p,M}$ for index c_1 and set $M = \{j_x \mid c_x = c_1\}$, leading to a corresponding post-state; each index variable $j_x \notin M$ can be instantiated by $c_x - 1$ in this case.

Otherwise, let $M = \{j_x \mid c_x = 0\}$. Transition $\rho_{\tau,p=M}$ can then be taken, leading to post-state $r'(k+1)$. Again, the j_x variables with $j_x \notin M$ can be instantiated by $c_x - 1$.

It is easy to see that $r' \models \phi(0, \ldots, k-2)$ if $r \models \phi(0, \ldots, k-1)$.

One can prove in a similar way that for each trace of $tr(\mathcal{S})(n-1)$ there is a corresponding trace of $\mathcal{S}(n)$.

Consequently, if $\mathcal{S}(n) \not\models \phi(0, \ldots, k-1)$, there exists a counterexample that satisfies $\neg\phi(0, \ldots, k-1)$, and hence, there is a corresponding counterexample of $tr(\mathcal{S})(n-1)$ that violates $\phi(0, \ldots, k-2)$, and vice versa. ■

Theorem 4.41
The verification problem for Σ_0^+ systems is decidable.

Proof: Assume we have to show $\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k)$. This problem can be reduced to showing $\mathcal{S}(\geq n_0) \models \phi(0, \ldots, k-1)$ due to Lemma 3.47.

Now we use the transformation given in Definition 4.39 k times and obtain a system $\mathcal{S}'$ and an LTL formula ϕ' without free index variables.

Check whether $\mathcal{S}'(\geq n_0 - k) \models \phi'$. If $n_0 \leq k$, check additionally whether $\mathcal{S}(k) \models \phi(0, \ldots, k-1)$ holds. Answer "valid" if no counterexample is found, otherwise return "invalid".

By Lemma 4.40, checking $\mathcal{S}'$ suffices to show correctness for all system instances $\mathcal{S}(n)$ with $n > k$. For $n = k$ this is proven directly, if necessary. ■

Remark 4.42
Since the class of Σ_0 systems with asynchronous control is clearly a subclass of the Σ_0^+ systems, this construction also shows decidability for the verification problem for Σ_0 systems with asynchronous control and all verification properties, using a different approach as the result presented in Section 4.1.1.

The results in Section 4.1.1 only apply for several subproblems of the general verification problem for Σ_0 systems with asynchronous control, namely for safety property verification, for non-blocking systems, and for the weak system class.

Comparing the results shows that the complexity of the decision procedures based on verifying system instances up to a certain bound are better. Let k be the number of existential quantifications in the property, ϕ be the LTL part of the property, $c \stackrel{\text{def}}{=} |\Sigma_G|$, and $u \stackrel{\text{def}}{=} |\Sigma_L|$. Then, the verification using the procedure of Section 4.1.1 can be done in time

$$\mathcal{O}(2^{|\phi|} \cdot c \cdot 2^{(u+k+1)\cdot \log u}) \ .$$

We use the known result that an LTL property ϕ can checked for a finite state system $\mathcal{S}$ in time $\mathcal{O}(2^{|\phi|} \cdot |\mathcal{S}|)$, see also Section 5.4. Model-checking is linear in the system size, but the system to check here is $\mathcal{S}(|\Sigma_L| + k + 1)$. This is a parallel system, hence the size of $\mathcal{S}(|\Sigma_L| + k + 1)$ is exponential in the size of the user process. This exponential blowup is called *state explosion.*

The bound used here is the bound of the decision procedure for non-blocking systems.

In contrast, the decision procedure based on the VASS can be applied in time

$$h(|\phi|, size(c), size(u)) + \mathcal{O}(2^{\mathcal{O}((c \cdot u^k)^a \cdot 2^{b \cdot (|\phi| + u \cdot \log u)})}) \ ,$$

where a and b are some constants and h is a polynomial function and *size* is the size of the description of the corresponding process including the transitions.

The bound is taken from Theorem 3.6 in [GS92]. The size of the control is in our case $c \cdot u^k$: from Definition 4.39 one can see that by moving one user process into the control, new control variables are introduced. Hence, the resulting system has control states $\Sigma_G \times \Sigma_L$. This transformation is applied k times. ♣

4.2 Verification of Σ_1 Systems

In this section we investigate the verification problem for Σ_1 systems, which allow to quantify universally over other processes. We show that this problem is decidable for asynchronous control, and prove undecidability in the synchronous case.

4.2.1 Decidability for Asynchronous Control

In this section we focus on the Σ_1 system class with asynchronous control. We first restrict to safety properties, and give a decidability result based on the following observation.

Having only universal quantification over process states at hand, if a transition can be taken in a particular state, then one can safely remove a process (which does not change its local state in that step), and the same transition remains possible.

So, choosing the smallest number of processes, which depends both on the particular initial state predicate and, of course, the property which is to verify, a combination of local states is reachable whenever it is reachable for *any* instance.

In a second step we show how to extend this result to general properties for non-blocking systems, and for general Σ_1 systems.

So assume throughout this section that $\mathcal{S}$ is a parameterized system in the class Σ_1 with asynchronous control, with initial state predicate $\theta = \exists_N i_1, \ldots, i_l. \forall_N i : \phi_\theta(i, i_1, \ldots, i_l)$, and transition relation predicate ρ.

Lemma 4.43
For all $n, \tilde{n} \in \mathbb{N}$, $\sigma, \sigma' \in \Sigma_n, \tilde{\sigma} \in \Sigma_{\tilde{n}}$, and $k < n, \tilde{k} < \tilde{n}$ with

- $\text{Localstates}(\sigma, \neq k) \supseteq \text{Localstates}(\tilde{\sigma}, \neq \tilde{k})$,
- $\pi_G(\tilde{\sigma}) = \pi_G(\sigma) = \pi_G(\sigma')$,
- $\pi_k(\sigma) = \pi_{\tilde{k}}(\tilde{\sigma})$, and $\pi_k(\sigma) \neq \pi_k(\sigma')$,

the following holds:

$$(\sigma, \sigma') \models \rho \text{ implies } (\tilde{\sigma}, \tilde{\sigma}') \models \rho \ ,$$

for $\tilde{\sigma}' \stackrel{\text{def}}{=} (\tilde{\sigma} : \tilde{k} \mapsto \pi_k(\sigma'))$.

A similar proposition holds also for controller steps. The first prerequisite reduces in this case to $\text{Localstates}(\sigma) \supseteq \text{Localstates}(\tilde{\sigma})$.

Proof: For Σ_1 systems, a transition guard has the form

$$\forall \bar{j} : \phi(i) \wedge (\bar{j} \neq i \Rightarrow \psi(i, \bar{j})) \ .$$

The validity of this formula depends only on the local states of process i and the quantified indices $\bar{j}$. If it is valid for each combination of indices $\bar{j}$ in σ, then for every combination of indices $\bar{j}$ of instance $\mathcal{S}(\tilde{n})$, it must also evaluate to true. Consequently, the same transition can be taken. Obviously the same values are assigned as in the step of $\mathcal{S}(n)$.

The result for controller steps is proved similarly. ■

The next lemma formalizes the idea that one can remove one process if there are "enough" processes left such that the reduced trace can satisfy θ and ϕ.

Lemma 4.44
Let $\phi(i_1, \ldots i_k)$ be an LTL formula, and let l be the number of existential quantifications in θ. For all $n \geq k + l$ and $n_1, \ldots, n_k < n + 1$, with $n_i \neq n_j$ for $i \neq j$, if

- α is a prefix of a trace in $[\![\mathcal{S}(n+1)]\!]$, and
- $\alpha \models \phi(n_1, \ldots, n_k)$,

then there exist a prefix α' of a trace in $[\![\mathcal{S}(n)]\!]$, and $n'_1, \ldots, n'_k < n$, with $n'_i \neq n'_j$ for $i \neq j$, such that

$$\alpha' \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: Let us assume that $\alpha \models \phi(n_1, \ldots, n_k)$. Since α is a prefix of a trace, $\alpha(0) \models \theta$. Consequently, there exist $j_1, \ldots, j_l$ such that $\alpha(0) \models \forall_N i : \phi_\theta(i, j_1, \ldots, j_l)$.

Since $n \geq k + l$, there exists $j < n + 1$, $j \notin \{n_1, \ldots, n_k, j_1, \ldots, j_l\}$. By Lemma 3.46 (this lemma is also true for prefixes instead of full traces), there exists a prefix α' of another trace in $[\![\mathcal{S}(n+1)]\!]$, such that $\alpha' \models \phi(n'_1, \ldots, n'_k)$, where the n'_j are the same as n_j, only that n and j are swapped, and α' is constructed from α by swapping the local traces of index j and index n.

Construct β from α' by removing the component n, and removing every stuttering step afterwards. This results in a sequence of system instance $\mathcal{S}(n)$. By Lemma 3.41 we conclude $\beta(0) \models \theta$. Lemma 4.43 states that every step in this sequence is a valid step, so β is a (prefix of a) trace of $\mathcal{S}(n)$. ■

In case α is finite, also β is finite. But even if α is maximal, it is not clear whether β is: since one local state is removed, *more* transitions may be enabled in the final state of β than in the final state of α.

And also in the case that the removed process j was the only process with infinitely many steps in α, the sequence β is finite; in this case it is also likely that β may be prolonged, such that it is not maximal.

But using a slightly higher bound, one can modify the construction such that infinite traces are translated to infinite traces of a smaller system, as the next corollary shows.

Corollary 4.45
Let $\phi(i_1, \ldots i_k)$ be an LTL formula, and l as in Lemma 4.44. For all $n > k+l$ and $n_1, \ldots, n_k < n+1$, with $n_i \neq n_j$ for $i \neq j$, if

- $\alpha \in [\![\mathcal{S}(n+1)]\!]$ is an infinite trace, and
- $\alpha \models \phi(n_1, \ldots, n_k)$,

then there exist an infinite trace $\alpha' \in [\![\mathcal{S}(n)]\!]$ and $n'_1, \ldots, n'_k < n$, with $n'_i \neq n'_j$ for $i \neq j$, such that

$$\alpha' \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: The difference to Lemma 4.44 is that we need to find an infinite trace instead of a possibly finite prefix. This can be obtained by observing that in an infinite trace, there must be at least one process (or the controller) with infinitely many steps.

In an instance $\mathcal{S}(n)$ with $n > k+1$, there is room for one "extra" process that can be used for this activity, even if the ϕ-processes and the processes used for θ make only finitely many steps.

If there is no process i, such that the i'th local state changes infinitely often, then there must be a stuttering transition that is enabled (since the trace of a non-blocking system is always infinite). So, if this stuttering is no controller transition, one can always choose this "extra" process such that it is the one that stutters in some local state forever. ■

Note the difference between Lemma 4.44 and Corollary 4.45: in the lemma, $n \geq k+l$ can be used, whereas the corollary holds for $n > k+l$.

Theorem 4.46
The verification problem for Σ_1 systems with asynchronous control is decidable for safety properties.

Proof: Assume that we have to prove $\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k)$, where ϕ is a safety property. Let l be the number of existential quantifiers. Then prove whether

$$\mathcal{S}(n) \models \phi(0, \ldots, k-1) \ ,$$

for all n, with $k \leq n \leq k+l$ and $n_0 \leq n$ (in case $n_0 > k+l$ check only for n_0). Return the result "valid" if all proofs succeed, and return "false" otherwise.

A negative answer is obviously correct since $n_0 \leq n$.

To prove that positive answers are correct, assume that the answer is "valid" and there is an instance $\mathcal{S}(n)$ such that $\mathcal{S}(n) \not\models \forall_d \phi(i_1, \ldots, i_k)$. If $n < k$, then $\mathcal{S}(n)$ fulfills the property trivially. With Lemma 3.47 we conclude that $n > k + l$ and $n > n_0$.

Then, there exist $n_1, \ldots, n_k < n$, with $n_i \neq n_j$ for $i \neq j$, and $\alpha \in [\![\mathcal{S}(n)]\!]$ such that $\alpha \models \neg\phi(n_1, \ldots, n_k)$.

Repeated application of Lemma 4.44 shows that there exist $n'_1, \ldots, n'_k$, with $n'_i \neq n'_j$ for $i \neq j$, and a prefix α' of a trace in $[\![\mathcal{S}(k+l)]\!]$ (resp. $[\![\mathcal{S}(n_0)]\!]$ if $n_0 > k + l$) such that $\alpha' \models \neg\phi(n'_1, \ldots, n'_k)$.

Since ϕ is a safety property, Corollary 3.17 implies that no matter how α' is prolonged to a full trace α'', the result will also not fulfill ϕ: $\alpha'' \models \neg\phi(n'_1, \ldots, n'_k)$, so $\mathcal{S}(k+l) \not\models \forall_d \phi(i_1, \ldots, i_k)$ (resp. $\mathcal{S}(n_0)$ if $n_0 > k + l$).

By Lemma 3.47, this is equivalent to $\mathcal{S}(k + l) \not\models \phi(0, \ldots, k-1)$ (resp. $\mathcal{S}(n_0)$), which contradicts our assumption! ■

The next example shows that there is no hope to improve the upper bound.

Example 4.47
We give an example for $k = l = 2$, that is easily extended to arbitrary values for k and l. Take a parameterized system with one array variable s ranging over $\{1, 2, 3, 4\}$, such that there is no transition leaving a state $s[i] = 1$ or $s[i] = 2$, and one transition $s[i] = 3 \rightarrow s[i] := 4$.

Choose the initial state condition $\theta \stackrel{\text{def}}{=} \exists_N i_1, i_2 . \forall_N i : s[i_1] = 1 \wedge s[i_2] = 2 \wedge s[i] \in \{1, 2, 3\}$, and the verification property $\forall_d \Box\neg(s[j_1] = 4 \wedge s[j_2] = 4)$.

Then, this property is valid for all instances $n < 4$, but invalid for instance $\mathcal{S}(4)$. Consequently, it is not possible to reduce the upper bound of the decision procedure. ♣

The decidability result holds also for general properties if we only investigate non-blocking systems.

Corollary 4.48
The verification problem for *non-blocking* Σ_1 systems with asynchronous control is decidable.

Proof: First we investigate the case $n_0 < k + l + 1$.

To prove $\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k)$ we check whether $\mathcal{S}(n) \models \phi(0, \ldots, k-1)$, for all n, $\max(k, n_0) \leq n \leq k+l+1$ (note the difference to Theorem 4.46:

there it suffices to use instances up to $\mathcal{S}(k+l)$). A negative answer is obviously correct since $n_0 \leq n$.

Assume now that the procedure gives the answer "valid" and there is a system instance $\mathcal{S}(n)$, $n > k+l+1$ (the other cases are similar to the proof of Theorem 4.46) such that $\mathcal{S}(n) \not\models \forall_d \phi(i_1, \ldots, i_k)$. Then, there exist an infinite trace $\alpha \in [\![\mathcal{S}(n)]\!]$ and some n_j, such that $\alpha \models \neg\phi(n_1, \ldots, n_k)$.

By repeated application of Corollary 4.45 we can reduce this counterexample to examples for smaller instances. Consequently, there exist an infinite trace $\beta \in [\![\mathcal{S}(k+l+1)]\!]$ and some (pairwisely distinct) $n'_j < k+l+1$, such that $\beta \models \neg\phi(n'_1, \ldots, n'_k)$. Hence, $\mathcal{S}(k+l+1) \not\models \forall_d \phi(i_1, \ldots, i_k)$. By Lemma 3.47, this is equivalent to $\mathcal{S}(k+l+1) \not\models \phi(0, \ldots, k-1)$, contradiction!

In case $n_0 \geq k+l+1$, we check $\mathcal{S}(n_0) \models \phi(0, \ldots, k-1)$. Similarly as in the previous case, counterexamples of larger instances may be reduced to traces of $\mathcal{S}(n_0)$ by repeated application of Corollary 4.45. ■

For the general verification problem, i.e., for all properties and the full system class, we can prove a similar result, but with a much higher bound.

Theorem 4.49
The verification problem for Σ_1 systems with asynchronous control is decidable.

Proof: This result is based on two observations. Let us assume that we have to prove

$$\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k) \ .$$

First, each infinite counterexample disproving ϕ can be found in one of the system instances $\mathcal{S}(k)$ up to $\mathcal{S}(k+l+1)$, due to Corollary 4.45.

Second, in a deadlocked trace, whenever *three or more* processes reach the same local state, then it is safe to remove one of them, getting a valid computation that also deadlocks, since each process is only able to quantify over all the *other* processes. So it makes no difference for enabledness of a transition, whether there is only one *other* process in a particular local state, or more than that.

Note that it is not possible to reduce to one process, since otherwise if process i is in local state s, and quantifies over *other* processes, then process i does not observe any other process in local state s anymore.

We now give the decision procedure for this verification problem. We additionally assume $n_0 < 2 \times |\Sigma_L| + k + l$ for this prove. This should be no real restriction in practice, since the bound is really high.

For every instance $\mathcal{S}(n)$, with $\max(k, n_0) \leq n \leq 2 \times |\Sigma_L| + k + l$, prove whether

$$\mathcal{S}(n) \models \phi(0, \ldots, k-1)$$

holds. If this is true for all instances, then the answer is "valid", otherwise answer "false".

To prove correctness of this decision procedure let us assume that the given answer is "valid" and there is some instance $\mathcal{S}(n)$ such that $\mathcal{S}(n) \not\models \forall_d \phi(i_1, \ldots, i_k)$. With the same argumentation as before we conclude $n > 2 \times |\Sigma_L| + k + l$, and there exist α and (pairwisely distinct) $n_1, \ldots, n_k$ such that $\alpha \models \neg\phi(n_1, \ldots, n_k)$.

If α is infinite, then by repeated application of Corollary 4.45, there exists already an infinite counterexample for the system instance $\mathcal{S}(2 \times |\Sigma_L| + k + l)$ which contradicts the assumption.

Now assume that α is finite. Let σ be the final state of α. Since α is a computation, $\alpha(0) \models \theta$. Consequently, there exist $j_1, \ldots, j_l$ such that $\alpha(0) \models \forall_N i.\phi_\theta(i, j_1, \ldots, j_l)$. Consider now

$$M \stackrel{\text{def}}{=} \{s \mid \exists j < 2 \times |\Sigma_L| + k + l : j \notin \{n_1, \ldots, n_k, j_1, \ldots, j_l\} \\ \wedge \pi_j(\sigma) = s\} \ .$$

Since $n > 2 \times |\Sigma_L| + k + l$, there must exist some $s \in M$ such that $\pi_j(\sigma) = s$ for *at least three* different j, even if we do not consider the indices $n_1, \ldots, n_k$ and $j_1, \ldots, j_l$. Choose such an index j.

Similarly as in Lemma 4.44, one can erase process j and remove stuttering steps afterwards. This results in a sequence β of $\mathcal{S}(n-1)$. By Lemma 4.43, every step is valid, and by Lemma 3.41, $\beta(0) \models \theta$. Consequently, β is at least a prefix of a $\mathcal{S}(n-1)$ run.

In fact, β is maximal, since for every process i, if i could make a step in the final state of β, then this would also be possible for this process in the final state of α, as explained in the beginning. The same is true for controller steps.

By repeating this construction, we get a counterexample of $\mathcal{S}(2 \times |\Sigma_L| + k + l)$, Lemma 3.47 states that this is a contradiction to our assumption. ■

Remark 4.50
For *weak* Σ_1 systems with asynchronous control, one can use the decision procedure of Theorem 4.49 with a lower bound. For these more restricted systems it is sufficient to check systems up to a bound of $|\Sigma_L| + k + l$ instead

of $2 \times |\Sigma_L| + k + l$, since the "environment" of each process includes the process itself. Hence, the very same construction, based on removing one of the processes in larger system instances, can be applied for this lower bound. ♣

4.2.2 Undecidability for Synchronous Control

In this section we show undecidability of the verification problem for Σ_1 systems with synchronous control. It is possible to reduce the halting problem for two counter machines to the verification problem for this class. The synchronous control can be used to model the state of the two counter machine by one global variable, and two parameterized arrays can be used to model the registers.

Although each instance of the system is finite, and can therefore only be used to store bounded counter values, each finite run of the two counter machine can be simulated by one instance, since in a finite run only bounded register values can occur. This suffices, since we are interested in finite computations of the two counter machine, namely the halting ones.

The universal quantification of Σ_1 systems can be used to check whether one or both counters are zero.

From the previous section it is clear that Σ_1 systems without control are decidable. Hence, we investigate models with weaker observability and try to find the boundary of decidability.

Theorem 4.51
The verification problem for Σ_1 systems with synchronous control is undecidable.

Proof: To prove this result we reduce the halting problem for 2CM to this decision problem. So let $M = (Q, q_0, R_0, R_1, T)$ be a deterministic 2CM. Our goal is to define a parameterized system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ which is able to simulate M.

Define $\mathcal{V} \stackrel{\text{def}}{=} \{q, x_0, x_1\}$, where x_0 and x_1 are arrays of type $\{0,1\}$ and $\text{type}(q) = Q$.

Intuitively, the number of entries with value 1 in each array x_j will simulate the value of register R_j, for $j = 0, 1$.

For each instruction $in = (p, b_0, b_1, k, D, p')$ of M we define one controller transition and one user transition by

$$\begin{aligned}\rho_{c,in} \stackrel{\text{def}}{=} \forall_N j : q = p \wedge \phi_2(b_0, b_1, j) \\ \rightarrow q := p'\end{aligned}$$

and

$$\begin{aligned}\rho_{u,in}(i) \stackrel{\text{def}}{=} \forall_N j : q = p \wedge \phi_1(k, D) \wedge \phi_2(b_0, b_1, i) \\ \wedge (j \neq i \Rightarrow \phi_2(b_0, b_1, j)) \\ \rightarrow \psi(k, D)\end{aligned}$$

where ϕ_1, ϕ_2, and ψ are defined by

$$\phi_1(k, D) \stackrel{\text{def}}{=} \begin{cases} x_k[i] = 0 & \text{if } D = inc \\ x_k[i] = 1 & \text{if } D = dec \end{cases}$$

$$\phi_2(b_0, b_1, v) \stackrel{\text{def}}{=} \begin{cases} x_0[v] = 0 \wedge x_1[v] = 0 & \text{if } b_0 \wedge b_1 \\ x_0[v] = 0 & \text{if } b_0 \wedge \neg b_1 \\ x_1[v] = 0 & \text{if } b_1 \wedge \neg b_0 \\ tt & \text{if } \neg b_0 \wedge \neg b_1 \end{cases}$$

and

$$\psi(k, D) \stackrel{\text{def}}{=} \begin{cases} x_k[i] := 0 & \text{if } D = dec \\ x_k[i] := 1 & \text{if } D = inc \\ skip & \text{if } D = nop \end{cases}$$

Sub-formula ϕ_1 ensures that the process taking the transition can indeed increase resp. decrease the counter which is modified by the instruction. The sub-formulae ϕ_2 makes sure that the transition can only be taken if the b_0 and b_1 conditions of the instruction are fulfilled.

Note that it is possible that one instance deadlocks, e.g., if all processes have set their $x_0[j]$ value to 1, and there is an instruction to simulate which increases counter R_0.

If $\mathcal{S}(n)$ makes a step, synchronously a controller transition and a user transition is taken. Since M is deterministic, at any time, only one such transition in each set can be enabled, namely the one derived from the same instruction. So, if a step is possible, it corresponds to an instruction of M.

With the constructed parameterized system $\mathcal{S}$, the 2CM M is not halting if and only if

$$\mathcal{S}(\geq 1) \models \Box(q \neq q_{stop}) \ .$$

Hence, the decision problem whether M is not halting, which is undecidable, is reduced to the verification problem of Σ_1 systems with synchronous control. Consequently, the latter problem must be undecidable as well. ■

Corollary 4.52
The verification problem for *weak* Σ_1 systems with synchronous control is undecidable.

Proof: Inspecting the previous proof one observes that all transitions used there correspond in fact to transitions of the weak Σ_1 system class, where $j \neq i$ is not used. ■

By the results of Section 4.2.1 we know the Σ_1 model without control is decidable. Hence, the boundary of decidability is between the full Σ_1 model with synchronous control and the class without control. Consequently, we are investigating classes with weaker observability as defined in Section 3.4.2.

Lemma 4.53
The verification problem for weak Σ_1 systems with non-reactive synchronous control is undecidable.

Proof: The main idea is that although the controller cannot refer to the user processes, it can be used to ensure that only exactly one process is able to simulate a 2CM instruction.

Let $M = (Q, q_0, R_0, R_1, T)$ be a 2CM. Let $\mathcal{V} \stackrel{\text{def}}{=} \{c, q, ph, x_0, x_1\}$, where ph, x_0, and x_1 are arrays of type $\{0,1\}$, q is an array of type Q, and type$(c) = \{g, y, r\}$. Intuitively, the only control variable c has values *green*, *yellow*, and *red*. It immediately changes state from *green* to *yellow*, allowing one user process to simulate an instruction. Then, the control can make arbitrary idle steps, allowing the user processes j to "guess" the right 2CM control state $q[j]$. Finally, the control moves to the *red* state, simultaneously with one "guessing" process. Then, control moves from *red* to *green*. The only process allowed to proceed simultaneously is the first one, this process checks whether all other processes guessed the right 2CM control state.

Hence, we define the following set of non-reactive controller transitions:

$$\rho_{g\to y} \stackrel{\text{def}}{=} c = g \to c := y$$
$$\rho_{y\to y} \stackrel{\text{def}}{=} c = y \to skip$$
$$\rho_{y\to r} \stackrel{\text{def}}{=} c = y \to c := r$$
$$\rho_{r\to g} \stackrel{\text{def}}{=} c = r \to c := g$$

Intuitively, the controller's "language" is $(gy^\star r)^\omega$.

For each instruction $in = (p, b_0, b_1, k, D, p')$ of M we define one user transition using ϕ_1, ϕ_2, and ψ as given in the proof of Theorem 4.51:

$$\begin{aligned}\rho_{in} \stackrel{\text{def}}{=} \forall_N j : q[i] = p \wedge c = g \wedge \phi_1(k, D) \wedge \phi_2(b_0, b_1, i)\\ \wedge (j \neq i \Rightarrow \phi_2(b_0, b_1, j))\\ \to \psi(k, D); ph[i] := 1\end{aligned}$$

The next user transitions are used to "guess" the 2CM control state of the process which simulated an instruction. For each $p' \in Q$ define:

$$\rho_{guess,p'} \stackrel{\text{def}}{=} c = y \wedge ph[i] = 0 \to q[i] := p'$$

The last user transition checks whether all guesses where right, deadlocking otherwise:

$$\rho_{u,r\to g} \stackrel{\text{def}}{=} \forall_N j : c = r \wedge ph[i] = 1 \wedge (j \neq i \Rightarrow q[j] = q[i]) \\ \to ph[i] := 0$$

The initial state predicate is given by

$$\theta \stackrel{\text{def}}{=} \forall i : c = g \wedge ph[i] = 0 \wedge q[i] = q_0 \wedge x_0[i] = 0 \wedge x_1[i] = 0 \ .$$

Assume that there is a finite computation r of M reaching q_{stop}. Let n be larger than the maximal value of any register in the computation. We show that $\mathcal{S}(n)$ has a trace reaching a corresponding state. The trace corresponds to the computation of M in the same way as in previous proofs: the number of processes with $x_k[i] = 1$ is the value of counter R_k.

For each step of M, where instruction in is taken, $\mathcal{S}(n)$ can make the following sequence of steps:

1. Simultaneously with $\rho_{g\to y}$, one process i takes transition ρ_{in} (there will be always one since n is larger than any register number in r). This process sets $ph[i] = 1$ and $q[i] = p'$.

2. Each of the other processes j but one takes transition $\rho_{guess,p'}$ simultaneously with the control transition $\rho_{y\to y}$, guessing the same 2CM control state $q[j] = p'$ as process i.

3. The last of the other processes guesses the same control state; simultaneously the controller takes transition $\rho_{y\to r}$.

4. Now process i proceeds with transition $\rho_{u,r\to g}$, and the controller takes $\rho_{r\to g}$. After this step, all processes have the same $q[i]$ value, namely p', $ph[i] = 0$ for all of them, and the number of processes with $x_k = 1$ is the value of register R_k.

Consequently, system instance $\mathcal{S}(n)$ can reach a state satisfying $ph[i] = 1 \wedge q[i] = q_{stop}$, so it violates the property

$$\mathcal{S}(\geq 1) \stackrel{\text{def}}{=} \forall\Box(c = g \Rightarrow q[i] \neq q_{stop}) \ .$$

Now let us assume there is some trace of some instance $\mathcal{S}(n)$ violating this verification property. Then, there exists a finite prefix of this trace ending in a state such that $c = g$ and for some index i, $q[i] = q_{stop}$.

We observe that $c = g$ can only be left, if one process i simultaneously takes a transitions simulating a 2CM instruction. This process i is then the

only one which can take a transition if $c = r$. Further steps of other processes and the controller does not change this. Hence, the controller can reach $c = g$ only if the process i takes simultaneously transition $\rho_{u,r\to g}$, but this transition checks whether the system is in a "proper" state, which corresponds to a 2CM configuration. Hence, the prefix corresponds to a 2CM computation which also reaches q_{stop}.

By checking the verification property given above, the halting problem for 2CM is reduced to the verification of Σ_1 systems with weak synchronous control. Hence, the verification problem for this class is undecidable.

It is easy to see that all transitions defined in this construction correspond to transitions of the weak system class. ■

Lemma 4.54
The verification problem for weak Σ_1 systems with synchronous independent control is decidable for control safety properties.

Proof: In this case, one only needs to investigate the controller on its own, computing the minimal paths to each reachable control state. Verification reduces to deciding whether the user processes of some instance are able to make k steps such that the controller can reach a control state in k steps. This is decidable here since it is sufficient to consider system instances up to size k. ■

4.3 Undecidability for $\Sigma_0 \cup \Sigma_1$ Systems

For a system in the class $\Sigma_0 \cup \Sigma_1$, each transition is either in the class Σ_0 or in the class Σ_1, so it is not allowed to use both existential and universal quantification in one transition guard, but both transition types may be used.

In this section we show that even if we only allow array variables, the verification problem for such a restricted class is undecidable. To prove this result, we proceed as follows.

First, we show that the verification problem for $\Sigma_0 \cup \Sigma_1$ systems with asynchronous control is undecidable. This is done by reducing the halting problem for 2CM to a verification problem of this class. Therefore, we encode an arbitrary 2CM into a parameterized system $\mathcal{S}$ in this class, such that every instance $\mathcal{S}(n)$ fulfills a particular simple property ϕ, if and only if the original 2CM is halting. So if we could decide $\mathcal{S} \models \phi$, we could decide the halting problem for the 2CM, which is known to be undecidable.

In a second step, we show that even if we restrict to array variables only, the verification problem is undecidable. This is done by showing that each $\Sigma_0 \cup \Sigma_1$ system with asynchronous control can be simulated (in some sense) by a $\Sigma_0 \cup \Sigma_1$ system without control.

Theorem 4.55
There is no decision procedure that decides for arbitrary parameterized systems in the class $\Sigma_0 \cup \Sigma_1$ with asynchronous control, whether the system fulfills a given property.

Proof: We will reduce the halting problem for 2CM to this decision problem. So let us assume we have a 2CM $M = (Q, q_0, R_0, R_1, T)$. Now we define a parameterized system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ which is able to simulate M.

First define $\mathcal{V} \stackrel{\text{def}}{=} \{q, ph, qc, x_0, x_1\}$, where x_0 and x_1 are arrays of type $\{0, 1\}$, type$(q) = Q$, qc is an array of type Q, and ph is an array of type $\{0, 1\}$.

The idea is that the number of entries with value 1 in array x_j will be the value of register R_j, for $j = 0, 1$. The variable ph will be used to signalize the other processes that process i is executing an instruction by setting $ph[i]$ to 1. Only one process will be able to do so. Variable qc will be used to store the target state. It is used to decouple the register value changes (which is done by changing array values) with the modifications of the global variable q, thus, leading to a system with asynchronous control.

An M instruction will be simulated by a number of both control and user transitions for one user process i. It starts with setting $ph[i]$ to 1, changing the register values, storing the target state in the $qc[i]$ array, and setting

$ph[i]$ to 1. In the second step, q is assigned the correct target state (if this is necessary). The simulation of the instruction ends with setting $ph[i]$ back to 0.

To simulate the execution of an instruction $in = (p, b_0, b_1, k, D, p')$ we have to define three transitions. The first one simulates the arithmetic part of the instruction and is a user transition:

$$\begin{aligned}\rho_{in}(i) \stackrel{\text{def}}{=} \forall_N j : ph[i] = 0 \wedge q = p \wedge \phi_1(k, D) \wedge \phi_2(b_0, b_1, i) \\ \wedge (j \neq i \Rightarrow ph[j] = 0 \wedge \phi_2(b_0, b_1, j)) \\ \rightarrow \psi(k, D); ph[i] := 1; qc[i] := p'\end{aligned}$$

where ϕ_1, ϕ_2, and ψ are defined by

$$\phi_1(k, D) \stackrel{\text{def}}{=} \begin{cases} x_k[i] = 0 & \text{if } D = inc \\ x_k[i] = 1 & \text{if } D = dec \end{cases}$$

$$\phi_2(b_0, b_1, v) \stackrel{\text{def}}{=} \begin{cases} x_0[v] = 0 \wedge x_1[v] = 0 & \text{if } b_0 \wedge b_1 \\ x_0[v] = 0 & \text{if } b_0 \wedge \neg b_1 \\ x_1[v] = 0 & \text{if } b_1 \wedge \neg b_0 \\ tt & \text{if } \neg b_0 \wedge \neg b_1 \end{cases}$$

and

$$\psi(k, D) \stackrel{\text{def}}{=} \begin{cases} x_k[i] := 0 & \text{if } D = dec \\ x_k[i] := 1 & \text{if } D = inc \\ skip & \text{if } D = nop \end{cases}$$

The second transition is a controller transition, which modifies the state (variable q). Actually, these transitions does not depend on a particular instruction, so we define them independently for each state $p' \in Q$

$$\rho_{p'} \stackrel{\text{def}}{=} \exists_N j : ph[j] = 1 \wedge qc[j] = p' \wedge q \neq p' \rightarrow q := p' \ .$$

The third transition is used to deactivate the process when the new state is written. It is again a user transition.

$$\rho_{fin}(i) \stackrel{\text{def}}{=} ph[i] = 1 \wedge qc[i] = q \rightarrow ph[i] := 0 \ .$$

As initial state condition we choose

$$\theta \stackrel{\text{def}}{=} \forall_N i : x_0[i] = 0 \wedge x_1[i] = 0 \wedge ph[i] = 0 \wedge q = q_0 \ .$$

Then, for each finite computation prefix r of M, there exists a corresponding sequence of some instance $\mathcal{S}(n)$, where each instruction taken is

simulated by (at least two of) these three transitions in the given order. A system instance $\mathcal{S}(n)$ with n greater than the maximal register value appearing in r suffices. Consequently, if M can reach a particular configuration, there exists a system instance reaching a corresponding state.

Note that at any time, at most one process i of $\mathcal{S}(n)$ can have $ph[i] = 1$, since they all start with value 0, and there is only one set of transitions changing this value, but if one process does so, this set of transitions is immediately disabled for the other processes.

On the other hand, if there exists a finite prefix r of run of $\mathcal{S}(n)$, it must be a sequence of transitions in the order given above, because initially, there are only transitions from the first set enabled, afterwards only one of the other, and if such a transition is taken, it is disabled afterwards. After $ph[i]$ is set back to 0, this cycle starts anew. So, such a transition sequence is a simulation of a unique instruction of M. Consequently, there exists a corresponding computation prefix of M.

In case M would reach a value of l in one register, then for each instance $\mathcal{S}(n)$ with $n < l$, this value cannot be represented. Hence, such an instance is in a deadlock state, unless also other prolongations of the trace are possible with lower register values (note that we do not require M to be deterministic here).

Now assume that the verification problem is decidable for the given class. Then, given a 2CM M, construct $\mathcal{S}$ as given above. This is an effective construction. We can then use the decision procedure for $\mathcal{S}$ to decide whether

$$\forall n \geq 1 : \mathcal{S}(n) \models \Box(q \neq q_{stop}) \ .$$

If this property is fulfilled, this implies for the 2CM M that M never reaches the halting state. Otherwise, M can reach the halting state. Therefore, we have a decision procedure for the 2CM halting problem, which is known to be undecidable. Contradiction! ■

Corollary 4.56
The verification problem for weak $\Sigma_0 \cup \Sigma_1$ systems with asynchronous control is not decidable.

Proof: All transitions in the proof of Theorem 4.55 correspond to transitions of the weak system class. ■

We now prove that systems with controller can be simulated by controller-less systems in the sense of safety properties: every trace of the system with controller has a counterpart in the other one, although there may be some traces of the controller-less system that are only prefixes of traces of the other system, followed by infinitely many "internal steps".

Theorem 4.57
For each (weak) $\Sigma_0 \cup \Sigma_1$ system $\mathcal{S}$ with asynchronous control, there exists a (weak) $\Sigma_0 \cup \Sigma_1$ system $\tilde{\mathcal{S}}$ without control such that

$$\mathrm{SC}([\![\mathcal{S}(n)]\!]) = \mathrm{SC}(f([\![\tilde{\mathcal{S}}(n)]\!])) \ ,$$

for all $n \geq 1$, where f is (the extension to sequences of) a function mapping states of an instance $\tilde{\mathcal{S}}(n)$ to $\mathcal{S}(n)$ states. Moreover, given $\mathcal{S}$, the system $\tilde{\mathcal{S}}$ is effectively computable.

Proof: Let us assume we have a parameterized system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ in the class $\Sigma_0 \cup \Sigma_1$ with asynchronous control, and assume that $\mathcal{V}_Y \neq \emptyset$ is the set of normal variables in $\mathcal{V}$ (otherwise there is nothing to do). Each such variable we will be replaced by a new array variable. Intuitively, every process has its own copy of the normal variables. Moreover, we introduce a new array variable ph of type $\{0, 1, 2, 3, 4\}$. This variable will be the state of the local copy, with the following intended meaning:

$ph = 0$**:** local copy is not valid,

$ph = 1$**:** local copy is valid,

$ph = 2$**:** local copy is valid; signal that this process wants to simulate a transition of $\mathcal{S}$,

$ph = 3$**:** this process has simulated a transition, hence, the other processes might have invalid copies now and should invalidate them,

$ph = 4$**:** this process has simulated a transition, and every other process has invalidated its local copy.

Now we will define the new system $\tilde{\mathcal{S}}$. The variables of $\tilde{\mathcal{S}}$ are $\mathcal{V} \cup \{ph\}$, where the normal variables in $\mathcal{V}$ are now interpreted as array variables.

For each controller transition given as a predicate

$$\rho_\tau = Q_N \bar{\jmath} : g(\bar{\jmath}) \rightarrow \bar{y} := \bar{c} \ ,$$

with $Q \in \{\forall, \exists\}$, $\bar{y}$ a vector of normal variables, define the user transition

$$\begin{aligned} \tilde{\rho}_\tau(i) \stackrel{\text{def}}{=} Q_N \bar{\jmath} : & \, ph[i] = 2 \wedge g(\bar{\jmath})[y[i]/y \mid y \in \mathcal{V}_Y] \\ & \rightarrow ph[i] := 3; \bar{y}[i] := \bar{c} \ . \end{aligned}$$

For each user transition given as a predicate

$$\rho_\tau(i) = Q_N \bar{\jmath} : g(i, \bar{\jmath}) \rightarrow \bar{x}[i] := \bar{c} \ ,$$

with $Q \in \{\forall, \exists\}$, $\bar{x}$ a vector of array variables of $\mathcal{S}$, define the transition

$$\tilde{\rho}_\tau(i) \stackrel{\text{def}}{=} Q_N \bar{\jmath} : ph[i] = 2 \wedge g(i,\bar{\jmath})[y[i]/y \mid y \in \mathcal{V}_Y] \\ \rightarrow ph[i] := 3; \bar{x}[i] := \bar{c} \ .$$

Both constructions for user and controller transitions define (weak) Σ_0 transitions, if ρ_τ is (weak) Σ_0, and (weak) Σ_1 otherwise.

Moreover, we need some transitions which take care of copying the correct local copy of the normal variables to another index. For each value of ph, we have at least one transition. The subscripts give the source and destination value of the $ph[i]$ variable.

The first transition is just for signaling: A process which takes it gains *exclusive access* on transitions simulating $\mathcal{S}$ transitions.

$$\rho_{1\rightarrow 2}(i) \stackrel{\text{def}}{=} \forall_N j : ph[j] = 1 \rightarrow ph[i] := 2 \ .$$

Next, the same process has to take a simulating instruction. After that one, potentially the local copy of the global state has changed, so it must to copied to all other indices (in fact, it would be enough to do this only for controller transitions of $\mathcal{S}$). This is done using the following transitions.

First, the other processes invalidate their local copy by taking

$$\rho_{1\rightarrow 0}(i) \stackrel{\text{def}}{=} \exists_N j : ph[i] = 1 \wedge ph[j] = 3 \rightarrow ph[i] := 0 \ .$$

If all local copies are invalidated, the first process which simulated a transition becomes active and signalizes that its local copy of the global state can be copied.

$$\rho_{3\rightarrow 4}(i) \stackrel{\text{def}}{=} \forall_N j : (ph[j] = 0 \vee ph[j] = 3) \wedge ph[i] = 3 \rightarrow ph[i] := 4 \ .$$

Now the valid value for the $\mathcal{V}_Y$ variables, denoted as $\bar{y}$, is copied to the local copy of index i. For each vector of possible values $\bar{v}$ for the variable vector $\bar{y}$ we define a transition

$$\rho_{\bar{v},0\rightarrow 1}(i) \stackrel{\text{def}}{=} \exists_N j : ph[i] = 0 \wedge ph[j] = 4 \wedge \bar{y}[j] = \bar{v} \\ \rightarrow ph[i] := 1; \bar{y}[i] := \bar{v} \ .$$

Once the valid values have reached every index, the simulating process changes state:

$$\rho_{4\rightarrow 1}(i) \stackrel{\text{def}}{=} \forall_N j : (ph[j] = 1 \vee ph[j] = 4) \wedge ph[i] = 4 \rightarrow ph[i] := 1$$

The transition relation is then given by the predicate:

$$\tilde{\rho} \stackrel{\text{def}}{=} \exists_N i : \rho_{1\rightarrow 2} \vee \bigvee_{\text{all transitions } \tau} \tilde{\rho}_\tau \\ \vee \rho_{1\rightarrow 0} \vee \rho_{3\rightarrow 4} \\ \bigvee_{\text{for all possible value vectors } \bar{v}} \rho_{\bar{v},0\rightarrow 1} \\ \vee \rho_{4\rightarrow 1}$$

The initial state predicate $\tilde{\theta}$ is defined as follows:

$$\tilde{\theta} \stackrel{\text{def}}{=} \exists_N i_1 . \forall_N i : \theta[y[i_1]/y \mid y \in \mathcal{V}_Y] \\ \wedge\, ph[i] = 1 \wedge \textstyle\bigwedge_{y \in \mathcal{V}_Y} y[i_1] = y[i] \ .$$

We now give some observations about the construction and invariance properties which are fulfilled by each instance of $\tilde{\mathcal{S}}$:

- There are no normal variables in the constructed transitions, thus, there is no control.
- There is always one process j with $ph[j] > 0$. Initially, all processes have that value. Transitions changing the value to 0 ensure that there remains at least one process having a value greater 0.
- There is never more than one process j with $ph[j] \geq 2$. This is implied by definition of transition $\rho_{1\to 2}$, which is the only transition setting $ph[j]$ from a value < 2 to a value ≥ 2.
- If $ph[j] = 4$, then $\bar{y}[j] = \bar{y}[k]$ holds for all k with $ph[k] = 1$. This holds, since the only way to set $ph[j]$ to 4 is transition $\rho_{3\to 4}$, and this transition can only be taken if every other index k has $ph[k] = 0$. Transition $\rho_{0\to 1}$ ensures $\bar{y}[j] = \bar{y}[k]$ for each index k reaching $ph[k] = 1$.
- If $ph[j] = 1$ for all indices j, then $\bar{y}[k] = \bar{y}[j]$ for all k, j.
- If $\mathcal{S}$ is a weak system, also $\tilde{\mathcal{S}}$ is weak.

For each run of $\mathcal{S}(n)$, there is a simulating run of $\tilde{\mathcal{S}}(n)$. For a transition τ taken in the run of $\mathcal{S}(n)$ by the controller or some process j, one simply has to simulate this transition by one index and copies the global state afterwards to all other indices.

On the other hand, if there is a run of $\tilde{\mathcal{S}}(n)$ reaching a particular global state $\tilde{\sigma}$, there exists a corresponding (prefix of a) run of $\mathcal{S}(n)$. Take simply the occurrences of $\tilde{\rho}_\tau$ steps in that run, ignoring all other steps. Then, the sequence of the corresponding ρ_τ steps is a (prefix of a) run of $\mathcal{S}(n)$, reaching a state σ which corresponds to $\tilde{\sigma}$ in such a way that all "valid" local copies of the global variables in $\tilde{\sigma}$ have the same values as the global variables in σ.

For each $n \geq 1$ define $f_n : \tilde{\Sigma}_n \to \Sigma_n$ by

$$f_n(\tilde{\sigma})(v) \stackrel{\text{def}}{=} \begin{cases} \tilde{\sigma}(v) & \text{if } v \in \mathcal{V}_X \\ \tilde{\sigma}(v)(j) & \text{if } v \in \mathcal{V}_Y, \text{ for some } j \text{ such that} \\ & \tilde{\sigma}(ph)(j) \text{ is maximal} \end{cases}$$

Note that this is only a partial function on $\tilde{\Sigma}_n$, but it covers the whole set of reachable states as argued before. Define $f \stackrel{\text{def}}{=} \bigcup_{n\geq 1} f_n$. If one evaluates f on a trace of $\tilde{\mathcal{S}}(n)$, this results in a stuttering (of a prefix) of a $\mathcal{S}(n)$ run, since the "internal" steps copying the global state copies become stutter steps when applying f, only the transitions derived from $\mathcal{S}$ transitions are observable.

The theorem is then a consequence of the correspondence of $\mathcal{S}$ runs and $\tilde{\mathcal{S}}$ runs as explained above. ■

Note that in the previous construction, there are infinite runs of $\tilde{\mathcal{S}}$ that do not have an infinite counterpart of $\mathcal{S}$, e.g., runs where one process keeps on copying the $\mathcal{V}_Y$ variables and invalidating its copy afterwards. The corresponding run of $\mathcal{S}$ is only consisting of the initial state.

This is the reason why we have to restrict to safety properties only, for which only the set of reachable states is important. If one would add a fairness constraint to $\tilde{\mathcal{S}}$ ruling out these infinite traces, the resulting system would be bisimilar to $\mathcal{S}$, hence allowing also to reason about liveness properties.

Corollary 4.58
There is no decision procedure that decides for arbitrary parameterized systems in the class of weak $\Sigma_0 \cup \Sigma_1$ systems without control, whether the system fulfills a given property.

Proof: Assume that there exists a decision procedure for weak $\Sigma_0 \cup \Sigma_1$ systems without control. Then, by the construction given in Theorem 4.57, an arbitrary weak $\Sigma_0 \cup \Sigma_1$ system $\mathcal{S}$ with asynchronous control can efficiently be translated to a weak $\Sigma_0 \cup \Sigma_1$ system $\tilde{\mathcal{S}}$ without control such that

$$\mathrm{SC}([\![\mathcal{S}(n)]\!]) = \mathrm{SC}(f([\![\tilde{\mathcal{S}}(n)]\!])) \ .$$

Assume now that ϕ is a safety property (involving controller variables only, this suffices due to the proof of Theorem 4.55). By the equivalence of the stutter closures of both systems, $\mathcal{S}(n)$ fulfills a safety property if and only if every $\alpha \in f([\![\tilde{\mathcal{S}}(n)]\!])$ do.

The latter is equivalent to $f(\beta) \models \phi$, for every $\beta \in [\![\tilde{\mathcal{S}}(n)]\!]$.

With the observations made in the proof of Theorem 4.55 and the definition of f, this can be equivalently transformed into $\beta \models \tilde{\phi}$ for

$$\tilde{\phi} \stackrel{\text{def}}{=} \forall i.ph[i] > 0 \Rightarrow \phi[y[i]/i \mid y \in \mathcal{V}_Y] \ ,$$

for every $\beta \in [\![\tilde{\mathcal{S}}(n)]\!]$, since there is always such a process i, and all processes i with $ph[i] > 0$ agree on their $\bar{y}$ values.

So, by checking $\tilde{\mathcal{S}}(\geq n_0) \models \tilde{\phi}$, we can solve the verification problem for weak $\Sigma_0 \cup \Sigma_1$ systems with asynchronous control and controller safety

properties, which we know by Corollary 4.56 is undecidable. Contradiction! ■

Since $\Sigma_0 \cup \Sigma_1$ without controller is the weakest $\Sigma_0 \cup \Sigma_1$ class, obviously, the verification problems for all $\Sigma_0 \cup \Sigma_1$ classes are not decidable. This includes, of course, also systems with synchronous control!

Corollary 4.59
The verification problem for (weak) $\Sigma_0 \cup \Sigma_1$ with synchronous control is undecidable.

Proof: Every $\Sigma_0 \cup \Sigma_1$ system without control is a $\Sigma_0 \cup \Sigma_1$ with synchronous control. ■

4.4 Decidability for Π_0 Systems

In this section we show that the verification problem for Π_0 systems with both synchronous and asynchronous control is decidable. The proof follows the idea presented in Section 4.2.1.

In this system class, only universal quantification over other processes is used, thus, removing one process does not invalidate transition guards. In contrast to the Σ_1 systems, both the asynchronous and the synchronous control is unproblematic.

So let $\mathcal{S}$ be a parameterized system in the class Π_0 with initial state predicate $\theta = \exists_N i_1, \ldots, i_l . \forall_N i : \phi_\theta(i, i_1, \ldots, i_l)$, and transition relation predicate ρ built up by transitions T_C and T_U.

Lemma 4.60
Let $\phi(i_1, \ldots i_k)$ be an LTL formula, and let l be the number of existential quantifications in θ. For all $n \geq k + l$ and $n_1, \ldots, n_k < n + 1$, with $n_i \neq n_j$ for $i \neq j$, if

- α is a prefix of a trace in $[\![\mathcal{S}(n+1)]\!]$, and
- $\alpha \models \phi(n_1, \ldots, n_k)$,

then there exist a prefix α' of a trace in $[\![\mathcal{S}(n)]\!]$, and $n'_1, \ldots, n'_k < n$, with $n'_i \neq n'_j$ for $i \neq j$, such that

$$\alpha' \models \phi(n'_1, \ldots, n'_k) \ .$$

Proof: The idea is to pick out the processes needed to satisfy θ, $j_1, \ldots j_l$, and to choose an index $j \notin \{j_1, \ldots, j_l, n_1, \ldots, n_k\}$. Such an index always exists for our choice of n.

Then, projecting α on all the other processes $\neq j$, one has a trace of system $\mathcal{S}(n)$ (if $j < n$, one has to swap the rôle of j and index n before, otherwise there would be a "hole" in the index set).

For every transition taken, the set of local states is a subset of that in α, so the transition remains enabled. The argumentation is similar to that of Lemma 4.43. This construction is the same for systems with synchronous as well as with asynchronous control.

Note that one can not be sure that the result is a *maximal* trace, since if α is finite, the set of local states in the final state could be smaller than in the final state of α, so there may be more transitions enabled. ■

This lemma shows how to verify safety properties for Π_0 systems: by checking system instances up to the bound $k + l$.

Theorem 4.61
The verification problem for Π_0 systems with both synchronous and asynchronous control is decidable for safety properties.

Proof: Follows from Lemma 4.60 following the same lines as in the proof of Theorem 4.46. ■

The decidability result holds also for general properties if we only investigate non-blocking systems. In contrast to Σ_1 systems, the bound is the same as for verification of safety properties.

Corollary 4.62
The verification problem for *non-blocking* Π_0 systems with both synchronous or asynchronous control is decidable.

Proof: Note that Lemma 4.60 shows the existence of an infinite trace of α is infinite. Hence, if the system is non-blocking, the resulting trace by the Lemma is always maximal. ■

Similarly as for Σ_1 systems, the bound is higher for general properties.

Theorem 4.63
The verification problem for Π_0 systems with both synchronous and asynchronous control is decidable.

Proof: The same idea as for Σ_1 systems can be applied. The observation is that each deadlock trace can be reduced to a trace of $\mathcal{S}(2 \times |\Sigma_L| + k + l)$, where k and l are chosen as before.

Assume we are interested in proving

$$\mathcal{S}(\geq n_0) \models \forall_d \, \phi(i_1, \ldots, i_k) \ .$$

Each infinite counterexample can already be found for one of the systems $\mathcal{S}(k)$ up to $\mathcal{S}(k+l)$, similar as in Corollary 4.62.

Now assume α is a deadlocked trace of some instance $\mathcal{S}(n)$ and α is a counterexample for the property. As usual, we can restrict the indices used for ϕ to $0, \ldots, k-1$. Choose indices $j_1, \ldots j_l$ used to fulfill θ.

Then, $n > 2 \times |\Sigma_L| + k + l$ must hold, so there exists $j \notin \{0, \ldots, k-1, j_1, \ldots, j_l\}$ such that the local state of index j appears at least 3 times in the final state of α.

Projecting away this index j (and swapping it first if $j \neq n$) leads to a trace of $\mathcal{S}(n-1)$ that also deadlocks and that is also a counterexample of ϕ. Repeating this construction leads to a counterexample of $\mathcal{S}(2 \times |\Sigma_L| + k + l)$, contradiction. ■

Remark 4.64
The main difference to asynchronous Σ_1 systems is that here, one can simply remove one index (resp. user process), and the result is also a real trace. For Σ_1 systems with asynchronous control, one has to remove stuttering to leave out the steps of the removed process.

For Σ_1 systems with synchronous control, removing a process is not possible in that way, since the controller always makes steps *synchronously* with one user process, so if such a process is projected away, some other process would have to fill that place. Our results show that this is in general not possible, since the verification problem for Σ_1 systems with synchronous control is undecidable. ♣

4.5 Decidability for Π_1 Systems

We show that the verification problem for Π_1 systems with both asynchronous and synchronous control is decidable for safety properties. This is done by showing that this problem can be reduced to the verification problem for *weak* Π_1 systems, a class that is known to be decidable.

Assume that $\mathcal{S}$ is a system in the class Π_1 with asynchronous or synchronous control. Let $\mathcal{S}'$ be the "weak counterpart" of $\mathcal{S}$, where each $i \neq j$ is substituted by tt.

We first observe that weak systems have at least the same behavior than their counterparts. This is only true for infinite traces, finite traces of $\mathcal{S}$ may be prolonged in some cases, such that some finite $\mathcal{S}$ traces are no $\mathcal{S}'$ traces.

Lemma 4.65
For each $n \geq 1$ and $\alpha \in [\![\mathcal{S}(n)]\!]$, α is a prefix of a $\mathcal{S}'(n)$ trace.

Proof: Let us first assume that $\mathcal{S}$ has asynchronous control. For each non-controller step, there exists a transition $\tau \in T_U$ for each process $i < n$ such that τ is taken by index i in that step. Hence, the guard of τ evaluates to true, and the assignment transforms the local pre-state of the step into the right local post-state.

Transition τ has a guard of the form $g(i) \equiv \phi(i) \wedge \exists_N \bar{j} : \bar{j} \neq i \wedge \psi(i, \bar{j})$. Then, the guard of the corresponding $\mathcal{S}'$ transition has the form $g'(i) \equiv \phi(i) \wedge \exists_N \bar{j} : \psi(i, \bar{j})$. Hence, in the same pre-state, the guard of this transition also evaluates to true, and leads to the same local post-state (for all indices).

For each controller step, one can use a similar argumentation to show that the corresponding weak transition can be taken if $\mathcal{S}$ takes a control transition τ in that step.

If $\mathcal{S}$ has synchronous control, for each step there exists a control transition and user transitions for each $i < n$ that are taken in this step. Again, all the weak counterparts are also enabled then.

Consequently, $\mathcal{S}'(n)$ can always take the same transitions in each step. For deadlocked traces, the trace is not necessarily a maximal trace for $\mathcal{S}'(n)$. ■

The next Lemma shows that at least for safety properties, the weak systems are equivalent to the "strong" versions, since each counterexample of a weak system may be transformed into one of the strong system.

Lemma 4.66
Let α' be a finite prefix of a trace in $[\![\mathcal{S}'(n')]\!]$ such that $\alpha' \models \phi(n_1, \ldots, n_k)$. Then, there exist $n \geq n'$ and a prefix α of trace in $[\![\mathcal{S}(n)]\!]$ such that $\alpha \models \phi(n_1, \ldots, n_k)$.

Proof: Let α' be finite. We observe that we can always add a new process that behaves as some existing process to the sequence, getting a "good" sequence.

The idea is to duplicate every process, so we introduce a new process for each of the n' processes in α' that behaves exactly as the original one. This guarantees that in each position of α', each local state appears at least twice. Then, a $\mathcal{S}'$ transition is enabled for some index i in some state of α' iff the corresponding $\mathcal{S}$ transition is enabled in the corresponding α state.

So choose α such that

- $\pi_{G,0,\ldots,n'-1}(\alpha) = \pi_{G,0,\ldots,n'-1}(\alpha')$, and
- $\pi_{n',\ldots,2n'-1}(\alpha) = \pi_{0,\ldots,n'-1}(\alpha')$.

This is a valid prefix of a trace of $\mathcal{S}'(2n')$, since adding a local state to the global state cannot disable transitions that are previously enabled (since we only have existential transition guards), and the duplicated process can always take the same transition as the original.

Moreover, for each $l < |\alpha|$, each local state of Localstates$(\alpha(l))$ appears at least twice. Therefore, α is also a valid prefix of a trace of $\mathcal{S}(2n')$. Intuitively, the "environment" of each index i (all the local state of other indices) is the full set of local states Localstates$(\alpha'(l))$. Thus, each transition that may be taken by a process i in α' may also be taken by process i and $n'+i$ in α. ■

Using the previous results, we can reduce the verification problem for full Π_1 systems to that of the weak class.

Theorem 4.67
The verification problem for Π_1 systems (with synchronous or asynchronous control) is decidable for safety properties.

Proof: Assume that we want to prove

$$\mathcal{S}(\geq n_0) \models \forall_d \phi(i_1, \ldots, i_k) \ ,$$

for some safety property ϕ.

First construct the weak counterpart $\mathcal{S}'$ of $\mathcal{S}$, and then decide whether $\mathcal{S}'$ fulfills this property, using the procedure for weak synchronous systems given in Section 4.7 and return the same result.

Assume that the answer is "valid", but there exists some counterexample for ϕ for system instance $\mathcal{S}(n)$. Using Lemma 3.47, we conclude there exist a counterexample $\alpha \not\models \phi(0, \ldots, k-1)$.

Lemma 4.65 shows that α is also counterexample for $\mathcal{S}'(n)$, since ϕ is a safety property and therefore it is not important whether α is maximal or not, see Corollary 3.17. Contradiction!

Now let us assume that the answer is "false". Then there exists some trace $\alpha' \in [\![\mathcal{S}'(n)]\!]$ such that α' is a counterexample for ϕ, so $\alpha' \not\models \phi(0, \ldots, k-1)$. Since ϕ is a safety property, there exists some finite "bad" prefix $\bar{\alpha}'$ of α', see Lemma 3.16.

Lemma 4.66 shows that there exists a corresponding prefix α of a trace of $\mathcal{S}(2n)$ such that $\alpha \not\models \phi(0, \ldots, k-1)$. Consequently, $\mathcal{S}$ does not fulfill the property, so the answer is correct. ■

Remark 4.68
In these constructions, the choice of synchronous or asynchronous control has not much influence. One can observe that for adding a new process to a computation, all others may remain untouched, since the user processes make steps synchronously. So one is not required to interleave some steps. Moreover, by adding a new process, controller steps remain always enabled if they were before, so one can simply ignore these steps in the construction. ♣

4.6 Undecidability for $\Pi_0 \cup \Pi_1$ Systems

In this section we show that the verification problem for the "full" synchronous system class $\Pi_0 \cup \Pi_1$ is undecidable, even for the restricted class without control. For the simpler *weak* system class without quantification over "other" processes (we forbid comparisons like "$j \neq i$"), the verification problem is decidable. This result was shown in [EN96] for similar systems; this is explained in detail in Section 4.7.

The main idea is that the $j \neq i$ comparison can be used to decouple one single process from the others, and letting this one simulate a two counter instruction in an asynchronous way as in Section 4.2 and 4.3.

We use $\phi_1(k, D)$, $\phi_2(b_0, b_1, v)$, and $\psi(k, D)$ defined as in the proof of Theorem 4.51. $\phi_1(k, D)$ expresses that the process fulfilling this predicate is able to increment/decrement counter k as given by D, $\phi_2(b_0, b_1, v)$ can be used to express that the corresponding counters are zero, and $\psi(k, D)$ is the modification of the x_0 and x_1 variables according to command D and counter R_k.

Theorem 4.69
The verification problem for $\Pi_0 \cup \Pi_1$ systems with asynchronous control is undecidable.

Proof: Let $M = (Q, q_0, R_0, R_1, T)$ be a 2CM, and $\mathcal{V} = \{q, qc, l, try\}$ with $\text{type}(q) = Q$, qc is an array over Q, l is an array over $\{0, 1, \ldots, 5\}$ (we call $l[i]$ also the location of process i), x_0, x_1, and try are arrays over $\{0, 1\}$. Intuitively, the number of entries 1 in x_k corresponds to the value of counter k, as in the previous constructions.

Let $in = (p, b_0, b_1, k, D, p')$ be an instruction of M.

The following user transitions are the "normal behavior", these processes run endlessly in a loop with $l[i] = 0, 1, 2, 3$.

$$\begin{aligned}
\rho_{0\to1,try} &\stackrel{\text{def}}{=} l[i] = 0 \to l[i] := 1; try[i] := 1 \\
\rho_{0\to1,\neg try} &\stackrel{\text{def}}{=} l[i] = 0 \to l[i] := 1; try[i] := 0 \\
\rho_{1\to2} &\stackrel{\text{def}}{=} l[i] = 1 \to l[i] := 2 \\
\rho_{2\to3} &\stackrel{\text{def}}{=} l[i] = 2 \to l[i] := 3 \\
\rho_{3\to0} &\stackrel{\text{def}}{=} l[i] = 3 \to l[i] := 0
\end{aligned}$$

If one process realizes it is the only one who is trying ($try[i] = 1$), it may veer out and simulate an instruction if this is possible. Otherwise, the com-

putation deadlocks.

$$\begin{aligned}
\rho_{1\to 4} &\stackrel{\text{def}}{=} \forall_N j : l[i] = 1 \wedge try[i] = 1 \wedge (j \neq i \Rightarrow try[j] = 0) \\
&\quad \to l[i] := 4 \\
\rho_{4\to 5,in} &\stackrel{\text{def}}{=} \forall_N j : q = p \wedge l[i] = 3 \wedge \phi_1(k, D) \wedge \phi_2(b_0, b_1, i) \\
&\quad \wedge (j \neq i \Rightarrow \phi_2(b_0, b_1, i)) \\
&\quad \to \psi(k, D); l[i] := 5; qc[i] := p' \\
\rho_{5\to 0} &\stackrel{\text{def}}{=} l[i] = 5 \wedge q = qc[i] \to l[i] := 0
\end{aligned}$$

If the 2CM state is changed by the simulated instruction, the controller has to make the corresponding adjustment between the user process step $\rho_{4\to 5,in}$ and $\rho_{5\to 0}$, by one of the following controller transitions for $p' \in Q$:

$$\rho_{c,p'} \stackrel{\text{def}}{=} \exists_N j : q \neq p' \wedge l[j] = 5 \wedge qc[j] = p' \to q := p'$$

As initial state condition we choose

$$\theta \stackrel{\text{def}}{=} \forall i. l[i] = 0 \wedge try[i] = 0 \wedge q = q_0 \ .$$

The transition relation ρ is built up by the transitions as usual.

For this parameterized system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$, the 2CM M is not halting if and only if

$$\mathcal{S}(\geq 1) \models \Box(q \neq q_{stop}) \ .$$

Assume there is a halting run of M. Let n be larger than the maximal register value occurring in the run. Then, the corresponding run of $\mathcal{S}(n)$ is reaching a state $q = q_{stop}$: for each M computation step, choose one process that is able to increase/decrease the corresponding counter (there is always one since n is larger than all register values) and let this process set $try := 1$, and all the other $try := 0$.

Then, this process i may go to location 4 when all other go to 2. In the next step, i simulates the instruction, setting $qc[i]$ to the next 2CM state, all other processes reach location 3. If q has not yet the right value, the user processes are blocked, since process i cannot proceed. So, the controller adjusts now q. Afterwards, all user processes go to location 0, which finishes the simulation of the step.

In this state, where all processes are in 0, the number of entries $x_l[j] = 1$ equals the value of counter R_l and q has the state corresponding to the one in the M computation. Consequently, there is a corresponding run of $\mathcal{S}(n)$.

Assume now there is a run of some $\mathcal{S}(n)$ that reaches $q = q_{stop}$. Then, each occurrence of a state with $l[i] = 5$ for some i corresponds to an M instruction. The construction ensures that this location can only be reached by *one* process, because this process must have taken transition $\rho_{1\to 4}$ previously.

The computation step of process i from location 4 to 5 together with a possible controller step has the same effect as the corresponding instruction of M. Consequently, there exists a corresponding computation of M that also reaches q_{stop}, hence, M is halting.

Consequently, the undecidable halting problem for 2CM is reduced to the verification problem for $\Pi_0 \cup \Pi_1$ systems with synchronous control, showing that this problem must be undecidable as well. ■

The following theorem implies that the undecidability result holds also for controller-less systems.

Theorem 4.70
For each $\Pi_0 \cup \Pi_1$ system $\mathcal{S}$ with asynchronous control there exists a system $\mathcal{S}'$ without control such that

$$\mathrm{SC}([\![\mathcal{S}]\!]) = \mathrm{SC}(f([\![\mathcal{S}']\!])) \ ,$$

for some function f mapping sequences of states of an instance $\mathcal{S}'(n)$ to sequences of $\mathcal{S}(n)$ states. Moreover, given $\mathcal{S}$, the system $\mathcal{S}'$ is effectively computable.

Proof: Let $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ be a system in the class $\Pi_0 \cup \Pi_1$ with asynchronous control, and assume $\mathcal{V}_Y \neq \emptyset$. For each $y \in \mathcal{V}_Y$ we introduce an array variable with the same name, which is intuitively a "local copy" of the global variables. Moreover, we introduce a "flag" array variable f over $\{c, u\}$ that signalizes whether a process has taken a user transition in the last step, or simulated a controller transition. Furthermore, there is one array variable ph over $\{0, 1\}$.

The basic idea is that all processes cycle through $ph[i] = 0$ and $ph[i] = 1$, in one step taking a user transition or simulating a controller transition, and checking in the other step whether all processes have made the same "choice".

We define for each controller transition $\tau \in T_C$ given by $Q_N \bar{\jmath} : g(\bar{\jmath}) \to \bar{y} := \bar{c}$ we define the transition

$$\begin{aligned}\rho_\tau \stackrel{\text{def}}{=}\ & Q_N \bar{\jmath} : ph[i] = 0 \wedge g[i/j \mid j \in \bar{\jmath}][y[i]/y \mid y \in \mathcal{V}_Y] \\ & \wedge (\bar{\jmath} \neq i\, q\, (\bigwedge_{M \subsetneq \bar{\jmath}} g[i/j \mid j \in M][y[i]/y \mid y \in \mathcal{V}_Y])) \\ & \to \bar{y}[i] := \bar{c}; ph[i] := 1; f[i] := c\end{aligned}$$

with $q = \wedge$ if $Q = \exists$, and $q = \Rightarrow$ otherwise. The controller transition is now taken by a user process i. Therefore, one has to modify the guard, since the controller refers to i as one or more of the $\bar{j}$ variables.

Each user transition $\tau \in T_U$ given by $Q_N \bar{j} : g(i, \bar{j}) \rightarrow \bar{x}[i] := \bar{c}$ is only slightly modified to:

$$\begin{aligned}\rho_\tau \stackrel{\text{def}}{=} & Q_N \bar{j} : ph[i] = 0 \wedge g(i, \bar{j})[y[i]/y \mid y \in \mathcal{V}_Y] \\ & \rightarrow \bar{x}[i] := \bar{c}; ph[i] := 1; f[i] := u\end{aligned}$$

In the second phase it is checked whether all processes simulated the same controller transition, or whether they all took a user transition. If this is not the case, the system deadlocks.

$$\begin{aligned}\rho_{check} \stackrel{\text{def}}{=} & \forall_N j : ph[i] = 1 \wedge \\ & (j \neq i \Rightarrow (f[i] = f[j] \wedge \bigwedge_{y \in \mathcal{V}_Y} y[i] = y[j])) \\ & \rightarrow ph[i] := 0\end{aligned}$$

Let θ' be defined by

$$\theta' \stackrel{\text{def}}{=} \exists_N i_1 : \theta[y[i_1]/y \mid y \in \mathcal{V}_Y] \wedge \forall_N i : \bigwedge_{y \in \mathcal{V}_Y} y[i_1] = y[i]$$

(this formula can be transformed in one fulfilling Restriction 3). Let $\mathcal{S}'$ be the resulting system.

The construction ensures that all processes have always the same ph value. Moreover, if $ph[i] = 0$, all y-copies have the same values.

If eventually $y[i] \neq y[j]$ holds for some processes i, j and some variable $y \in \mathcal{V}_Y$, the system deadlocks. The system deadlocks also immediately if there are processes taking a controller derived transition and others taking a transition derived from a user transition.

For a trace of $\mathcal{S}'(n)$, the idea is simply to project on all states with $ph[i] = 0$ for all processes, this leads to a trace prefix of $\mathcal{S}(n)$.

Hence, we define for a sequence r of $\mathcal{S}'(n)$ states such that all $y[i]$ are equal in $r(0)$,

$$f'(r, i) = \begin{cases} r(i)|_{\mathcal{V}_X} \cup \{y \mapsto r(i)(y)(0) \mid y \in \mathcal{V}_Y\} & \text{if } \forall i : ph[i] = 0 \\ f'(r, i-1) & \text{otherwise} \end{cases}$$

This function picks out the last "valid" state of the sequence, and transforms that state in a $\mathcal{S}(n)$ state (there is always one for each position in any trace

of $\mathcal{S}'(n)$). Then, $f'(r)$ is the sequence of $\mathcal{S}(n)$ states defined by $f'(r)(i) \stackrel{\text{def}}{=} f'(r,i)$, and $f(r) \stackrel{\text{def}}{=} \natural f'(r)$. The latter is the proper sequence of $\mathcal{S}(n)$ states corresponding to all valid states of r.

For each run r of some system instance $\mathcal{S}(n)$, there is a corresponding run of $\mathcal{S}'(n)$: if $\mathcal{S}(n)$ makes a controller step τ, let all processes choose to simulate exactly that transition. Afterwards, the τ_{check} transition can be taken. This leads to a run r' of $\mathcal{S}'(n)$ such that every second state has $ph[i] = 0$ for all i, and all $y[i]$ and $f[i]$ agree in that state. Hence, $f(r') = r$.

Now let r' be a run of $\mathcal{S}'(n)$. By construction, $\mathcal{S}'(n)$ alternates between "normal" transitions and τ_{check}. If τ_{check} is enabled, either all processes took a user transition in the last step, or they all took the same controller step. In both cases, $\mathcal{S}(n)$ can make the corresponding transition. If τ_{check} is not enabled, $\mathcal{S}'(n)$ deadlocks, and up to the previous states, every second state was valid.

Consequently, every run r' of $\mathcal{S}'(n)$ corresponds to a prefix of a $\mathcal{S}(n)$ run, and for each run of $\mathcal{S}(n)$ there is a corresponding run of $\mathcal{S}'(n)$. Hence, the set of all stutterings derived by their traces is equal modulo f. ■

Corollary 4.71
The verification problem is undecidable for $\Pi_0 \cup \Pi_1$ systems without control.

Proof: Follows immediately from Theorem 4.69 and Theorem 4.70, since for safety properties, the stuttering closure equivalence is sufficient. ■

Since every controller-less $\Pi_0 \cup \Pi_1$ system is also one with synchronous control, we immediately get undecidability for synchronous control systems.

Corollary 4.72
The verification problem is undecidable for $\Pi_0 \cup \Pi_1$ systems with synchronous control.

4.7 Decidability for Weak $\Pi_0 \cup \Pi_1$ Systems

In this section we recall that the "weak" synchronous $\Pi_0 \cup \Pi_1$ systems with synchronous control is decidable. This result was shown in [EN96] for a similar full synchronous system class where both existential and universal quantifications may be used on one transition guard. We follow closely this approach. We will extend the $\Pi_0 \cup \Pi_1$ systems to cover also this type of transition guards, and discuss an extension for systems with asynchronous control.

Definition 4.73
The class of *weak* $\Pi^+_{0,1}$ is the class of systems with transitions $\tau \in T_U$ of the form

$$\begin{aligned}\rho_\tau(i) = \phi(i) &\wedge \forall_N \bar{j}_u : \psi(i, \bar{j}_u) \\ &\wedge \exists_N \bar{j}_e : \psi(i, \bar{j}_e) \\ &\rightarrow \bar{x}[i] := \bar{c}_x\end{aligned}$$

and similar controller transitions $\tau \in T_C$ with assignment part $\bar{y} := \bar{c}_y$, such that i is not free in ϕ or ψ.

The transition relation predicate is built up by these transition sets as follows

$$\rho \stackrel{\text{def}}{=} \left(\bigvee_{\tau \in T_C} \hat{\rho}_\tau \right) q \left(\forall_N i : \bigvee_{\tau \in T_U} \hat{\rho}_\tau(i) \right) \quad ,$$

with $q \in \{\wedge, \vee\}$, where $\hat{\rho}_\tau$ is the translation of the guarded command style transition to a predicate, as in Definition 3.26. If $q = \wedge$, we say the system has synchronous control, otherwise asynchronous control. □

Remark 4.74
This class subsumes weak $\Pi_0 \cup \Pi_1$ systems, with both synchronous and asynchronous control. One only needs to restrict quantification to one type in each transition, and use at most one existential quantifier. ♣

The idea is to construct an abstract system for a given parameterized system, such that each path of an instance has a corresponding path in the abstraction.

We assume that a weak $\Pi^+_{0,1}$ system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ is given with transition sets T_C and T_U.

Consider a user transition $\tau \in T_U$ of $\mathcal{S}$, some process index i and some state σ. A transition guard is able to express that all the other processes are

in some local states, depending on the control state and process i'th local state, using universal quantification. Moreover, using existential quantification, a guard can express that there exist some processes in some local states, also depending on the control state and i'th local state.

Thus, given process i'th local state and the control state, all one needs to know is the set of all local states of the current state to evaluate the transition guard.

Technical Note 4.75
Compared to [EN96], one of our transitions τ covers a whole set of their transitions, namely one for each possible local pre-state. Hence, for each local state $s \in \Sigma_L$ one can define

$$\rho_{\tau,s} \stackrel{\text{def}}{=} \bigwedge_{x \in \mathcal{V}_X} x[i] = s(x) \wedge \rho_\tau \ .$$

These transitions correspond to the [EN96] transitions, in that they can be viewed as a transition leaving state s and reaching one particular local post-state (namely s modified by the assignment part of τ), with guards that are boolean combinations of predicates $\exists j.\mathcal{E}(j)$, where $\mathcal{E}(j)$ is a boolean expression containing both control variables and user process variables (perhaps one has to split the transition further to reach exactly this form). Having determined process i'th pre-state (namely s), the expressions in $\rho_{\tau,s}$ referring to $x[i]$ variables may be substituted by constants.

A second difference is that we also investigate systems with asynchronous control. ♣

Inspired by the observation that one only needs to know the set of local states, an abstract system that exactly resembles this information can be defined.

Definition 4.76 (Abstract states)
An *abstract state* is a pair (c, S) such that $c \in \Sigma_G$, and $S \subseteq \Sigma_L$. For each state $\sigma \in \Sigma_{\mathcal{S},n}$ we define $\alpha(\sigma) = (\text{Globalstates}(\sigma), \text{Localstates}(\sigma))$. We say an abstract state (c, S) *represents* σ if $\alpha(\sigma) = (c, S)$. □

Now we define when an abstract state satisfies a given transition guard to be able to formally define the abstract transition relation.

Definition 4.77
Given a transition $\tau \in T_C$, an abstract state (c, S), and $c' \in \Sigma_G$ we define $(c, S), c' \models \rho_\tau$, if $(\sigma, \sigma') \models \rho_\tau$, for some σ and σ' such that $\alpha(\sigma) = (c, S)$ and $\pi_G(\sigma') = c'$.

Similarly, we define $(c, S), s, s' \models \rho_\tau$ for some $\tau \in T_U$ and $s, s' \in \Sigma_L$, if there exists j and $\sigma, \sigma' \in \Sigma$ such that $(\sigma, \sigma') \models \rho_\tau[j/i]$, $\pi_j(\sigma) = s$, $\pi_j(\sigma') = s'$, and $\alpha(\sigma) = (c, S \cup \{s\})$. □

Note that the actual choice of σ and σ' in this definition is not relevant, since if one such pair (σ, σ') satisfies ρ_τ, then this holds for all these pairs. Moreover, one can effectively decide for all $(c, S), c'$ whether $(c, S), c' \models \rho_\tau$ and the analogous user transition formula holds.

The following set of abstract transitions is chosen.

Definition 4.78 (Abstract system)
$(c, S) \xrightarrow{X} (c', S') \in \mathcal{R}$, iff

- there exists a transition $\tau \in T_C$ such that $(c, S), c' \models \rho_\tau$,
- $X \subseteq S \times S'$,
- X is total on S, and X^{-1} is total on S', and
- for each $(s, s') \in X$, there exists a user transition $\tau \in T_U$ such that $(c, S), s, s' \models \rho_\tau$.

The abstract system $\mathcal{A}$ is given by the set of abstract states and $\mathcal{R}$.

A *path* through the abstract system $\mathcal{A}$ is a sequence of states such that adjacent states are in $\mathcal{R}$. □

The label X shows what happens to the user processes in each local state. If some user process is leaving s in that computation step and entering state s', the pair (s, s') is added to X. These information may be used to determine "good" paths that have a concrete counterpart in $\mathcal{S}$, and remove "bad" paths that don't.

Proposition 4.79
Let r be a finite path in $\mathcal{A}$. Then, there exists $n \geq 1$ such that there exists a corresponding prefix of a computation of $\mathcal{S}(n)$.

The concrete sequence corresponds to the abstract path if for all positions j the abstract state in this position represents the concrete state in the position.

This proposition can be used for safety property verification, since it shows that whenever there exists a finite counterexample for the abstract system, it can be concretized for some instance, so it suffices to check all finite paths in the finite abstract system.

Proposition 4.80
For weak $\Pi^{+}_{0,1}$ systems, the verification problem for safety properties is decidable.

To verify liveness properties as well, one can use the construction given in [EN96]: after building the synchronous product $\mathcal{M}$ of the automaton for $\neg\phi$ and the automaton for $\mathcal{A}$, cycles are resolved into a "threaded graph". This graph makes explicit the abstract successor state of each local user state, as given by the abstract transition labels.

The following example shows the need for such a construction, since not all abstract cycles can be concretized, a problem that is also encountered in our verification method based on abstraction when liveness properties should be established, see Chapter 6.

Example 4.81
We consider the system where each user process can be described as illustrated in Figure 4.3.

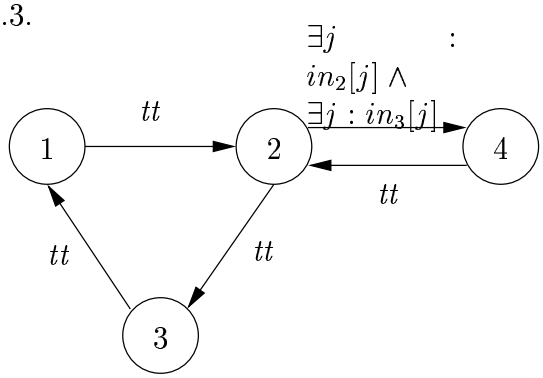

Figure 4.3: Challenging system for liveness verification

The initial state condition is given by

$$\theta \stackrel{\text{def}}{=} \forall i : \exists i_1, i_2 : in_2[i_1] \wedge in_3[i_2] \wedge \neg in_1[i] \ ,$$

so all processes start at location 2 or 3.

We can observe the following: whenever there is a process going from location 2 to location 4, the process moves from the set of processes that started at location 3 and enters the set of processes that started at location 2. Since there are only finitely many processes involved, this can happen only finitely often.

So either the system chooses at some point not to enter location 4 forever, although the transition from 2 to 4 is infinitely often enabled, or the system finally reaches a state where all processes are at the same location, so the transition leading from 2 to 4 can no longer be taken.

Therefore, this system $\mathcal{S}$ satisfies the property

$$\mathcal{S}(\geq 2) \models \forall_d \Diamond\Box\neg in_4[i] \ .$$

Now to the abstract system constructed for $\mathcal{S}$. In the abstraction, one can only observe that there are some processes at each location, so this step from location 2 to location 4 can be taken infinitely often. Consequently, the abstract system does not satisfy the same property. ♣

Theorem 4.82
The verification problem for $\Pi^+_{0,1}$ systems is decidable.

For the formal construction we refer to [EN96]. The basic result is that the partial graph induced by a path of the synchronous product automaton $\mathcal{M}$ as described above has a concrete counterpart, if and only if all its strongly connected components are isolated.

The proof given there can easily be generalized to the property class we use, referring to k processes instead of $0, 1$, or 2 which is investigated in [EN96].

Remark 4.83
The verification problem for all Π classes with asynchronous control and general verification properties can be reduced to the verification problem of the full synchronous versions. This construction works only for non-reactive or full observability.

The reduction can be done as follows: introduce a new boolean control variable b which is initialized arbitrary. Intuitively, if b is tt, it is the controller's turn, otherwise the user processes have to proceed.

In each control transition, add the predicate b to the transition guard, and add $\neg b$ to the guard of each user transitions. Furthermore, introduce an idle transition for users guarded with b, and two idle transitions for the controller guarded with $\neg b$, which set b non-deterministically to tt or ff.

Let $\mathcal{S}$ be a parameterized system with asynchronous control, and assume we have to show some property ϕ. Let $\mathcal{S}'$ be derived by the construction given above. If $\mathcal{S}'$ satisfies $(\Box\Diamond b \wedge \Box\Diamond\neg b) \Rightarrow \phi$, we can conclude $\mathcal{S}$ satisfies ϕ. ♣

4.8 Numeric Variables

In this section we shortly discuss extensions of parameterized systems with numeric variables.

We first observe that it is obvious that systems with two counter variables z_0, z_1 are undecidable, if one allows to compare the counters with 0 (or at least one constant to which the variable can be set), and one can increment and decrement them. In this case, it is straightforward to simulate a 2CM with the parameterized system.

Consequently, if one aims at decidability, one has to investigate very restrictive systems. The following section presents such a class.

4.8.1 Σ_0 Systems

The construction used to decide properties of Σ_0 systems with synchronous control given in Section 4.1.2 is based on a translation to vector addition systems with states (VASS). So in some sense, this translation substitutes array variables by numeric variables.

If one would allow numeric variables in the parameterized system with the same meaning as the VASS counters, so all one can do is to increment them, or decrement them if they are greater zero, it is straightforward to extend this transformation, yielding a VASS as well.

Definition 4.84 (Weak counter)
We say that z is a *weak counter* variable in a parameterized system $\mathcal{S}$, if z is a numeric variable, and the only operations used on z are incrementation by one, checking for $x > 0$, and decrementation for positive x. □

Theorem 4.85
The verification problem for Σ_0^+ systems with *weak counters* is decidable.

Proof: Basically the same construction as in Section 4.1.2 can be used, for each local state and for each weak counter variable one VASS counter is introduced. Checking for $x > 0$ in one $\mathcal{S}$ transition corresponds to decrement the corresponding counter first, and increment it in the second simulation step. ■

4.9 Overview and Conclusions

We summarize the decidability results presented in this chapter. Table 4.2 shows the results of systems with asynchronous processes, Table 4.3 presents results for the different classes of systems consisting of synchronous processes. In both versions, there may be a controller connected to them either synchronously or asynchronously.

Table 4.2: Decidability Results for Σ Classes

Control mode	Σ_0	Σ_1	$\Sigma_0 \cup \Sigma_1$
asynchronous control	decidable	decidable	undecidable (weak+no control)
synchronous control	decidable (even Σ_0^+)	undecidable (even non-react.)	undecidable (weak+no control)

Table 4.3: Decidability Results for Π Classes

Control mode	Π_0	Π_1	$\Pi_0 \cup \Pi_1$	weak $\Pi_0 \cup \Pi_1$
asynchronous control	decidable	decidable (safety)	undecidable (no control)	decidable (even $\Pi_{0,1}^+$)
synchronous control	decidable	decidable (safety)	undecidable	decidable (even $\Pi_{0,1}^+$)

The weak class variants are the variants given in Definition 3.27, where we do not distinguish the other processes from the one taking the transition. Hence, we forbid comparisons $j \neq i$ in transition guards, where i corresponds to the process making the step and j is universally or existentially quantified over processes. Consequently, all conditions specified refer to *all* processes *including* the process taking the transition itself. Interestingly, the ability to compare in this way strengthens the synchronous model such that it becomes undecidable, whereas otherwise the model is decidable even if one mixes both quantification types in one transition guard (this is the class $\Pi_{0,1}^+$).

The Σ_0 classes may both be extended with weak counters as explained in Section 4.8 without losing decidability.

Related work. The asynchronous classes Σ_0 and Σ_1 can be seen as a generalization of the classes presented in [EK00]. We apply a similar idea to prove decidability by showing that it is sufficient to verify system instances up to a computable bound. This idea was in fact already given in [GS92].

For Σ_0 systems, our results are weaker than those presented in [EK00], since in the paper some problems were overlooked. We were able to apply the idea of verification up to a bound for a couple of subproblems of the general verification problem. The general decidability of the verification problem for this class follows from the construction given for Σ_0 with synchronous control. This construction uses a translation from parameterized systems to vector addition systems with states, a construction also used in [GS92]. The systems in [GS92] are asynchronous systems with synchronization steps, which can also be seen as a form of existential (or disjunctive) guard.

The weak $\Pi^+_{0,1}$ class with synchronous control can be seen as a generalization of the class considered in [EN96]. A decision procedure very similar to the one given in the paper can be used for weak $\Pi^+_{0,1}$. Therefore, we only present a rough idea of the procedure, pointing out how they relate and which problems occur if liveness properties are to be proven.

Conclusions. We gave a uniform modeling framework for parameterized systems from which a number of system classes can be derived. Among them are many classical parameterized system classes. We explore the boundary of decidability by removing and adding constraints.

There are many more system classes and ways of parameterization one can think of. All the classes we investigated have shared memory concurrency. It would be interesting to look at systems with other forms of communication, for instance systems using broadcast or systems with message passing. There already is some work on parameterized broadcast protocols [EN98, EFM99].

Furthermore, one could consider other forms of observability as the ones presented, e.g., memory-less control with transitions $x, y \to x'$ and $x \to y'$.

On the other hand, known results indicate that for instance loosing the constraints on index terms to also allow indices of the form $i+k$ for constants k will most often lead to undecidable system classes.

A possible extension of our system class are asynchronous systems where more than one process proceeds in a step. This type of system corresponds to a Σ class with more than one existential quantifier referring to indices making a step.

Chapter 5

Verification of Parameterized Systems by Abstraction

The decidability results in Chapter 4 have shown that for several system classes, the verification problem for certain properties is decidable. Therefore, for such systems, decision procedures provide a way to obtain verification results.

Nevertheless, we also found several interesting undecidable classes, and also many existing parameterized systems are not in a decidable class, and thus, the decision procedures cannot be used. Even worse, all the methods have in common that they do not give reliable positive results applied on systems not belonging to the corresponding system class. If such a procedure applied on a system not in the class finds no counterexample, one cannot conclude correctness of the system.

Moreover, some decision procedures have a very high complexity, such that the decidability result is more a theoretic one, and the procedures are not applicable in many cases. Thus in practice, even for some decidable system classes there is no effective verification method so far.

Due to the undecidability of the general verification problem it is not possible to have a complete and sound verification method. Therefore, one can only try to use a less "perfect" methodology.

We propose to use the the popular approach of verification by abstraction. The idea is to verify a (finite) abstract system that has "more" behavior instead of the original one. This approach is applicable for universally path-quantified properties which express that for all computations should satisfy some property, our LTL properties are of that type. Intuitively, if one over-approximates the system behavior, and all the traces of the over-approximation satisfy a given property, one can conclude that also the orig-

inal (concrete) system satisfies the property.

In case verification of the abstract system fails, this semi-automatic method requires user interaction. If the user is able to find a concrete counterpart for a counterexample found for the abstract system, the concrete system does not satisfy the property. Otherwise, the counterexample may not correspond to any concrete behavior, and the user has to refine the abstract system and redo the verification of the abstraction.

Instead of giving the abstract system and establishing the abstraction relation manually, we propose to give an abstraction relation that relates concrete with abstract states and compute the abstraction automatically.

The user interaction then reduces to refine the abstraction relation if a counterexample for the abstract system is found.

It is not obvious how to use the verification-by-abstraction approach for parameterized systems, since a parameterized system corresponds to a set of systems, and each member of the set should be verified. The notion of abstraction refers only to one system. Our idea is to model a parameterized system as one system and compute finite abstractions of this system to be able to use existing model-checking techniques to verify properties of the abstractions. Our verification method is based on the following:

- We model parameterized systems as higher-order transition systems in the decidable logic WS1S. These are systems with variables ranging over finite sets of natural numbers.

- Given an abstraction relation, the decidability of WS1S allows to automatically compute a finite abstraction of the whole system. This finite abstraction is an abstraction for every instance of the system.

- Model-checking techniques can then be used to establish properties valid for all system instances.

Following this approach one can observe that it is hardly applicable for liveness properties, since a lot of cycles are introduced by abstraction, and these cycles invalidate most liveness properties. Therefore, we will extend the method in Chapter 6 to be able to prove liveness properties as well.

Before we explain the verification approach in detail, we first define the necessary foundations. We start by explaining abstraction, show how the general abstraction idea is reduced to the notion of simulation, and verification by abstraction. Especially, we focus on the conditions under which the approach is applicable. In short, our approach is usable for deadlock-free parameterized systems and predicate abstractions.

We then explain the idea of incremental verification, which means that some properties already established can be used to "improve" the system without changing the system behavior. This improvement usually allows to verify properties that were not as easy verified before.

After explaining shortly the idea of model-checking, which is the technique we use to establish properties for the computed finite abstractions, we present the decidable logic WS1S. We call transitions systems described in this logic WS1S systems and explain how to transform parameterized systems into WS1S systems.

We end this chapter by discussing how to abstract WS1S systems and giving ideas on how to choose abstraction relations from which the abstract systems are computed.

5.1 Verification by Abstraction

We first give the definition of abstraction and the idea of proving properties of systems by abstraction. Intuitively, in this setting a system abstracts another one if it has more behavior, i.e., more traces than the other. Since two systems may have different state spaces, one needs to relate their state spaces to identify corresponding traces.

Definition 5.1 (Abstraction)
Let $\mathcal{S}_C$, $\mathcal{S}_A$ be two systems with state spaces Σ_C, Σ_A and semantics $[\![\mathcal{S}_C]\!]$, $[\![\mathcal{S}_A]\!]$ that are sets of maximal state traces. Let $\alpha \subseteq \Sigma_C \times \Sigma_A$. Then we say that $\mathcal{S}_A$ is an *abstraction* of $\mathcal{S}_C$ with respect to α, denoted by $\mathcal{S}_C \sqsubseteq_\alpha \mathcal{S}_A$, if

$$\begin{aligned} \forall r_C \in [\![\mathcal{S}_C]\!].\exists r_A \in [\![\mathcal{S}_A]\!] : |r_C| = |r_A| \wedge \\ \forall i < |r_C| : (r_C(i), r_A(i)) \in \alpha \ . \end{aligned} \tag{5.1}$$

If α is a total function, this condition reduces to $\alpha([\![\mathcal{S}_C]\!]) \subseteq [\![\mathcal{S}_A]\!]$. □

For a given LTL formula ϕ and a system with state space Σ, let $[\![\phi]\!] \subseteq \Sigma^\omega$ denote the set of models of ϕ.

Before we proceed we give some basic definitions for relations.

Definition 5.2
Let $R \subseteq A \times B$ and $S \subseteq B \times C$ be some relations.

- For a set M, $id_M \stackrel{\text{def}}{=} \{(a,a) \mid a \in M\}$ is the identity relation on M.
- $R\,;S \stackrel{\text{def}}{=} \{(a,c) \in A \times C \mid \exists b \in B : (a,b) \in R \wedge (b,c) \in S\}$ is the *sequential composition* of R and S.

- For $M \subseteq A$, we define $M\,;R \stackrel{\text{def}}{=} id_M\,;R$, and correspondingly $R\,;M$ for $M \subseteq B$.
- $R^{-1} \stackrel{\text{def}}{=} \{(b,a) \mid (a,b) \in R\}$ is the *inverse* of R.
- For $a \in A$, the *relational image* of a is defined by $R(a) \stackrel{\text{def}}{=} \{b \mid (a,b) \in R\}$. This is extended to subsets of A as usual.
- For a sequence $\gamma \in A^{\underline{\omega}}$, we define $R(\gamma) = \{\gamma' \in B^{\underline{\omega}} \mid |\gamma| = |\gamma'| \wedge \forall i < |\gamma| : (\gamma(i), \gamma'(i)) \in R\}$.
- For a set of sequences $\Gamma \subseteq A^{\underline{\omega}}$ we define $R(\Gamma) \stackrel{\text{def}}{=} \bigcup_{\gamma \in \Gamma} R(\gamma)$.

□

Remark 5.3
Using the relational image on sequences, (5.1) can be equivalently expressed by

$$\forall r_C \in [\![\mathcal{S}_C]\!] : \alpha(r_C) \cap [\![S_A]\!] \neq \emptyset \ . \tag{5.1'}$$

♣

The following preservation result [CGL94, LGS+95] is the key property for the verification-by-abstraction approach. It states that it is sufficient to establish a property for the abstract system to conclude that the property holds also for the concrete system.

Theorem 5.4 (Preservation)
Let $\mathcal{S}_C$ and $\mathcal{S}_A$ be two systems, and ϕ_C, ϕ_A be LTL formulae. Then, from

- $\mathcal{S}_C \sqsubseteq_\alpha \mathcal{S}_A$,
- $\alpha^{-1}([\![\phi_A]\!]) \subseteq [\![\phi_C]\!]$, and
- $\mathcal{S}_A \models \phi_A$,

we can conclude $\mathcal{S}_C \models \phi_C$.

In case the abstract system $\mathcal{S}_A$ is finite, one can *automatically* check whether $\mathcal{S}_A \models \phi_A$ holds, using existing model-checking techniques (see Section 5.4).

Intuitively, these conditions express the following. If an abstract system has "more" traces than the concrete one, and we are interested in showing that all traces satisfy some property, then whenever the abstract system

fulfills the property or an even sharper one, it must also be valid for the concrete one.

Similar preservation results hold for any temporal logic without existential quantification over paths, e.g., $\forall CTL^\star$ or $\mu_\Box$ [CGL94, DGG94, LGS+95].

A question that remains is how to establish the abstraction relation between two systems. This can be solved using simulation as explained next.

5.2 Simulation

Simulation establishes abstraction by reducing the global criterion of abstraction (every concrete trace has an abstract counterpart) to verification conditions for each computation step.

Intuitively, the existence of a simulation relation between a concrete and an abstract system expresses that every step the concrete system makes can be simulated by the abstract one by taking a corresponding step. Formally, the simulation relation relates abstract and concrete states.

In this section we extend the system model such that the transition relation ρ is replaced by a *set of transitions* $\mathcal{T}$, where formally, each transition is a transition relation as before. The systems transition relation corresponds to the union of all transitions. We use this model for the rest of the thesis. The reason is, that we are now also interested in the information which transition was taken in a computation step, essential information for the verification of liveness properties presented in Chapter 6.

Let $\mathcal{S}_A = (\mathcal{V}_A, \mathcal{T}_A, \theta_A)$ and $\mathcal{S}_C = (\mathcal{V}_C, \mathcal{T}_C, \theta_C)$ be two systems with state spaces Σ_A and Σ_C, where each state in Σ_A (resp. Σ_C) assigns values to the variables $\mathcal{V}_A$ (resp. $\mathcal{V}_C$). Each transition $\tau_A \in \mathcal{T}_A$ is given by $\tau_A \subseteq \Sigma_A \times \Sigma_A$. The set of initial states is given by $\theta_A \subseteq \Sigma_A$. $\mathcal{S}_C$ is built correspondingly. For each $\tau_C \in \mathcal{T}_C$ we assume there exists a corresponding transition $\tau_A \in \mathcal{T}_A$.

We formally define the notion of simulation [Mil71].

Definition 5.5 (Simulation)
Given two systems $\mathcal{S}_C$ and $\mathcal{S}_A$, we say that a relation $\alpha \subseteq \Sigma_C \times \Sigma_A$ is an *L-simulation* (of $\mathcal{S}_C$ by $\mathcal{S}_A$) if

$$\theta_C \subseteq \theta_A \,;\, \alpha^{-1} \tag{5.2}$$

$$\alpha^{-1} \,;\, \tau_C \subseteq \tau_A \,;\, \alpha^{-1} \tag{5.3}$$

for corresponding transitions $\tau_C \in \mathcal{T}_C$ and $\tau_A \in \mathcal{T}_A$. We denote L-simulation by $\mathcal{S}_C \sqsubseteq^L_\alpha \mathcal{S}_A$.

Similarly, we say that α is a *U-simulation* (of $\mathcal{S}_C$ by $\mathcal{S}_A$), if

$$\theta_C \subseteq \theta_A \,;\, \alpha^{-1} \tag{5.4}$$
$$\alpha^{-1} \,;\, \tau_C \,;\, \alpha \subseteq \tau_A \tag{5.5}$$

U-simulation is denoted by $\mathcal{S}_C \sqsubseteq^U_\alpha \mathcal{S}_A$. □

Figure 5.1 illustrates L-simulation and U-simulation.

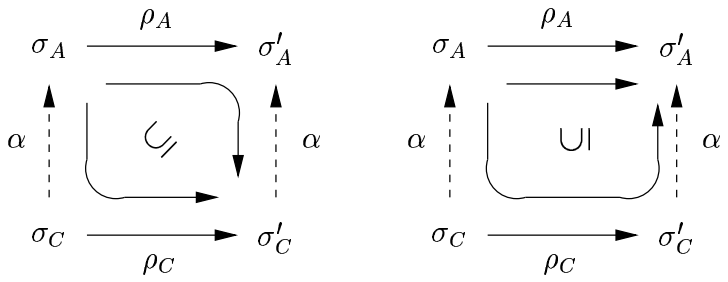

Figure 5.1: L-simulation and U-simulation

There are four different types of simulation, one for each way that the diagram commutes. We refer to [dRE98] for details. We only need L- and U-simulation: U-simulation is the only one of these four simulation notions that can be used to *construct* an abstract system from a concrete one and an abstraction relation. On the other hand, L-simulation corresponds better to the notion of abstraction, as the next theorem shows.

Theorem 5.6
Let $\alpha \subseteq \Sigma_C \times \Sigma_A$ be a simulation of $\mathcal{S}_C$ by $\mathcal{S}_A$. If $\mathcal{S}_C$ is deadlock-free and $\mathcal{S}_A$ is finitely branching, then $\mathcal{S}_A$ is an abstraction of $\mathcal{S}_C$ with respect to α.

Proof: (Sketch) For each trace r of $\mathcal{S}_C$, the initial state of r is an element of θ_C. Consequently, there exists a corresponding state of Σ_A in θ_C and which is related to $r(0)$ by α.

For each step of the concrete system, we conclude by L-simulation there exists some corresponding step of the abstract system. By induction one can show that the set of prefixes of traces of $\mathcal{S}_C$ is a subset of the set of prefixes of $\mathcal{S}_A$ traces (modulo α).

Since $\mathcal{S}_C$ is deadlock-free, each trace of $\mathcal{S}$ is infinite. Since $\mathcal{S}_A$ is finitely branching (which mean in our context that each state has only a finite number of successors), there must exist a corresponding infinite trace of $\mathcal{S}_A$.

Note that if $\mathcal{S}_C$ is not deadlock-free, for a deadlocked trace, a corresponding sequence of states of $\mathcal{S}_A$ is not necessarily maximal, and thus $\mathcal{S}_A$ is not required to be an abstraction of $\mathcal{S}_C$. ■

Hence, if we restrict ourselves to non-blocking systems only (and use finitely branching abstractions which is a harmless restriction from the practical point of view), this theorem shows how to reduce the notion of abstraction to simulation.

We aim at *computing* the abstract system from a given concrete one and an abstraction relation. Obviously, L-simulation is not really suited for that. But U-simulation shows a way to do it: if α and τ_C are given, and $\alpha^{-1};\tau_C;\alpha$ is computable, then the result is obviously a candidate of an abstract transition corresponding to τ_C. In fact, it is the *least* abstract transition abstracting τ_C (adding more pairs to the transition would also lead to an abstraction of τ_C).

Therefore, we have to investigate under which conditions U-simulation implies abstraction. This can be done by considering the relationship of U-simulation and L-simulation. In fact, U-simulation implies L-simulation, if α is a total relation, as the next theorem states.

Theorem 5.7
Let $\alpha \subseteq \Sigma_C \times \Sigma_A$ be total, i.e., there is a pair $(\sigma_C, \sigma_A) \in \alpha$ for all $\sigma_C \in \Sigma_C$. If $\mathcal{S}_C \sqsubseteq^U_\alpha \mathcal{S}_A$, then $\mathcal{S}_C$ L-simulates $\mathcal{S}_A$.

Proof: Since α is total, $id_{\Sigma_C} \subseteq \alpha\,;\alpha^{-1}$. Obviously, $\alpha^{-1}\,;\tau_C\,;\alpha \subseteq \tau_A$ implies $\alpha^{-1}\,;\tau_C\,;\alpha\,;\alpha^{-1} \subseteq \tau_A\,;\alpha^{-1}$. Consequently, $\alpha^{-1}\,;\tau_C = \alpha^{-1}\,;\tau_C\,;id_{\Sigma_C} \subseteq \alpha^{-1}\,;\tau_C\,;\alpha\,;\alpha^{-1} \subseteq \tau_A\,;\alpha^{-1}$. ■

Note that Theorem 5.7 does not hold if α is not total.

From Theorem 5.6 and 5.7 we deduce that a verification method for general LTL properties based on computing abstractions works if

1. the concrete system is deadlock-free, and
2. α is total.

Since we are interested in computing finite abstractions, all abstract systems we use are finitely branching. Nevertheless, for safety properties these restrictions can be weakened.

Safety properties. For safety properties, one can weaken the definition of abstraction such that for each concrete trace, we only require an abstract counterpart that has equal length or is longer.

If there does not exist a counterexample of the abstract system for the safety property, this implies that there is also no counterexample of the concrete system.

L-simulation suffices to establish this type of abstraction, as well. The preservation result for safety properties also holds correspondingly. Therefore, for verification of safety properties, the first restriction can be removed and the verification method works under the condition that α is total.

5.3 Incremental Verification

If we have already proven some invariance property ψ of $\mathcal{S}$, i.e., $\mathcal{S} \models \Box\psi$, we can strengthen Condition (5.5) to

$$\alpha^{-1}\,;(\tau_C \cap \{(\sigma,\sigma') \mid \sigma \models \psi \wedge \sigma' \models \psi\})\,;\alpha \subseteq \tau_A \ . \tag{5.5'}$$

This allows to establish simulation for smaller abstract systems $\mathcal{S}_A$, for which usually more properties can be verified. Intuitively, a smaller abstraction is a better over-approximation of the concrete one. We denote this type of U-simulation by $\mathcal{S} \sqsubseteq_\alpha^{U,\psi} \mathcal{S}_A$. Note that $\mathcal{S} \sqsubseteq_\alpha^{U} \mathcal{S}_A$ holds iff $\mathcal{S} \sqsubseteq_\alpha^{U,tt} \mathcal{S}_A$.

If we compute the abstract transitions as proposed in Section 5.2 by computing the result of

$$\alpha^{-1}\,;(\tau_C \cap \{(\sigma,\sigma') \mid \sigma \models \psi \wedge \sigma' \models \psi\})\,;\alpha \ ,$$

the result will always be a subset of that computed with the original formula. Intuitively, the abstract transitions are then more precise, thus the abstraction is better and often satisfies more properties, since we only investigate properties that should be valid for all system traces. Having a better abstraction corresponds to having less abstract traces. This can formally be stated as follows.

Lemma 5.8
$\mathcal{S} \sqsubseteq_\alpha^{U} \mathcal{S}_A$ and $\mathcal{S} \sqsubseteq_\alpha^{U,\psi} \mathcal{S}'_A$ implies $\mathcal{S}'_A \sqsubseteq_{id}^{L} \mathcal{S}_A$.

Proof: Follows immediately by observing that $\tau'_A \subseteq \tau_A$ holds for each concrete transition τ_C. ■

Note that since the abstract systems might deadlock, in general $\mathcal{S}'_A \sqsubseteq_{id} \mathcal{S}_A$ does not hold.

Counterexamples. One way to obtain invariance properties that can be used to strengthen the abstract system is by analyzing counterexamples obtained by verification attempts. Usually, the analysis gives an idea of a property that is violated by the example and which one expects to hold for the concrete system. Often, such properties can be established first using a

simpler abstraction. If one has found an invariance property, the incremental verification approach can be applied, and the system can be strengthened with the invariant.

Abstraction relation refinement. Another way to deal with properties found by analyzing counterexamples is to "refine" the abstraction relation by introducing a new abstract variable to the abstraction relation that "observes" the new property. Following this approach, counterexamples are used to obtain better and better abstractions. Counterexample guided verification is an active field of research [CGJ+00, LBBO01, CGKS02, CCK+02].

Both, strengthening the system and refining the abstraction relation are examples of *incremental verification*, where information gained so far is used to make further progress. In our experience, this approach is very useful in practice.

We will apply these methods in Chapter 8 where we present a case study of a cache-coherence protocol.

5.4 Model-checking

If we succeed to reduce the verification problem for parameterized systems to the verification of a single finite state system, we then have to care about verifying properties for such systems. Model-checking [QS81, CE81, CGP99] techniques can be used for this. It is a popular algorithmic approach for checking whether a given (usually) finite state system fulfills a temporal logic formula. In contrast to other methods of ensuring quality of programs like testing and simulation, model-checking is able to cover all possible system behaviors.

There are model-checking algorithms for a lot of temporal logic formalisms, we mention here only CTL, the computational tree logic, and LTL that we use in this thesis, but there are many more. Where LTL argues over one *trace* of the system, the underlying time model of CTL is based on a branching structure: at every moment, there is a set of distinct possible futures possible. A formula can quantify over this set, expressing that for all possible futures some property is valid, or for some future this is the case.

One difference of these formalisms is that properties are usually more intuitively expressible in LTL than in the branching-time logic CTL. Both logics are incomparable, i.e., there exist properties expressible in one of each logic but not expressible in the other. Nevertheless, "standard" properties are expressible in both of them. Another well-known logic is CTL*, a logic that subsumes both LTL and CTL.

From the complexity point of view, model-checking CTL is advantageous. CTL model-checking is PTIME-complete and, given a property ϕ and a Kripke structure $\mathcal{S}$, can be done in time $\mathcal{O}(|\phi| \cdot |\mathcal{S}|)$ and space $\mathcal{O}(|\phi| \cdot \log^2 |\mathcal{S}|)$ [CES86, KV95].

LTL model-checking has a higher complexity, it is PSPACE-complete [SC82, SC85], and can be done in time $\mathcal{O}(2^{|\phi|} \cdot |\mathcal{S}|)$ and space $\mathcal{O}((|\phi| + \log(|\mathcal{S}|))^2)$ [VW94].

Since we use LTL, we shortly explain the automata-theoretic model-checking approach for LTL [VW86]. The basic idea is to reduce the verification problem to the emptiness problem for automata on infinite words, called Büchi automata, a problem which is known to be decidable. Assume a Kripke structure $\mathcal{S}$ and an LTL formula ϕ. By $\mathcal{L}(A)$ we denote the set of words recognized by an automaton A.

- Construct the automaton $A_{\mathcal{S}}$ for $\mathcal{S}$.
- Build the automaton for the negation of ϕ, $A_{\neg\phi}$. This can be done effectively for every LTL formula. One could also build A_ϕ and complement this Büchi automaton.
- By observing $\mathcal{L}(A_{\mathcal{S}}) \subseteq \mathcal{L}(A_\phi)$ iff $\mathcal{L}(A_{\mathcal{S}}) \cap \mathcal{L}(A_{\neg\phi}) = \emptyset$ iff $\mathcal{L}(\mathsf{A}_{\mathcal{S}} \times \mathsf{A}_{\neg\phi}) = \emptyset$, the verification problem can be reduced to the emptiness problem for Büchi automata, which is a decidable problem. Here, $\mathsf{A}_{\mathcal{S}} \times \mathsf{A}_{\neg\phi}$ is the synchronous product of both automata.

Using this procedure, one can decide properties of finite abstract systems.

5.5 WS1S Logic

The monadic second-order logic (interpreted over finite words) of one successor, short WS1S, goes back to work of Büchi [Büc60] and Elgot [Elg61], who showed that finite automata and monadic second-order logic interpreted over finite words have the same expressive power. Moreover, these results show that transformations from formulae to automata are effective, as well as transformations from automata to formulae.

We now define the WS1S logic.

Definition 5.9 (WS1S)
Terms of weak second-order logic of one successor (WS1S for short) [Büc60, Elg61, Tho90] 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

$$f ::= b \mid t_1 = t_2 \mid t_1 < t_2 \mid t \in X$$

where b is a boolean variable, t, t_1, and t_2 are terms, and X is a set variable, i.e., a 2nd-order variable.

WS1S formulae are built up from atomic formulae by applying the boolean connectives and quantification over both 1st-order and 2nd-order variables. □

WS1S formulae are interpreted in models that assign finite sub-sets of $\mathbb{N}$ to 2nd-order variables and elements of $\mathbb{N}$ 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 $\text{free}(f)$.

Finally, we recall the following decidability result by Büchi [Büc60] and Elgot [Elg61].

Theorem 5.10
A language of finite words is recognizable by a finite automaton iff it is definable in WS1S, and both conversions, from automata to formulae and vice versa, are effective.

This is a central result for our approach, since it shows that the set of all models of a WS1S formula is representable by a finite automaton. Hence, if we model the concrete system and the abstraction relation in WS1S, computing the set of all models computes the corresponding abstract system.

5.6 WS1S Systems

Now we are able to define WS1S transition systems, which are transition systems with variables ranging over finite subsets of $\mathbb{N}$, and show how they can be used to represent parameterized systems.

Definition 5.11 (WS1S Transition Systems)
A *WS1S transition system* $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ 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.
- $\mathcal{T}$: A finite set of transitions where each $\tau \in \mathcal{T}$ is represented as a WS1S formula $\rho_\tau(\mathcal{V}, \mathcal{V}')$, where primed variables refer to the transition post-state.

- θ: A WS1S formula with $\text{free}(\theta) \subseteq \mathcal{V}$ describing the initial states.

The computations of $\mathcal{S}$ are defined as usual. Let $[\![\mathcal{S}]\!]$ denote the set of computations of $\mathcal{S}$. □

WS1S systems can be used to model parameterized systems as defined in Definition 3.1. Assume that a system $\mathcal{S} = (N, \mathcal{V}, \rho, \theta)$ is given with only boolean types. If one wants to use more general finite types, this can be done by encoding variables of such a type using a number of boolean variables. The main idea is to

- substitute the parameter N by a set P of process indices,
- replace sub-terms $i < N$ by $i \in P$ in all system formulae,
- add the constraint $P' = P$ to the transition relation,
- introduce a set variable X for each array variable x in $\mathcal{V}$,
- replace sub-terms $x[i]$ by $x \in X$ in all formulae, and
- add some formula to the initial state condition expressing that P is initialized with some interval $[0 \ldots k]$, for some k, and add some formula expressing that all set variables are subsets of P.

Intuitively, X is the set of all processes that have an x entry tt. The initialization of P chooses one possible set of indices, this set corresponds to one specific instance of the parameterized system. Each transition of the WS1S system leaves the set P unchanged. Hence, in some sense the parameterization becomes *initialization*.

The normal variables $\mathcal{V}_Y$ and the numeric variables $\mathcal{V}_Z$ need special treatment. They can be modeled by using set-variables with some additional constraints. For a boolean variable y,

- one can use a set variable Y,
- add the formula $\forall i.i \in Y \Rightarrow i = 0$,
- and replace each (atomic) sub-formula y by $0 \in Y$.

A numeric variable z can be modeled similarly by

- introducing a set variable Z,
- adding the formula $\forall i.\forall j.i \in Z \wedge j \in Z \Rightarrow i = j \wedge \exists i.i \in Z$ expressing that Z is a singleton, and

- adding an existential quantification $\exists z.z \in Z.$ in the beginning of the WS1S system description.

The result is a precise translation of the original parameterized system $\mathcal{S}$ into a WS1S system. This WS1S system is in fact bisimilar to $\mathcal{S}$, in the sense that for each instance of $\mathcal{S}(n)$, there is a suitable initialization for P such that the so-initialized WS1S system is bisimilar to $\mathcal{S}(n)$.

Theorem 5.12
Let $\mathcal{S}$ be a parameterized system, and let $\mathcal{S}'$ be the WS1S system resulting by the translation given above. Then, $\mathcal{S}$ is bisimilar to $\mathcal{S}'$.

Proof: There is an obvious bijection f between state space of $\mathcal{S}(n)$ and the state space of $\mathcal{S}'(n)$ (when one assumes that the state space of $\mathcal{S}'$ is not the set of all values for $\mathcal{S}'$ variables, which are all finite subsets of $\mathbb{N}$, but only states fulfilling the "additional" constraints for normal and numeric variables). $\mathcal{S}'(n)$ is the "instance" of $\mathcal{S}'$ where P is initialized to $\{0, \ldots, n-1\}$. So, for $\sigma \in \Sigma$ define

$$f(\sigma)(X) \stackrel{\text{def}}{=} \{i \mid \sigma(x)(i) = tt, i < n\}$$

for all array variables x, where set variable X corresponds to x. For normal variables y define

$$f(\sigma)(Y) \stackrel{\text{def}}{=} \begin{cases} \{0\} & \text{if } \sigma(y) = tt \\ \emptyset & \text{if } \sigma(y) = ff \end{cases}$$

Similarly, define for each numeric variable z

$$f(\sigma)(Z) \stackrel{\text{def}}{=} \{\sigma(z)\} \ .$$

One can check that whenever one of the systems can take a transition, the other system can make a corresponding transition. We leave this prove out here. ■

In practice, the latter two translations for normal and numeric variables need not be applied explicitly in our case, since we use the MONA [HJJ+96] tool to decide formulae. Instead, we simply can use the predefined MONA types. See Section 7.2 for a few words on MONA.

The following example illustrates the translation.

Example 5.13
Recall the resource allocation algorithm given in Example 3.2. Variable `s` is encoded by two booleans, `s_a` and `s_b`. Similarly, the array `l` is encoded by two boolean arrays `l_a` and `l_b`. The encoding is given by:

`l[i]`	`l_a[i]`	`l_b[i]`
1	*ff*	*ff*
2	*tt*	*ff*
3	*ff*	*tt*
4	*tt*	*tt*

Variable `s` is encoded similarly.

We give the translations in MONA [HJJ+96, KM01] syntax. Most of the syntax is self-explaining. Universal quantification over booleans is denoted by `all0`, `all1` is quantification over natural numbers, and `all2` quantifies over finite sets of natural numbers. Existential quantification is denoted correspondingly by `ex0`, `ex1`, and `ex2`. Conjunction is denoted by `&`, disjunction by `|`.

We start with the initial state predicate, given by

$$\theta \stackrel{\text{def}}{=} s = 2 \wedge \forall i < N.l[i] = 1 \wedge \neg c[i] \ ,$$

which has the following shape after encoding `l`:

$$s = 2 \wedge \forall i < N.\neg l\_a[i] \wedge \neg l\_b[i] \wedge \neg c[i] \ .$$

Translated to WS1S it looks as follows:

```
s = 2
& all1 i: i in P =>
    (i notin l_a & i notin l_b & i notin c)
```

This could also be written more concisely as:

```
s = 2
& l_a = empty & l_b = empty & c = empty
```

We show the translation of the first transition leaving location 1 and entering location 2:

$$(l[i] = 1 \wedge s > 0 \wedge l'[i] = 2 \wedge s' = s - 1$$
$$\wedge\, c' = c \wedge \forall j < N.j \neq i \Rightarrow l'[j] = l[j])$$

Omitting the encoding step, the translation to WS1S of this transition is given below:

```
# guard: l[i]=1, s>0
(ex1 i: i in P & i notin l_a & i notin l_b & s > 0
# effect
& i in l_a' & i notin l_b' & s' = s-1
& c' = c
& (all1 j: (j in P & j ~= i) =>
    ((j in l_a' <=> j in l_a) & (j in l_b' <=> j in l_b))))
```


5.7 Abstracting WS1S Systems

We now want to verify WS1S systems by abstraction, based on the ideas presented in Section 5.1 and 5.2.

Let $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ be a WS1S system and let α be a total boolean abstraction relation given by a WS1S formula $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A})$, where $\mathcal{V}_\mathcal{A}$ are all the abstract boolean variables. Based on U-simulation, as described in Section 5.2, abstract initial states and abstract transitions can be given as WS1S predicates.

The initial states of the abstract system can be characterized by the formula

$$\hat{\theta}_A \stackrel{\text{def}}{=} \exists \mathcal{V} : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \ . \tag{5.6}$$

Note that this is a WS1S formula. For each concrete transition τ, there is one abstract transition τ_A in the set of abstract transitions $\mathcal{T}_A$ and it is described by the formula

$$\rho_{\tau_A} \stackrel{\text{def}}{=} \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A}) \tag{5.7}$$

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

To *compute* the abstract system, one has to find all abstract states satisfying these formulae. Since both $\widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A})$ and the transitions in $\mathcal{T}$ are expressed in WS1S, the characterizing predicates (5.6) and (5.7) are WS1S predicates. By the decidability result for WS1S given in Theorem 5.10, the set of all models for each formula can be computed effectively, which gives us an *effective* construction of the abstract system.

Let $\mathcal{S}_A = (\mathcal{V}_\mathcal{A}, \mathcal{T}_A, \theta_A)$ be the result of this computation. This system $\mathcal{S}_A$ is an abstraction of $\mathcal{S}$ *by construction*:

Proposition 5.14
$\mathcal{S} \sqsubseteq^U_\alpha \mathcal{S}_A$, thus $\mathcal{S}_A$ is an abstraction of $\mathcal{S}$ with respect to α.

Since the abstract variables are booleans, the constructed abstract system is finite state and can be model-checked.

Abstraction relations. We know by Theorem 5.7 that abstraction relations are required to be total in order to make the method applicable. Therefore we restrict ourselves to relations of a generic shape which are guaranteed to be total.

Definition 5.15
An *abstraction predicate* is a predicate $\phi(\mathcal{V})$ over $\mathcal{V}$. □

The generic abstraction relation shape is based on abstraction predicates. So assume that a family of abstraction predicates $\phi_i(\mathcal{V})$, for $1 \leq i \leq k$, is given. Each such predicate identifies some concrete state property. We define the abstraction relation based on these predicates by

$$\widehat{\alpha} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} (a_i \Leftrightarrow \phi_i(\mathcal{V})) \ . \tag{5.8}$$

This abstraction relation is both total and functional. The abstract system constructed from this abstraction relation has variables $\mathcal{V}_\mathcal{A} = \{a_1, \ldots, a_n\}$. We restrict ourselves to this type of relation. This ensures that results of our verification method are correct, i.e., the concrete system satisfies a property if the computed abstract system does.

Nevertheless, it is not clear how to choose abstraction predicates; see Section 5.8 for heuristics on this topic.

Incremental verification. Following the incremental verification approach presented in Section 5.3, if we have already proven that the WS1S system $\mathcal{S}$ satisfies some invariance property ψ given as a WS1S predicate, we can compute more precise abstract transitions based on the following characterizing predicate:

$$\begin{aligned} \rho_{\tau_A} \stackrel{\text{def}}{=} \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \psi(\mathcal{V}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \\ \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A}) \wedge \psi(\mathcal{V}') \ . \end{aligned} \tag{5.9}$$

We say that we *strengthen* $\mathcal{S}$ with invariant ψ. This predicate corresponds to computing the abstract transition with respect to the strengthened condition (5.5′) for U-simulation. The abstract initial states are characterized correspondingly.

Relationship to the original parameterized system. Recall that the method we propose consists in translating a parameterized system $\mathcal{S}$ to a WS1S system $\mathcal{S}'$ and computing its abstraction $\mathcal{S}'_A$. Assume a set of abstraction predicates ϕ_i on the parameterized system, which are translated

to WS1S abstraction predicates ϕ'_i in the same way as the system. Let the abstraction relation α' be constructed from the ϕ'_i.

Then $\mathcal{S}'_A$ is clearly an abstraction of every system instance $\mathcal{S}(n)$. Intuitively, if there is some trace of $\mathcal{S}(n)$, by Theorem 5.12, there is some initialization such that there is a corresponding trace of $\mathcal{S}'$, and hence, a corresponding trace of $\mathcal{S}'_A$.

One could also think of building an abstraction relation α directly based on the ϕ_i and construct an abstraction of a system instance $\mathcal{S}(n)$. This can effectively be done by adding a corresponding formula to $\mathcal{S}'$ expressing $P = \{0, \ldots, n-1\}$ and computing the abstraction as before, yielding an abstract system $\mathcal{S}^n_A$. Obviously, the state space of $\mathcal{S}^n_A$ is the same as that of $\mathcal{S}'_A$, so the question is how they relate to each other.

Since in $\mathcal{S}'$ variable P is existentially quantified, the answer to that question is that each transitions of $\mathcal{S}^n_A$ is a *subset* of the corresponding transition of $\mathcal{S}'_A$, so each transition τ_A of $\mathcal{S}'_A$ fulfills $\tau_A = \bigcup_{n \geq 1} \tau^n_A$, where τ^n_A is the corresponding transition of $\mathcal{S}^n_A$.

Our tool set PAX can be used to automatically compute abstractions as described here, and analyze the abstract systems. For a description on PAX see Chapter 7.

5.8 Abstraction Heuristics

The main user interaction needed for verification by the method we propose is to choose suitable abstraction predicates. This cannot be done automatically, since the general verification problem for parameterized systems is undecidable.

The choice of abstraction predicates depends on both the concrete system and the property one needs to establish, therefore it is hard to give precise and generally useful heuristics.

Assume that $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ is a WS1S system. We restrict ourselves to abstraction relations based on abstraction predicates ϕ_i as presented in Section 5.7.

5.8.1 Existential Abstraction

The first heuristic is inspired by the decidability results given in Section 4.1.1 and Section 4.7. There, asynchronous systems with existential quantification over indices are investigated, and synchronous systems with both existential and universal quantification. In the decidability constructions one can ob-

serve that only the set of local states induced by a state enables a particular transition.

Therefore, the idea of existential abstraction is to introduce for each set variable an abstraction predicate that is true whenever the set is empty. Intuitively, this abstract variable observes whether there is some process with array entry tt. The corresponding abstraction predicate is $\phi_X \equiv X = \emptyset$ for a set variable $X \in \mathcal{V}$.

This alone is usually not sufficient, so the abstraction relation must be strengthened by adding more abstraction predicates. These are derived both from the property to verify and from the system description. Assume that we have to verify an invariance property ϕ and that each transition is given by a predicate

$$\exists i : G \wedge L(i) \wedge C(i) \rightarrow \mathcal{V}' := e(\mathcal{V}, \mathcal{V}') \ ,$$

where

- G is a 1st-order closed WS1S formula describing some *global condition*, for example, whether there are any processes in a particular local state.
- $L(i)$ is a quantifier-free formula expressing some conditions on process i'th local state.
- $C(i)$ is a condition on both process i'th local state and other processes. We call such conditions the *context* of process i.

Then, one can start with an abstraction based on predicates

$$a_G \equiv G$$
$$a_L \equiv \exists i : L(i)$$
$$a_C \equiv \exists i : C(i)$$

Additionally, in order to make the property ϕ that is to prove "observable" in the abstraction, one has to introduce abstract variables for each state predicate occurring in ϕ.

If this does not work, practical experience show that one can on the one hand try to split the predicates used before, such that partial information is already observable. On the other hand, it is often useful to combine several predicates, for example, some local conditions $L(i)$ and context conditions $C(i)$, which might be derived from different transitions, and using an abstract predicate

$$a_{L,C} \equiv \exists i : L(i) \wedge C(i) \ .$$

5.8.2 Counting Abstractions

A second idea can be deduced by the construction for decidability for Σ_0 systems with synchronous control given in Section 4.1.2. There, a VASS represents system instances, counting the number of processes in local states. Obviously, this representation cannot be used as an abstraction, because it is not finite state. But one can use a restricted *counting abstraction* instead that "counts" the number of processes in a particular local state up to a certain bound b. So, all states with $j \geq b$ processes in a local state will be identified.

This heuristic is a generalization of the existential abstraction. The existential abstraction can be seen as counting abstraction distinguishing only value 0 and greater or equal zero.

This idea can be carried over to WS1S systems by using predicates observing the number of identifiers in a given set $X \in \mathcal{V}$. The following predicates can be used to "count" up to a bound n (the translation of these predicates to WS1S is straightforward):

$$
\begin{aligned}
a_{X,0} &\equiv X = \emptyset \\
a_{X,\leq 1} &\equiv (\forall j_1, j_2 : \{j_1, j_2\} \in X \Rightarrow j_1 = j_2) \\
a_{X,\leq 2} &\equiv (\forall j_1, j_2, j_3 : \{j_1, j_2, j_3\} \in X \Rightarrow (j_1 = j_2 \vee j_1 = j_3 \vee j_2 = j_3)) \\
&\ldots \\
a_{X,\leq n} &\equiv (\forall j_1, \ldots, j_n : \{j_1, \ldots, j_n\} \in X \Rightarrow (\bigvee_{1 \leq k < l \leq n} j_k = j_l))
\end{aligned}
$$

More generally, one can also count the number of identifiers that satisfy a given predicate, e.g., some local conditions $L(i)$ derived from a transition description, as explained in Section 5.8.1. The abstraction predicates are similar to the previous ones, only that $j_l \in X$ is replaced by $L(j_l)$.

This kind of abstraction is especially useful to prove mutual exclusion or similar properties, since it is essential for this type of property how many processes reach certain states.

5.8.3 Focusing Abstractions

For the class of so-called *universal* progress or response properties, we use a slightly different type of abstraction relation. Universal properties are basically the verification properties presented in Section 3.3, where we use universal quantification over process identifiers. The only difference is that we reason there about parameterized systems, and here we investigate arbitrary WS1S systems with set variables.

For example, universal properties can express that each single process i eventually makes some progress, or that each request by i to j eventually is answered by j. The abstractions presented so far are not sufficient for this type of property, since one is not able to reason about *specific* processes.

Liveness properties like the response property described before are not easily verified using abstraction, since abstraction introduces a lot of behavior that has no concrete counterpart, but prevents us to verify liveness properties. We address this problem in Chapter 6.

Throughout this section, let $\mathcal{S}$ be a WS1S transition system with variables $\{X_1, \ldots, X_k\}$.

Definition 5.16 (Universal property)
Let ϕ be an LTL formula with $\{X_1, \ldots, X_k\}$ as free set-variables and free 1st-order variables $\{p_1, \ldots, p_n\}$. Then, ϕ is called a *universal property.*

A computation γ satisfies ϕ if and only if for every injective mapping $\mathcal{I}'$ from $\{p_1, \ldots, p_n\}$ into $\mathbb{N}$, γ satisfies $\phi[\mathcal{I}'(p_1), \ldots, \mathcal{I}'(p_n)/p_1, \ldots, p_n]$. □

In other words, the computation γ satisfies ϕ, if it satisfies all the temporal formulae obtainable by instantiating the variables $p_1, \ldots, p_n$.

Abstraction relation. The abstraction relation is chosen by abstraction predicates as before, but now the 1st-order variables $p_1 \ldots, p_n$ may occur freely in them. For example, one can use predicates like

$$a_{p,X} \equiv p \in X$$

expressing that the p is contained in X. Intuitively, from the point of view of a parameterized system translated into a WS1S system, this can be used to reason about one process p being in a particular state. Hence, the behavior of one arbitrary but fixed process is observed. Let α be obtained by a set of abstraction predicates in the usual way. Then, α has free variables $p_1, \ldots, p_n$.

The central question is how this abstraction relation can be used to compute abstractions of $\mathcal{S}$.

Construction of $\mathcal{S}_A^c$. We observe that instantiating the free p_j variables in the abstraction predicates results in a valid abstraction relation, sufficient to compute a corresponding abstract system. For the sake of readability, we restrict ourselves to only one free variable here, called p.

Let $c \in \mathbb{N}$. If we instantiate p by c, then we are able to construct abstractions $\mathcal{S}_A^c$ as usual, by computing the set of all models of

$$\xi(c, \alpha, \tau) \equiv \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}[c/p](\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}[c/p](\mathcal{V}', \mathcal{V}'_\mathcal{A})$$

for each concrete transition τ, and by computing the initial states

$$\xi(c, \alpha) \equiv \exists \mathcal{V} : \widehat{\alpha}[c/p](\mathcal{V}, \mathcal{V}_{\mathcal{A}}) \ .$$

With $\xi(p, \alpha, \tau)$ and $\xi(p, \alpha)$ we denote the original predicates with free variable p.

The following observation is easily established.

Proposition 5.17
$\mathcal{S} \sqsubseteq^{U}_{\alpha_c} \mathcal{S}^c_A$, where α_c is the abstraction relation corresponding to the predicate $\widehat{\alpha}[c/p]$.

Since $\widehat{\alpha}[c/p]$ and ρ_τ are WS1S formulae, by Büchi and Elgot's result, $\mathcal{S}^c_A$ can be constructed effectively.

We now show that the set $\{S^c_A \mid c \in \mathbb{N}\}$ can be effectively constructed. For $c \in \mathbb{N}$, we define the following WS1S formula:

$$\begin{aligned}\Delta(c) \stackrel{\text{def}}{=} & (\exists \mathcal{V}_{\mathcal{A}} : \xi(c, \alpha) \not\leftrightarrow \xi(p, \alpha)) \\ & \vee \bigvee_{\tau \in \mathcal{T}} (\exists \mathcal{V}_{\mathcal{A}}, \mathcal{V}'_{\mathcal{A}} : \xi(c, \alpha, \tau) \not\leftrightarrow \xi(p, \alpha, \tau))\end{aligned}$$

Thus, $\exists p.\Delta(c)$ expresses that there exists a $k \in \mathbb{N}$ such that the abstract system S^k_A is *different* from S^c_A. By Büchi and Elgot's result, this is decidable, and moreover, such a k is *effectively computable*. The set $\{S^c_A \mid c \in \mathbb{N}\}$ is computed by the algorithm given in Table 5.1. This algorithm converges,

Table 5.1: Algorithm computing all focusing abstractions

```
x := 1;
repeat
    R := R ∪ {S^x_A};
    I := I ∪ {x};
    choose x ∈ ℕ that satisfies ∃p. ⋀_{j∈I} Δ(j)
      (if not satisfiable set x :=⊥)
until x =⊥;
return(R)
```

since there are only finitely many different abstract transition systems and each execution of the body of the loop adds a new abstract transition system to R.

Intuitively, this set R describes all possible observables for some p, so if all these system satisfy a certain property, then this property is valid for all instantiations of p. To be able to verify a given property ϕ, this property is required to be observable by the abstraction, so one has to introduce abstraction predicates for all state formulae of $\mathcal{S}$, which may contain the free 1st-order variables, here only p.

Then, denote by ϕ_A the translation of ϕ that replaces every state formula ψ in ϕ by the corresponding abstract variable a_ψ.

Corollary 5.18
If $\mathcal{S}_A^k \models \phi_A$ for all $\mathcal{S}_A^k \in R$, then $\mathcal{S}$ satisfies the universal property ϕ.

Construction of $\mathcal{S}_A$. Although $\{S_A^i \mid i \in \mathbb{N}\}$ is theoretically computable, it is computationally costly to do. Therefore, we show how one can construct a system that abstracts each of the elements of this set, and hence, by transitivity of $\sqsubseteq$ abstracts $\mathcal{S}$.

A first solution could consist in defining the transitions of $\mathcal{S}_A$ by the following WS1S formula:

$$\exists \mathcal{V}, \mathcal{V}' : (\exists_P p : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A})) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge (\exists_P p : \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A})) \ .$$

This can be used, but one can do better.

Instead, a more precise abstract system can be obtained by defining the abstract transitions of $\mathcal{S}_A$ by the following WS1S formula:

$$\exists_P p : \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A}) \ . \tag{5.10}$$

Thus, we make sure that the choice of p in the concretizations of the source and target states of an abstract transition is the same. We can then show the following:

Proposition 5.19
$\mathcal{S} \sqsubseteq^U_{\alpha'} \mathcal{S}_A$ with $\widehat{\alpha}' \equiv \exists p.\widehat{\alpha}(p)$, hence, $\mathcal{S}_A$ is an abstraction of $\mathcal{S}$ with respect to α'.

Using the same transformation for ϕ as in Corollary 5.18, we have:

Proposition 5.20
If $\mathcal{S}_A \models \phi_A$, then $\mathcal{S}$ satisfies the universal property ϕ.

5.8.4 Examples

We present a some abstractions usable for the resource allocation algorithm. A more challenging verification example can be found in Chapter 8.

We apply these ideas to Example 3.2. We start with a counting abstraction that can be used to verify the correctness of the algorithm.

Example 5.21
For the resource allocation algorithm we choose a counting abstraction that counts for location 2, 3, and 4, whether there are no processes in this location, or whether there is less or equal one process there. The following two abstraction predicates are used for location 2, the predicates for the other locations are similarly.

```
# abstract variable: a_l2_empty
(all1 j: j in P => (j notin l_a | j in l_b))

# abstract variable: a_l2_leq1
(all1 j, k: j in P' & k in P' =>
  ((j in l_a' & j notin l_b'
    & k in l_a' & k notin l_b')
    => j=k))
```

The idea is that there can be at most two processes in this area, so we choose another variable observing whether this is the case. In some sense, this replaces the counting abstraction predicates for each location and value 2. We also observe the value of `s` following the idea of counting abstraction.

```
# abstract variable: a_s_eq0
(s = 0)
# abstract variable: a_s_leq1
(s <= 1)
# abstract variable: a_s_leq2
(s <= 2)

# abstract variable: a_l234_leq2
(all1 j, k, l: j in P & k in P & l in P =>
  ((j in l_a union l_b & k in l_a union l_b & l in l_a union l_b)
    => (j=k | k=l | j=l)))
```

Finally, we use two properties to relate the `c[i]` values with the `l[i]` values, `c[i]` has value *tt* if and only if `l[i]=3`. Moreover, we need a variable to observe the property of interest.

```
# abstract variable: a_s_c_l
```

```
(all1 j: j in P => (j in c <=> (j notin l_a & j in l_b)))

# abstract variable: a_prop
(all1 j, k, l: j in P & k in P & l in P =>
  (j in c & k in c & l in c
   => (j=k | k=l | j=l)))
```

♣

As second simple example we present a focusing abstraction usable to show how $c[i]$ value and $l[i]$ value are related.

Example 5.22
The focusing abstraction based on the following abstraction predicates can be used to observe the behavior of one process:

```
# abstract variable: a_p_la
(p in l_a)

# abstract variable: a_p_lb
(p in l_b)

# abstract variable: a_p_c
(p in c)
```

Observe that p is simply a free variable here. Technically, this variable is existentially quantified in the system description, as shown in (5.10).

The corresponding abstract system can be constructed from this abstraction predicates, a state exploration of the abstraction yields the following set of reachable states. Here, 1 denotes tt, 0 denotes ff, and the first line gives the order of the abstract variables.

```
% a_p_la a_p_lb a_p_c
s0: 000
s1: 100
s2: 011
s3: 110
```

This shows that the following property is valid for this system:

$$\Box(p \in \mathtt{c} \Leftrightarrow (p \notin \mathtt{l\_a} \wedge p \in \mathtt{l\_b}))\ ,$$

a property that corresponds to the following property of the original parameterized system:

$$\forall_d\, \Box(c[p] \Leftrightarrow l[i] = 3)\ .$$

Since this is an *invariance* property, we can now strengthen the system with this property, as described in Section 5.3. ♣

For a third example we modify the algorithm for the sake of brevity. Here, we use $s := 1$ as initial value instead of $s := 2$. So, mutual exclusion should be ensured for this algorithm.

Example 5.23
The abstraction relation used here focuses on two 1st-order variables `p1` and `p2`, that stand for arbitrary process identifiers. For each process we introduce variables that observe the location of each of these two processes. The following predicates are used for `p1`, the predicates for `p2` are similar:

```
# abstract variable: a_p1_la
(p1 in l_a)

# abstract variable: a_p1_lb
(p1 in l_b)
```

Moreover, we observe the value of `s` as in Example 5.21, and have abstraction predicates for observing also processes different from `p1` and `p2`:

```
# abstract variable: a_other_l2
(ex1 j: j in l_a & j notin l_b)

...

# abstract variable: a_l234
(all1 j, k:
  (  (j in l_a | j in l_b)
   & (k in l_a | k in l_b))
  => j=k)
```

Constructing the abstract system and translating it to the input for some model-checker, one can easily prove

$$\forall_d \, \Box\neg(l[p_1] = 3 \wedge l[p_2] = 3) \ .$$

♣

5.9 Blocking Systems

Since our verification method only works for non-blocking systems, it is of interest how one can formally prove that a system is deadlock-free. We assume that a WS1S system $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ is given.

A first approach is simply to ensure that every state has a τ-successor, for some transition $\tau \in \mathcal{T}$, which can be expressed by

$$\forall \mathcal{V} \exists \mathcal{V}' : \bigvee_{\tau \in \mathcal{T}} \rho_\tau(\mathcal{V}, \mathcal{V}') \ . \tag{5.11}$$

If this formula is valid, which can be checked automatically, the system is deadlock-free.

Unfortunately, there may be non-blocking systems that do not fulfill (5.11). If a system deadlocks only in *non-reachable* states, it is nevertheless non-blocking. In this case we propose to use the verification method for invariance properties to strengthen the system, so that non-reachable states are not reachable in the strengthened system. Then, by deciding validity of Formula (5.11) one can check whether the strengthened system, and thus also the original system, is non-blocking.

If this succeeds, one can verify liveness properties as well, using the original system, or the strengthened one.

Chapter 6

Liveness Verification by Abstraction

In this chapter we extend the verification method for parameterized systems using abstraction presented in Chapter 5 to handle liveness properties as well. The abstraction process introduces a lot of cycles into the abstract system, cycles that do not have a concrete counterpart but prevent verification of liveness properties.

We give an algorithm that can be used to derive fairness constraints that are guaranteed to be satisfied for the concrete WS1S system. Since each abstract transition corresponds to one concrete transition, these fairness constraints can be lifted to the abstract system to rule out some behavior without concrete counterpart.

We show how augmentation of the abstract system allows to express these fairness constraints. Technically, augmentation is extending the abstraction with a new boolean variable for each transition, a variable which is set to true whenever the transition is taken.

Liveness properties are often only provable under the assumption of additional fairness conditions that the concrete system satisfies, e.g., a condition expressing that the scheduler is fair such that every process which continuously tries to execute some command will eventually succeed. These additional fairness assumptions are clearly also needed on the abstract level, so we show how to lift them to the abstract system.

6.1 Abstractions and Liveness Verification

A well-known obstacle to the verification of liveness properties by abstraction is that often the abstract system has computations looping continuously

through some cycle that do not correspond to computations of the concrete system.

The following simple example illustrates this problem.

Example 6.1 (Simple mutual exclusion)
Consider the parameterized system consisting of processes described in Figure 6.1. Initially, all processes are at ℓ_0, and all *turn* variables are set to *ff*.

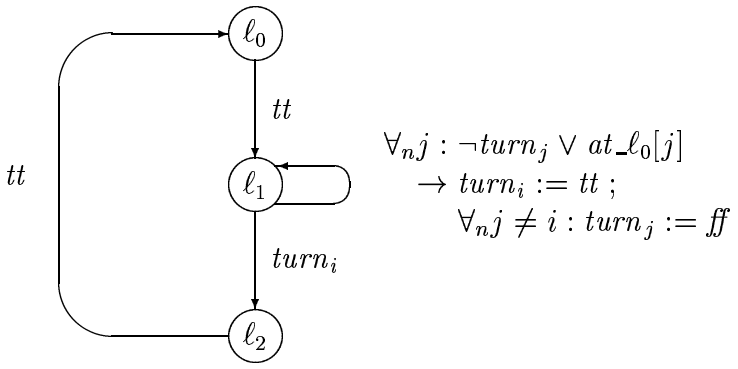

Figure 6.1: Simple mutual exclusion algorithm

Location ℓ_2 represents the critical section. We use t_{xy} to denote the transition from location ℓ_x to ℓ_y.

Modeling this system using array variables is straightforward, for example one can choose three boolean arrays, one for each location, and one array for *turn*. A WS1S system can be obtained using the translation given in Chapter 5, with set variables at_ℓ_0, at_ℓ_1, at_ℓ_2, and *turn*.

We are interested in the property that each process p reaches its critical section infinitely often, i.e., $\forall_n p : \Box\Diamond at\_\ell_2[p]$, which is a universal liveness property.

Obviously, one has to assume some fairness conditions to prove that, namely that the self-loop t_{11} is eventually taken if it is infinitely often enabled (strong fairness), and moreover, one has to assume that all other transitions are eventually taken if they are continuously enabled (weak fairness), since otherwise only some other processes could make progress.

Since we are interested in a universal property, one can try for instance a *focusing* abstraction relation as explained in Section 5.8.3 for verification. Intuitively, such an abstraction observes one process, using the following predicates:

```
# abstract variable: a_p_turn
(p in turn)
# abstract variable: a_p0
```

```
(p in l0)
# abstract variable: a_p1
(p in l1)
# abstract variable: a_p2
(p in l2)
```

Computing the corresponding abstract system, there is a loop in the abstract system as follows: there is a computation step, where the transition from ℓ_0 to ℓ_1 is taken, and this step goes from abstract state 0010 to 0010; this is the abstract state where it is not p's turn and p is in location ℓ_1.

The next few lines show a fragment of the abstract transition relation as computed by our tool PAX, see Chapter 7 for more details on PAX. An "X" denotes that both values 0 and 1 are possible in the successor state.

```
Variables: a_p_turn a_p0 a_p1 a_p2 -> a_p_turn' a_p0' a_p1' a_p2'
Abstract transitions:
t01 : 0000 -> 00X0
t01 : 0001 -> 00X1
t01 : 0010 -> 0010
t01 : 0011 -> 0011
t01 : 0100 -> 0010
t01 : 0100 -> 0100
```

Of course, such a loop can be taken forever, such that process p does not progress in the abstract system. On the other hand, in any concrete system instance, such a loop can only be taken finitely often, since if finally all process have left ℓ_0, there is no process left that can take this transition. ♣

As this example shows, one has to distinguish between behavior preventing us from liveness verification that has its source in the concrete system, and behavior that is introduced by the abstraction process. In case of behavior of the concrete system, one can sometimes assume additional fairness assumptions that remove this type of behavior. Hence, although the system does not satisfy the liveness property, it does under these assumptions. Behavior introduced by abstraction is more difficult to handle, since additional assumptions of the abstraction need to be justified by the concrete system.

The standard approach to prove that some system can only make some action finitely often is to choose a well-founded set and a function mapping system states into this set, and to show that each system step decreases this value. This approach cannot be used here, since the decreasing behavior is simply not observable on the abstract level.

What can be proved is that some transitions of the concrete WS1S system can only be taken infinitely often if some other transitions are also taken

infinitely often, using the idea of well-founded ranking values. The result is a fairness condition that the WS1S system satisfies. Since every concrete transition corresponds exactly to one abstract transition, one can use this correspondence to lift these fairness conditions to the abstract system. Intuitively, if some trace of the abstraction violates such a fairness condition, there cannot be any concrete counterpart of it.

General idea. Our approach for liveness property verification is to compute first fairness conditions (guaranteed to be) valid for the concrete system, or in case of a translated parameterized system, for every instance of the original parameterized system $\mathcal{S}$. Then, each trace of an abstraction of $\mathcal{S}$ not satisfying such a fairness condition cannot have a concrete counterpart *in any instance*. Therefore, it is safe to remove such a trace.

We present a marking algorithm that given a concrete system, marks some transitions as decreasing and others as increasing transitions. From these markings, strong fairness conditions are derived that can be lifted to the abstract system. The enriched abstraction has the property that to each concrete computation corresponds an abstract *fair* one. The enriched abstract system is used to prove liveness properties of the WS1S systems. Since parameterized networks can be transformed into WS1S systems, this shows how to prove liveness properties for parameterized networks.

These strong fairness conditions are based on the concrete system transitions, requiring that some transitions cannot be taken infinitely often. This is the reason why we have refined our general model to this type of transition relation based on a set of transitions $\mathcal{T}$ instead of a single transition relation predicate ρ.

We restrict ourselves to systems that contain some set variable P that is not changed during computation steps, a property which is formally defined as follows.

Definition 6.2 (P-invariance)
A system $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ with $P \in \mathcal{V}$ is called *P-invariant*, if $\rho_\tau \Rightarrow (P' = P)$ is valid for all transitions $\tau \in \mathcal{T}$. □

This condition is always satisfied for WS1S systems that are derived by translation of a parameterized system, namely for the variable P representing the set of all processes.

Throughout this chapter, we fix a P-invariant WS1S system $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$ and an abstraction relation α given by a predicate $\widehat{\alpha}$. Then, let $\mathcal{S}_A = (\mathcal{V}_A, \mathcal{T}_A, \theta_A)$ be the finite abstract system obtained by the method introduced in Chapter 5.

See Example 6.13 for an example where the algorithm fails if one does not require P-invariance.

We show how to compute fairness conditions for $\mathcal{S}_A$ that can safely be added to the system. If this procedure yields a couple of LTL fairness conditions $\phi_1, \ldots, \phi_k$, verification of a property ψ reduces to prove that the abstract system satisfies

$$\left(\bigwedge_{1 \leq i \leq k} \phi_i \right) \Rightarrow \psi \ ,$$

which can sometimes be proven, although the abstract system does not fulfill ψ directly.

We now explain the algorithm computing fairness conditions that can safely be added to the abstract system. We call it *marking algorithm*, since it is based on marking some concrete transitions as decreasing and others as increasing transitions. Intuitively, a transition is decreasing if the ranking value for some ranking function decreases, and it is increasing if taking the transition increases the ranking value. The fairness constraints are derived from the markings and can be lifted to the abstract system.

6.2 Marking Algorithm

We use WS1S formulae to express ranking functions. Intuitively, the function value of such a predicate is the size of the set of integers fulfilling the predicate.

Definition 6.3 (Ranking predicate)
A *ranking predicate* $\chi(i, P, X_1, \cdots, X_k)$ is a predicate with i as free 1st-order variable and $P, X_1, \cdots, X_k \in \mathcal{V}$ as free 2nd-order variables.

The following set, which we call a χ-*set*, is associated with a ranking predicate $\chi(i, P, X_1, \cdots, X_k)$ and an evaluation s of the variables in $\mathcal{V}$:

$$\chi(s) \stackrel{\text{def}}{=} \{i \in s(P) \mid \chi(i, s(P), s(X_1), \ldots, s(X_k))\} \ .$$

□

Given a state s of $\mathcal{S}$, the ranking value $\zeta_\chi(s)$ associated to s by a ranking predicate χ is the cardinality of $\chi(s)$. This set $\chi(s)$ is always finite, since it is a subset of the finite set $s(P)$, and the value of P is not changed by computation steps. Therefore, such a χ-set cannot decrease infinitely often.

In fact, decreasing and increasing of the cardinality is not expressible in WS1S. Therefore, we use a *stronger* condition: given a pre-state s and a

post-state s', we can observe that $\chi(s') \subsetneq \chi(s)$ implies $\zeta_\chi(s') < \zeta_\chi(s)$. In some sense, using the condition $\chi(s') \subsetneq \chi(s)$ makes the decrease of ζ_χ only partially observable. Nevertheless, if a transition is decreasing due to the stronger condition, it is also decreasing due to the original condition. Instead of expressing that the ranking value increases we use the *weaker* condition: $\zeta_\chi(s') > \zeta_\chi(s)$ implies $\chi(s') \not\subseteq \chi(s)$. Hence, if we use this weaker condition, every transition marked as increasing by the original condition $\zeta_\chi(s') < \zeta_\chi(s)$ is also marked by our condition.

The marking algorithm we present labels some of the concrete transitions with the symbols $+_\chi$ and $-_\chi$. Intuitively, a transition τ is labeled by $-_\chi$, if it is guaranteed that τ decreases the ranking value, i.e., $(s, s') \in \tau$ implies $\zeta(s) > \zeta(s')$. The label $+_\chi$ denotes that the τ *potentially increases* the ranking value. Otherwise, the transition is not marked.

The following definition defines predicates expressing that the ranking value decreases resp. potentially increases.

Definition 6.4
Let χ be a ranking predicate. We define the following WS1S predicates

$$\begin{aligned}\Delta_-(\chi, \tau) \stackrel{\text{def}}{=} \forall \mathcal{V}, \mathcal{V}' : \rho_\tau(\mathcal{V}, \mathcal{V}') \Rightarrow \\ ((\forall i : (i \in P \wedge \chi') \Rightarrow \chi) \wedge (\exists i : i \in P \wedge \chi \wedge \neg\chi'))\end{aligned} \tag{6.1}$$

and

$$\Delta_+(\chi, \tau) \stackrel{\text{def}}{=} \exists \mathcal{V}, \mathcal{V}' : \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \exists i : i \in P \wedge \chi' \wedge \neg\chi \ . \tag{6.2}$$

Here, χ' is the predicate that results in substituting all $\mathcal{V}$ variables by their primed counterparts in $\mathcal{V}'$. □

Formula (6.1) expresses that whenever the concrete transition τ is taken, the set described by χ decreases; more precisely: the χ-set induced by the post-state is a strict subset of the pre-state χ-set. Thus, the ranking values decreases. Hence, it describes the following property:

$$\Delta_-(\chi, \tau) = \forall \mathcal{V}, \mathcal{V}' : \rho_\tau(\mathcal{V}, \mathcal{V}') \Rightarrow \{i \mid \chi'(i)\} \subsetneq \{i \mid \chi(i)\} \ .$$

The second formula (6.2) expresses that there are cases in which the concrete transition cannot ensure that the post-state χ-set is a subset of the pre-state χ-set. Consequently, there is a chance that the cardinality of the χ-set increases. The formula $\exists i : i \in P \wedge \chi' \wedge \neg\chi$ expresses that the χ'-set is not a subset of the χ-set:

$$\Delta_+(\chi, \tau) = \exists \mathcal{V}, \mathcal{V}' : \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \{i \mid \chi'(i)\} \not\subseteq \{i \mid \chi(i)\} \ .$$

Table 6.1: Marking algorithm

Input: P-invariant WS1S system $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$, set of ranking predicates $\chi(i, P, X_1, \cdots, X_k)$ over $\mathcal{V}$.
Output: Labeling of $\mathcal{T}$.
Description: For each $\chi(i, P, X_1, \cdots, X_k)$, for each transition $\tau \in \mathcal{T}$: Mark τ with $-_\chi$, if $\Delta_-(\chi, \tau)$ is valid. Mark τ with $+_\chi$, if $\Delta_+(\chi, \tau)$ is valid.

The marking algorithm we propose is given in Table 6.1.

Intuitively, since each χ-set is finite, it is safe to add the fairness constraint that a transition labeled with $-_\chi$ can only be taken infinitely often, if also one of the transitions labeled with $+_\chi$ is taken infinitely often. To express this we define the following transition sets.

Definition 6.5
Let $\mathcal{S}$ be a WS1S system labeled by the marking algorithm. We define for each ranking predicate χ:

$$D_\chi \stackrel{\text{def}}{=} \{\tau \in \mathcal{T} \mid \tau \text{ is marked with } -_\chi\}$$
$$I_\chi \stackrel{\text{def}}{=} \{\tau \in \mathcal{T} \mid \tau \text{ is marked with } +_\chi\}$$

We use the same sets also for the abstract transitions, since it is always clear which set of transitions is meant. □

Then, the idea is to add for each such χ the fairness condition (D_χ, I_χ) *to the abstract system* which states that a transition $\tau_A \in D_\chi$ can only be taken infinitely often if one of the transitions in I_χ is taken infinitely often. This condition is certainly satisfied for the concrete system, hence, abstract traces violating this condition have not concrete counterpart. To be able to express the fairness condition the WS1S systems have to be extended.

6.3 Augmented Systems

To express the fairness conditions, one must be able to reason about the transition taken in the last computation step. Therefore, we modify the systems such that the transition of the last step is observable. This is done by

introducing a variable for each transition set to true whenever the transition is taken.

Definition 6.6 (Augmented system)
For a WS1S system $\mathcal{S}$, define the *augmented system* $\mathcal{A}(\mathcal{S}) = (\bar{\mathcal{V}}, \bar{\rho}, \bar{\theta})$ by

$$\bar{\mathcal{V}} \stackrel{\text{def}}{=} \mathcal{V} \cup \{tk_\tau \mid \tau \in \mathcal{T}\}$$
$$\bar{\rho} \stackrel{\text{def}}{=} \bigvee_{\tau \in \mathcal{T}} \left(\rho_\tau \wedge tk'_\tau \wedge \bigwedge_{\tau' \neq \tau} \neg tk'_{\tau'} \right)$$
$$\bar{\theta} \stackrel{\text{def}}{=} \theta \wedge \bigwedge_{\tau \in \mathcal{T}} \neg tk_\tau$$

where we assume that all tk_τ are fresh boolean variables.

For an abstract system $\mathcal{S}_A = (\mathcal{V}_A, \mathcal{T}_A, \theta_A)$ we define $\mathcal{A}(\mathcal{S}_A)$ with the same variable set as before and

$$s \in \bar{\theta}_A \text{ iff } s|_{\mathcal{V}_A} \in \theta_A \wedge \neg s(tk_\tau) \text{ for all } \tau \in \mathcal{T}_A$$
$$(s, s') \in \bar{\tau} \text{ iff } (s|_{\mathcal{V}_A}, s'|_{\mathcal{V}_A}) \in \tau \wedge (s'(tk_{\tau'}) \Leftrightarrow \tau' = \tau)$$

hold for all states s, s' over $\bar{\mathcal{V}}$ and transitions $\tau \in \mathcal{T}$. □

It is easy to see that augmentation does not change the system behavior.

Proposition 6.7
Let ϕ be a property over $\mathcal{V}$. Then, $\mathcal{S}$ satisfies ϕ iff $\mathcal{A}(\mathcal{S})$ satisfies ϕ.

Having the augmentation at hand, we are able to express formally the fairness conditions derived by the marking algorithm.

Proposition 6.8
Let χ be a ranking predicate such that $D_\chi \neq \emptyset$, and $\tau \in D_\chi$. Then,

$$\mathcal{A}(\mathcal{S}) \models (\Box\Diamond tk_\tau) \Rightarrow (\Box\Diamond(\bigvee_{\tau' \in I_\chi} tk_{\tau'})) \ .$$

Definition 6.9 (Derived fairness constraint)
Let χ be a ranking predicate such that $D_\chi \neq \emptyset$, and $\tau \in D_\chi$. The formula $(\Box\Diamond tk_\tau) \Rightarrow (\Box\Diamond(\bigvee_{\tau' \in I_\chi} tk_{\tau'}))$ is called a *fairness constraint* derived by the marking algorithm. □

We have to transfer this result to the abstract system to improve liveness verification. In principle, there are two ways to do so: one can either augment the abstract system, so construct $\mathcal{A}(\mathcal{S}_A)$, or one can compute the abstraction of $\mathcal{A}(\mathcal{S})$. The result is equivalent, if one uses the straightforward abstraction relation that observes each tk_τ variable using one abstract variable $a\_tk_\tau \equiv tk_\tau$:

Proposition 6.10
Let α be based on a set of abstraction predicates, and assume that for each $\tau \in \mathcal{T}$ there is an abstraction predicate $a\_tk_\tau \equiv tk_\tau$. Then, $\mathcal{A}(\mathcal{S}_A)$ is identical to the system $\mathcal{A}(\mathcal{S})_A$ that is constructed from α and $\mathcal{A}(\mathcal{S})$ (modulo renaming of a_tk_τ in tk_τ).

We prefer to augment the abstract system, since otherwise the construction of the abstract system is more complex, whereas the augmentation of the abstraction can be made syntactically.

Theorem 6.11
Let ϕ be a fairness condition derived by the marking algorithm. $\mathcal{A}(\mathcal{S}_A) \models \phi \Rightarrow \psi$ implies $\mathcal{S} \models \psi$.

Proof: Let ϕ_A be the ϕ formula where all tk_τ variables are replaced by a_tk_τ. Then,

	$\mathcal{A}(\mathcal{S}_A) \models \phi \Rightarrow \psi_A$	
implies	$\mathcal{A}(\mathcal{S})_A \models \phi_A \Rightarrow \psi_A$	(by Proposition 6.10)
implies	$\mathcal{A}(\mathcal{S}) \models \phi \Rightarrow \psi$	(preservation result)
implies	$\mathcal{A}(\mathcal{S}) \models \psi$	(by Proposition 6.8)
implies	$\mathcal{S} \models \psi$	(by Proposition 6.7)

■

Remark 6.12 (Strengthening)
One can again use the strengthening approach to improve the results obtained by this algorithm. The idea is to first establish invariance properties of the concrete system, and add them to the system description. Intuitively, if the ranking conditions (6.1) and (6.2) are evaluated only for a *subset of pre- and post-states*, it is more likely that the results are useful. ♣

Example 6.13
This example shows that without requiring P-invariance, the marking algorithm could fail. Assume the system $\mathcal{S} = (\{X\}, \{\tau\}, \theta)$ such that $\theta \stackrel{\text{def}}{=} X = \emptyset$, and $\tau \stackrel{\text{def}}{=} \exists j.j \notin X \wedge X' = X \cup \{j\}$. Choose the ranking predicate $\chi \stackrel{\text{def}}{=} i \notin X$.

The marking algorithm would then mark τ with $-_\chi$, since $\Delta_-(\chi, \tau)$ is valid (where the P related clauses are removed), the conditions looks as follows:

$$\forall X, X' : \exists j.j \notin X \wedge X' = X \cup \{j\} \Rightarrow$$
$$((\forall i : i \notin X' \Rightarrow i \notin X) \wedge (\exists i : i \notin X \wedge i \in X'))$$

Consequently, the derived fairness condition would forbid to take τ infinitely often, but there is in fact no reason why this is not allowed.

Note that the χ-set is not finite in this case, hence, it is no WS1S set, although the implications in Δ_- are valid. The quantification in Δ_- ranges over all *finite* sets. ♣

Finding ranking predicates. First we note that for *any* ranking predicate, the fairness constraints obtained by the marking algorithm are valid, i.e., it is guaranteed that the concrete system satisfies them. Therefore, the choice of these predicates is absolutely free; in the worst case, the fairness constraints obtained are trivial or does not help in verifying the goal.

On the other hand, it is often not really complicated to find useful predicates, since one only has to look at the very local behavior of a transition, not on the global behavior of the whole system. For each transition, one only needs a set that decreases (or increases) whenever the transition is taken.

Especially for WS1S systems that were constructed by translation of some parameterized system, this is often very obvious. If a transition is a user process transition such that some process i alters its local state, this local state is given by the membership of i to the different set variables. Hence, there is always a set such that i is added to it or removed from it, so either $i \in X$ or $i \notin X$ can be used as ranking predicate for such a set variable X.

6.4 Abstractions and Fairness Constraints

We have already seen that one can augment a concrete or an abstract system to express fairness constraints derived by the marking algorithm. Example 6.1 shows that for liveness verification, usually also additional fairness assumptions of the concrete system are necessary.

To be able to use these assumptions also on the abstract level, one needs to "lift" these assumptions to the abstract system. Let us first define the notion of fairness conditions for a parameterized system. For a thorough discussion of fairness we refer to [Fra86].

Definition 6.14 (Fairness conditions)
A transition τ is called *strongly fair*, if for each process i, it is eventually taken in case it is infinitely often enabled.

A transition τ is called *weakly fair*, if for each process i, it is eventually taken in case it is continuously enabled at some time. □

Intuitively, if a transition is marked as fair, one rules out all traces of the system that do not satisfy the corresponding fairness requirement. So a

system $\mathcal{S}$ satisfies a property ϕ, if all *fair* traces satisfy ϕ.

To be able to express fairness also on the abstract level, one has first to express these conditions in WS1S systems, and secondly, one has to make it visible on the abstract level when a transition is enabled for some process i, and when a transition is taken by some process.

The notion of fairness for WS1S systems can defined by pairs of set variables (E, T). The intended meaning is that only behavior is allowed such that if $i \in E$ infinitely often (resp. continuously), then $i \in T$ infinitely often.

We introduce sets E_τ and T_τ for each transition τ of the system. These sets contain all $i \in P$ such that τ is enabled for i, resp. such that i has taken τ in the last step. Using these sets, fairness conditions of the parameterized system can be translated to WS1S systems.

Next, these assumptions have to be lifted to the abstract level. For strongly fair transitions, this can be done by introducing the abstraction predicate

$$e_\tau \equiv E_\tau \neq \emptyset$$

which observe whether the transition is enabled for *some* i. Then, for each strongly fair transition τ, it is safe to add the constraint

$$(\Box\Diamond e_\tau) \Rightarrow (\Box\Diamond tk_\tau) \tag{6.3}$$

to the abstract system, where tk_τ is introduced by augmenting as before.

Proposition 6.15
Let ϕ be the fairness constraint (6.3) for some strongly fair transition τ. $\mathcal{S}_A \models \phi \Rightarrow \psi_A$ implies $\mathcal{S} \models \psi$.

Proof: If e_τ is infinitely often true, then there must be some i that is infinitely often in E_τ, hence, in the concrete system, $i \in T_\tau$ holds infinitely often. Consequently, also tk_τ is true infinitely often, so ϕ is satisfied by $\mathcal{S}$. ■

If one uses focusing abstractions, then for both weakly and strongly fair transitions one can use abstraction predicates to observe whether a process i is contained in E_τ and T_τ. Hence, for this kind of abstraction fairness conditions can be lifted directly.

Example 6.16
We present briefly the results for Example 6.1. The following ranking predicates are used:

$$\chi_0 \equiv i \in at\_\ell_0$$
$$\chi_1 \equiv i \in at\_\ell_1$$
$$\chi_2 \equiv i \in at\_\ell_2$$

The results of the marking algorithm for these predicates are the following fairness sets

$$\begin{array}{ll} D_{\chi_0} = \{t_{01}\} & I_{\chi_0} = \{t_{20}\} \\ D_{\chi_1} = \{t_{12}\} & I_{\chi_1} = \{t_{01}\} \\ D_{\chi_2} = \{t_{20}\} & I_{\chi_2} = \{t_{12}\} \end{array}$$

Additionally, we can lift for the process p the weak fairness conditions that t_{10} is eventually taken if continuously enabled, and the strong fairness condition that t_{11} is eventually taken if infinitely often enabled.

Using all these fairness conditions enables us to prove that p eventually reaches location ℓ_2. ♣

Chapter 7

The PAX System

PAX ("parameterized systems abstracted and explored") is a tool set for the verification of WS1S systems. This class of WS1S systems can be used to model parameterized systems as explained in Section 5.6. The system to verify must be given to the tool by the set of transition predicates and the initial state condition, together with abstraction predicates that define the abstraction relation. The PAX tool uses the MONA tool [HJJ+96, KM01] to compute the models of several predicates, therefore we use the usual MONA syntax. The MONA tool can be used to decide WS1S predicates and to compute a finite automaton that represents the set of all models of a given formula. Our tool PAX can be used to

- automatically compute a finite abstract systems, given a WS1S system and an abstraction relation,
- analyze this finite abstract system by computing the set of reachable states,
- transform the computed abstract system into the input languages of existing model-checkers.

The PAX analysis allows to establish invariance properties, and is also useful if the construction of the abstract system fails for some of the transitions. Even in this case, the state exploration can be used to compute the set of reachable states, and the exploration also computes the missing abstract transitions. This is explained in detail in Section 7.8.

Figure 7.1 shows the main functionalities of PAX. Each node is basically a file that is given to PAX or computed by one of the PAX tools. The edges between them stand for PAX tools or external tools in case of the model-checkers. The dashed line from the model-checker result to the abstraction

predicates indicates user interaction. This line refers to the incremental verification approach, where information gained by analyzing counterexamples is used to define additional abstraction predicates as explained in Section 5.3.

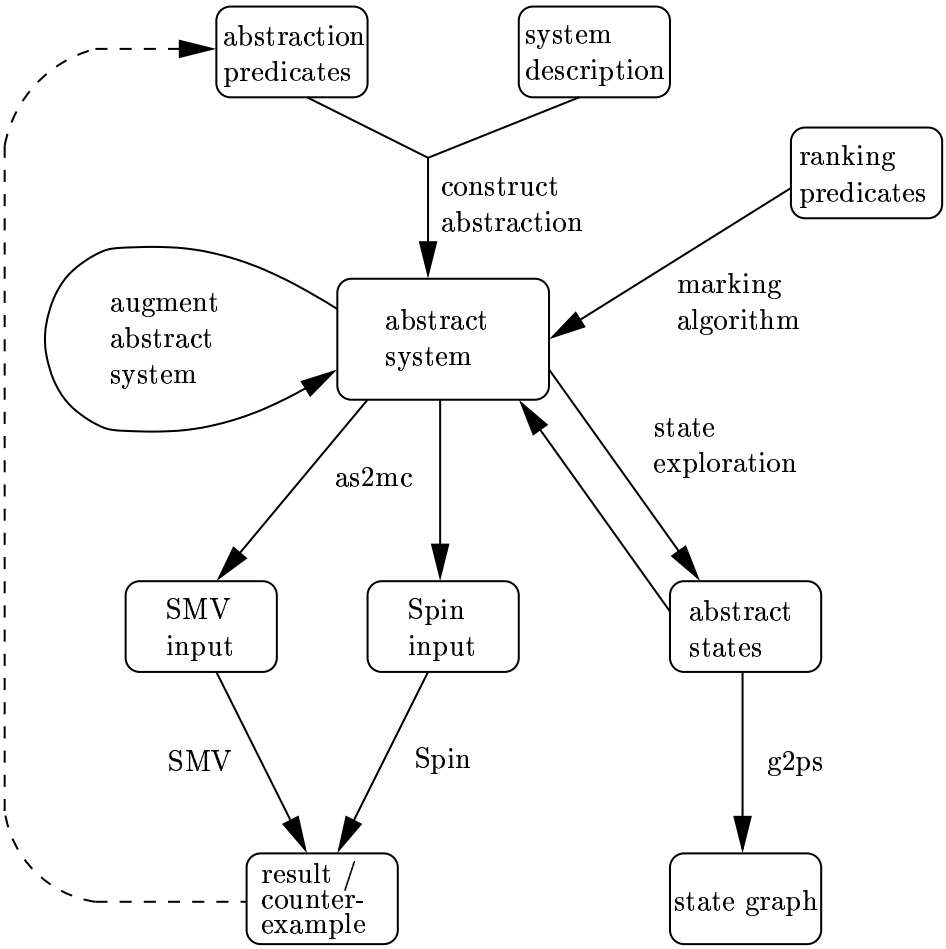

Figure 7.1: PAX Overview

7.1 System Inputs

The user has to provide the following input for the PAX system:

- The concrete system description given by an initial state predicate $\theta(\mathcal{V})$ and some transition predicates $\rho_\tau(\mathcal{V}, \mathcal{V}')$ for a number of transitions $\tau \in \mathcal{T}$. These transitions define the transition relation $\rho \stackrel{\text{def}}{=} \bigvee_{\tau \in \mathcal{T}} \rho_\tau(\mathcal{V}, \mathcal{V}')$.

- A number of abstraction predicates $\phi_v(\mathcal{V})$, for $v \in \mathcal{V}_\mathcal{A}$, that induces the abstraction relation predicate $\widehat{\alpha} \stackrel{\text{def}}{=} \bigwedge_{v \in \mathcal{V}_\mathcal{A}} (v \Leftrightarrow \phi_v)$.

The transition relation defined in this way corresponds more to an asynchronous execution model. But also synchronous execution can be modeled by giving only one transition predicate describing the steps of all processes.

When using the marking algorithm, one has to give additionally some ranking predicates as input.

7.2 Construct Abstractions using MONA

Recall from Section 5.7 that

$$\theta_A(\mathcal{V}_\mathcal{A}) \stackrel{\text{def}}{=} \exists \mathcal{V} : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \ . \tag{7.1}$$

is a predicate describing the abstract initial state, with free variables $\mathcal{V}_\mathcal{A}$; the abstract counterpart for each concrete transition τ is characterized by

$$\rho_{\tau_A}(\mathcal{V}_\mathcal{A}, \mathcal{V}'_\mathcal{A}) \stackrel{\text{def}}{=} \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A}) \ , \tag{7.2}$$

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

Exactly these predicates can be constructed by PAX from its inputs, and given to MONA to compute the set of their models.

MONA [HJJ+96, KM01] is an implementation of decision procedures for WS1S and WS2S, the weak monadic second-order theories of one and two successors. It has been developed at BRICS since 1994, and has matured into an efficient and popular tool. Through the years, many different approaches to improve the efficiency have been tried out and a number of implementation "secrets" [KMS02] have been discovered and integrated in MONA.

Especially, MONA has provided the foundation of or has been integrated into a range of tools as is the case for our PAX tool. For a list of examples we refer to [KMS02].

The connection to PAX is the possibility of MONA to return the *automaton* that represents the set of all models. We are only interested in the abstract *boolean* variables, since we want to compute the abstract system. From the automaton description, one can easily extract these values.

The abstract system can be represented by two lists, one list of initial states of the following form:

```
Variables: a_p1_la a_p1_lb a_p2_la a_p2_lb a_p3_la ...
Initial states:
000000001
000000101
X0000101
```

The first line shows the order of the abstract variables, and an abstract state is given by the list of values of these variables, where `X` means that both `0` and `1` are possible (so such a state is again an abbreviation for a set of abstract states). The transitions are given by the list:

```
Variables: a_p1_la a_p1_lb ... a_p1_la' a_p1_lb' ...
Abstract transitions:
t12 : 000000000 -> 0000X000X
t12 : 000000000 -> 00100000X
t12 : 000000000 -> 10000000X
...
t23 : 000000000 -> 000000000
t23 : 000000001 -> 000000001
t23 : 000000011 -> 000000011
...
```

Each line describing a transition begins with the transition name, that is the concrete transition from which the abstract one is derived, followed by abstract pre- and post-states.

7.3 Augmenting the Abstract System

Augmenting the abstract system simply means to extend the set of abstract variables and states by variables tk_τ for each transition, variables that are set to tt whenever the last step was a τ step. As explained in Section 6.3, this is needed to express fairness constraints derived by the marking algorithm.

Given the list of initial states and abstract transitions, this can easily be done syntactically by appending these variables V to the variable lists, extending each initial state with value "0" for each variable $v \in |V|$, and changing each line in the transition list

```
trans: <pre> -> <post>
```

into

```
trans: <pre>X...X -> <post>0..010..0
```

where $|V|$ is the number of `X`'s, and the `1` in the post-state corresponds to the position of tk_τ in the V list.

7.4 Marking Algorithm

The marking algorithm presented in Section 6.2 can be used to derive fairness constraints which are guaranteed to be satisfied by the concrete system. Intuitively, the constraints rule out some abstract behavior that has no concrete counterpart. These conditions can be used as additional premises for verification of liveness properties on the abstract level. Section 6.2 explains the algorithm and the derived constraints.

The algorithm is based on additional input: some ranking predicates must be provided by the user. Intuitively, each predicates describes a set, e.g., a set of processes if a parameterized system is modeled as a WS1S system. For each ranking predicate and transition, the algorithm proves whether the transitions always reduce the set described by the ranking predicate, or if the set may increase for some states. If the transition decreases the set, the transition can only be taken infinitely often, if also some increasing transition is taken infinitely often, since all sets of a WS1S systems are finite.

This analysis is done for the concrete system. Since the abstract system corresponds closely to the concrete one, i.e., each abstract transition corresponds to one concrete transition, the computed fairness constraints can be lifted to the abstract system.

To be able to express them as premises, one needs to augment the abstract system with tk_τ variables that as explained before. These variables exhibit which transition was taken in the last system step.

7.5 Model-checking

The PAX tool can translate the constructed abstract systems to two well-known model-checkers.

One of them is SMV (Symbolic Model Verifier) [McM92, BCM+92], the first symbolic model-checker using a BDD representation [Bry85, Bry86, Bry92] of the state space and the systems transition relation. This model-checker does not support LTL, which we choose as our specification language. Instead it uses the branching time logic CTL. But two re-implementations of this model-checker, called NuSMV [CCGR99, CCG+02] resp. CadenceSMV [McM99a, McM99b] allow also LTL specifications.

A program in the SMV language is a set of modules, but we need to use the main module only. A module consists of variable declarations, variable initializations, and a description of the transition relation. The transition relation can be described using predicates over the current and the next state of the module's variables, the expression `next(v)` refers to the transition's

post-state for each module variable `v`.

There are two ways to define the transition relation, either by using an imperative style

```
ASSIGN
  next(v) := expr
```

for each variable `v`, or in an implicit style using a predicate over pre- and post-state variables:

```
TRANS
  <pred>
```

All valuations of the free variables that satisfy the given predicate define the transition relation. One can also define several `TRANS` predicates; in this case the transition relation corresponds to the conjunction of all these predicates. Hence, the resulting model has a synchronous semantics, fitting to hardware verification for which SMV was originally designed.

For our translation we only use the main module and the implicit style for describing a transition relation. An asynchronous execution model can be defined by giving one single `TRANS` predicate that is the disjunction of all transition predicates derived from abstract transitions.

The `as2mc` translator creates input for these model-checkers by translating the abstract system. We illustrate this translation for the SMV language. Suppose the system is given by lists of states and transitions as follows:

```
Initial states:
000000001

Abstract transitions:
t12 : 000000000 -> 0000X000X
t12 : 000000000 -> 00100000X
```

Then, the translation to the SMV language is straightforward:

```
MODULE main
VAR
  a_p1_la : boolean;
  a_p1_lb : boolean;
  a_p2_la : boolean;
  ...

INIT
```

```
   ( ! a_p1_la & ! a_p1_lb & ! a_p2_la & ! a_p2_lb &
     ! a_p3_la & ! a_p3_lb & ! a_s_eq0 & ! a_s_leq1 & a_s_leq2 )

TRANS
   ( ! a_p1_la & ! a_p1_lb & ! a_p2_la & ! a_p2_lb &
     ! a_p3_la & ! a_p3_lb & ! a_s_eq0 & ! a_s_leq1 &
     ! a_s_leq2 & ! next(a_p1_la) &
     ! next(a_p1_lb) & ! next(a_p2_la) & ! next(a_p2_lb) &
     ! next(a_p3_lb) & ! next(a_s_eq0) & ! next(a_s_leq1) )
 | ( ! a_p1_la & ! a_p1_lb & ! a_p2_la & ! a_p2_lb &
     ! a_p3_la & ! a_p3_lb & ! a_s_eq0 & ! a_s_leq1 & ! a_s_leq2 &
     ! next(a_p1_la) & ! next(a_p1_lb) & next(a_p2_la) &
     ! next(a_p2_lb) & ! next(a_p3_la) & ! next(a_p3_lb) &
     ! next(a_s_eq0) & ! next(a_s_leq1) )
 | ...
```

In the verification of parameterized systems, the bottleneck is usually the computation of the abstract system. The model-checking of the result is in most cases quite fast and unproblematic.

The second model-checker we use is the Spin [Hol91] model-checker, an enumerative model-checker which is very well suited for protocol verification. The translation to Promela, the C-like input language of Spin, is similar. Instead of a predicate describing the transition relation as in the SMV case, in Promela one can use a non-deterministic choice command to execute one of the abstract transitions, and the state change is modeled by assignments to the abstract variables.

The bottleneck of the verification using the PAX tool is in nearly all cases the construction of the abstract system. Therefore, in practice, model-checking always succeeds, i.e., the model-checker does not run "out of time" or "out of space". Only if a lot of additional fairness constraints are computed using the marking algorithm, it takes sometimes substantial time, since these constraints are then additional premises to the property one likes to prove.

If model-checking shows that the property is not valid, the model-checker returns a counterexample. Analyzing this counterexample there is either a concrete counterpart of this trace, showing that the property is indeed not valid for the concrete system, or there is no such concrete counterpart. In the latter case, the analysis can help to improve the abstract system. In practice this means that the user has to modify or to add some abstraction predicates. Computing the abstract system corresponding to this refined abstraction relation results in a more precise abstraction. This is indicated

in Figure 7.1 by the dashed line from result/counterexample to abstraction predicates.

7.6 State Exploration

There is also a built-in simple forward state exploration in PAX. This program can be used to compute the set of reachable states. Thus, it can be used to establish invariance properties by checking whether there are "bad" states reachable.

This is something every model-checker can do. Nevertheless, this exploration is useful if PAX cannot construct some of the abstract transitions since memory is exceeded. In this case, the PAX state explorer has some features that allow to proceed. We discuss this feature in detail in Section 7.8.

As a side effect, this program produces a *state graph* of the reachable states, which can be visualized if the abstract state space is reasonably small.

7.7 Visualizing the State Graph

The program `g2ps` produces a *state graph* of the abstract system. The program basically translates the state graph produced by the state explorer into the input format of the `dot` program, which is part of the `Graphviz` tool set. This tool set is a package of graph drawing programs from AT&T and Lucent Bell Labs. It can be obtained together with documentation at `http://www.research.att.com/~north/graphviz/` .

7.8 Pushing the Limits

In this section we discuss some solutions if the computing power does not suffice to construct the abstract system.

The decision procedures for WS1S logic have a non-elementary worst case complexity. Therefore, in practice one occasionally hits the limits of the machine, i.e., memory is exceeded or the computing power does not suffice to calculate the abstract system.

To make progress in this case, one can try to reduce the system to simplify the computation, improve the computation algorithms, or one can try to simplify the task the computer has to do.

System model. First, from the theoretical point of view, one can try to reduce the system size by finding redundant variables, for instance:

- If one has to model variables over a particular finite domain, one can carefully examine the set of variables that is used. Often, redundancy can be removed. Moreover, the domain of the variables should be as small as possible, such that the minimal set of boolean variables or arrays is used to encode the variable's values.

- If one does not succeed in showing the property, try to use the incremental verification approach, see Section 5.3, showing a simpler property first that can hopefully be established using a simpler abstraction relation.

Tool side. One can also try to make improvements on the the practical side. The underlying decision procedures are MONA internal, which is an efficient, robust, and, in our experience, reliable tool which has been continuously improved for a long time. Therefore, there is no sense in trying to redo all the development and to build own algorithms.

So any "improvement" made here must change the things to compute. These "things" are in our case the WS1S theories, which the PAX tool gives MONA, and for which MONA computes the automaton representing all its models. How this can help to push the limits is explained next.

7.8.1 Partial Transition Relations

This section discusses an approach that allows to make verification progress although the PAX system is not able to construct some of the abstract transitions.

We first observe the difference in quality of computing a *full* abstract transition as given in Formula (7.2) and the abstract post-state for a given pre-state $s \in \Sigma_{\mathcal{V}_\mathcal{A}}$:

$$\rho_{\tau_A} \stackrel{\text{def}}{=} Eq(\mathcal{V}_\mathcal{A} = s) \wedge \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}(\mathcal{V}', \mathcal{V}'_\mathcal{A}) \ , \quad (7.3)$$

where $Eq(\mathcal{V}_\mathcal{A} = s) \stackrel{\text{def}}{=} \bigwedge_{v \in \mathcal{V}_\mathcal{A}} v = s(v)$. Though this formula looks even more complex than Formula (7.2) at first sight, it has only half the free variables, since the $\mathcal{V}_\mathcal{A}$ variables are all substituted by *constants*, so only the $\mathcal{V}'_\mathcal{A}$ variables are left. Consequently, it is much easier to compute the set of all models for this formula.

So when PAX encounters problems during the computation of some transitions, it leaves out these transitions, computing only the remaining abstract transitions. Then, there are two possibilities to proceed.

First, one can compute now the unsuccessful transitions in the way Formula (7.3) proposes: compute for each pre-state the set of corresponding post-states and add the results to the set of system transitions. Note that although it is likely that each such computation is successful and faster than the computation of the full abstract transition, the number of computations makes this approach impractical. If one uses n abstract boolean variables, there are 2^n pre-states and computations.

A second possibility is to compute the missing transitions *on-the-fly* during a state space exploration. This often reduces the number of necessary computations significantly. Exactly this can be done by the PAX state space exploration. In this way, only *partial* abstract transitions are computed. This is sufficient, since the partial abstract transition contains all transition successors for all reachable abstract states, so the part that is omitted has no relevance.

The algorithm given in Table 7.1 can be used for this on-the-fly computation. We assume the following input given: a WS1S system $\mathcal{S} = (\mathcal{V}, \mathcal{T}, \theta)$, a (partial) abstraction $\mathcal{S} = (\mathcal{V}_\mathcal{A}, \mathcal{T}_A, \theta_A)$ (some abstract transitions may be missing), and abstraction relation predicates ϕ_v.

Table 7.1: State exploration algorithm

```
P := ∅
U := θ_A
while U ≠ ∅ do
    choose s ∈ U
    U := U \ {s} ; P := P ∪ {s}
    for each τ ∈ T do
        if τ_A ∈ T_A compute M := τ_A(s)
            U := U ∪ (M \ P)
        if τ_A ∉ T_A compute M := τ_A(s) by (7.3)
            U := U ∪ (M \ P)
            add all (s, s') to τ̃_A for s' ∈ M
    od
od
return(P)
```

The result returned by this algorithm is the set of all reachable states. As a side effect, for each missing abstract transition τ_A, the "reachable part" $\widetilde{\tau}_A$ of τ_A is computed. Hence, this algorithm "completes" the abstract system.

Proposition 7.1
Let $\mathcal{S}'_A$ be the abstract system with full abstract and also partial abstract transitions $\widetilde{\tau}_A$ derived by the algorithm. Then, $\mathcal{S}_A \models \psi$ iff $\mathcal{S}'_A \models \psi$ for all properties ψ.

7.8.2 Transition Over-Approximation

Another way to deal with the problem of insufficient computing power is to compute *less precisely*. Instead of computing the correct abstract transitions, one can also compute an over-approximation of it.

We recall that there is an abstraction predicate ϕ_v given for each $v \in \mathcal{V}_\mathcal{A}$. The idea is to compute a number of *partial post-states* of a transition for subsets $V \subseteq \mathcal{V}_\mathcal{A}$, instead of computing the "full' post-state. In other words, the post-state is computed *independently* for subsets of $\mathcal{V}_\mathcal{A}$. Such a computation for a subset of $\mathcal{V}_\mathcal{A}$ might be successful, although the "full" abstract transition is not computable, since the computation involves fewer variables. A similar idea is proposed in [BLO98].

If one does so for several subsets, such that each abstract variable $v \in \mathcal{V}_\mathcal{A}$ occurs at least in one of the subsets, one can combine the partial abstract post-states into a set of abstract post-states. This set will be an over-approximation of the precise abstract post-states.

Assume that $M \subseteq 2^{\mathcal{V}_\mathcal{A}} \setminus \emptyset$ is given with $\bigcup M = \mathcal{V}_\mathcal{A}$. The members of M are all the subsets of $\mathcal{V}_\mathcal{A}$ for which partial abstract post-states are computed.

For each $V \in M$ compute the set of all models of the formula

$$\rho_{\tau_A}(\mathcal{V}_\mathcal{A}, \mathcal{V}'_\mathcal{A}) \stackrel{\text{def}}{=} \exists \mathcal{V}, \mathcal{V}' : \widehat{\alpha}(\mathcal{V}, \mathcal{V}_\mathcal{A}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}') \wedge \widehat{\alpha}_V(\mathcal{V}', V') \ , \tag{7.4}$$

where V' contains the primed version for each variable in V and

$$\widehat{\alpha}_V \stackrel{\text{def}}{=} \bigwedge_{v \in V} (v \Leftrightarrow \phi_v) \ .$$

The result for each $V \in M$ of this computation is a *partial* abstract transition $\tau_A^V \subseteq \Sigma_A \times \Sigma_{A,V}$, where $\Sigma_{A,V} = [V \to \mathbb{V}]$.

The following property is easy to show.

Proposition 7.2
$(s, s') \in \tau_A^V$ holds iff there exists an abstract post-state $\bar{s}' \in \Sigma_A$ with $(s, \bar{s}') \in \tau_A$ and $\bar{s}_A$ and s' agree on V.

Combining these partial abstract transitions leads to an over-approximation of the precise abstract transition.

Define

$$\widetilde{\tau}_A \stackrel{\text{def}}{=} \{(s, s') \mid \forall V \in M : \exists s'_V.(s, s'_V) \in \tau_A^V \wedge s'_V = s'|_V\} \ . \tag{7.5}$$

This approximation can be computed effectively: for each state one has to combine all its post-states of the different partial transitions such that they agree on common variables.

A similar construction can also be done for θ_A. In practice, this is usually not necessary since θ_A is much easier to compute than an abstract transition. The complexity of θ_A is significantly lower since no post-state variables occur in the predicate characterizing θ_A.

The resulting system is an *abstraction* of the *abstract system.*

Proposition 7.3
Let $\mathcal{S}'_A = (\mathcal{V}_\mathcal{A}, \mathcal{T}'_A, \theta_A)$, with $\mathcal{T}'_A = \{\widetilde{\tau}_A \mid \tau \in \mathcal{T}\}$. Then, $\mathcal{S}'_A$ is an abstraction of $\mathcal{S}_A$, i.e., $\mathcal{S}_A \sqsubseteq_{id} \mathcal{S}'_A$.

Proof: Follows from Proposition 7.2 and Formula (7.5). ■

Chapter 8

Case Study: Cache-Coherence Protocol

In this chapter we present a case study proving both safety and liveness properties of a cache-coherence protocol by Steve German which firstly appeared in an SPL notation in [PRZ01]. We use our proof method based on abstraction, and show how PAX can be applied to verify certain properties of the protocol.

8.1 Protocol Description

We give the protocol description in a slightly different notation than that in [PRZ01] using a guarded command language. Moreover, compared to [PRZ01], we consider some of the client transitions as home (controller) transitions, since these transitions modify only home variables.

The protocol consists of a central controlling component, called *home*, and a parameterized number of *client* processes. Messages are sent via three channels from *home* to a *client* c and vice versa:

- `chan1`$[c]$: The client sends requests for *shared* or *exclusive* access to the cache line to home via this channel.

- `chan2`$[c]$: Used by the home process to send *grants* to client c or the *invalidate* command enforcing the client to invalidate its cache status.

- `chan3`$[c]$: The client c uses this channel to send acknowledgments about invalidating its cache status to home.

8.1.1 Clients

The data structure of client processes is rather simple, each client is equipped with a variable `cache` that holds the actual state of its cache line. Possible values for this variable are `invalid`, `shared`, and `exclusive`. Hence, there is one `cache` array over these values.

8.1.2 Controlling Component

The home process, i.e., the controller, uses far more data structures. These are:

- `command` and `current_client` that hold the current job and client it has to process. If home receives a request from a client process c, then this request will be stored in `command` and c in `current_client` until the request is processed. So, `current_client` is a numeric variable, and `command` ranges over the `empty` value and the both requests for exclusive and shared access.
- A boolean variable `excl_granted` which is set to tt whenever an exclusive grant was given to a client.
- A boolean array `sharer_list` that stores all processes to which some grant has been given.
- A boolean array `invalidate_list` that is used during the invalidation process. If the home process has to invalidate some clients, e.g., because they have shared access and some other process requests for exclusive access, then all processes which must be invalidated are stored in `invalidate_list`.

8.1.3 Protocol Transitions

The transitions of the *home* process and one *client* c are given in Table 8.1 and Table 8.2 in a guarded command style language.

The next example illustrates the way how transitions given in the guarded command like style are translated to transitions of a WS1S system.

Example 8.1
We give transition c_1 as example. Since there are three different values for `cache`$[c]$, and we can only translate boolean arrays, we have to do some encoding. Formally, we encode the array using two boolean arrays; these arrays are then translated to sets of a WS1S system.

Table 8.1: Transitions of the *home* process

h_0 :	$(\texttt{command} = \texttt{req\_shared} \wedge \neg\texttt{excl\_granted}$ $\wedge\ \texttt{chan2}[\texttt{current\_client}] = \texttt{empty})$ $\rightarrow\ \texttt{sharer\_list}[\texttt{current\_client}] := tt\ ;\ \texttt{command} := \texttt{empty}\ ;$ $\texttt{chan2}[\texttt{current\_client}] := \texttt{grant\_shared}$
h_1 :	$(\texttt{command} = \texttt{req\_exclusive} \wedge \texttt{chan2}[\texttt{current\_client}] = \texttt{empty}$ $\wedge\ \forall i : [1 \ldots N].\neg\texttt{sharer\_list}[i])$ $\rightarrow\ \texttt{sharer\_list}[\texttt{current\_client}] := tt\ ;\ \texttt{command} := \texttt{empty}\ ;$ $\texttt{chan2}[\texttt{current\_client}] := \texttt{grant\_exclusive}\ ;$ $\texttt{excl\_granted} := tt$
h_2 :	$(\texttt{command} = \texttt{empty} \wedge \texttt{chan1}[c] \neq \texttt{empty})$ $\rightarrow\ \texttt{command} := \texttt{chan1}[c]\ ;\ \texttt{chan1}[c] := \texttt{empty}\ ;$ $\texttt{invalidate\_list} := \texttt{sharer\_list}\ ;\ \texttt{current\_client} := c$
h_3 :	$(((\texttt{command} = \texttt{req\_shared} \wedge \texttt{excl\_granted})$ $\vee\ \texttt{command} = \texttt{req\_exclusive})$ $\wedge\ \texttt{invalidate\_list}[c] \wedge \texttt{chan2}[c] = \texttt{empty})$ $\rightarrow\ \texttt{chan2}[c] := \texttt{invalidate}\ ;\ \texttt{invalidate\_list}[c] := ff$
h_4 :	$(\texttt{command} \neq \texttt{empty} \wedge \texttt{chan3}[c] = \texttt{invalidate\_ack})$ $\rightarrow\ \texttt{sharer\_list}[c] := ff\ ;\ \texttt{excl\_granted} := ff\ ;$ $\texttt{chan3}[c] := \texttt{empty}$

Table 8.2: Transitions of a *client* c

c_0 :	$\texttt{skip}$
c_1 :	$(\texttt{cache}[c] = \texttt{invalid} \wedge \texttt{chan1}[c] = \texttt{empty})$ $\rightarrow\ \texttt{chan1}[c] := \texttt{req\_shared}$
c_2 :	$((\texttt{cache}[c] = \texttt{invalid} \vee \texttt{cache}[c] = \texttt{shared}) \wedge \texttt{chan1}[c] = \texttt{empty})$ $\rightarrow\ \texttt{chan1}[c] := \texttt{req\_exclusive}$
c_3 :	$(\texttt{chan2}[c] = \texttt{invalidate} \wedge \texttt{chan3}[c] = \texttt{empty})$ $\rightarrow\ \texttt{chan2}[c] := \texttt{empty}\ ;\ \texttt{chan3}[c] := \texttt{invalidate\_ack}\ ;$ $\texttt{cache}[c] := \texttt{invalid}$
c_4 :	$\texttt{chan2}[c] = \texttt{grant\_shared}$ $\rightarrow\ \texttt{cache}[c] := \texttt{shared}\ ;\ \texttt{chan2}[c] := \texttt{empty}$
c_5 :	$\texttt{chan2}[c] = \texttt{grant\_exclusive}$ $\rightarrow\ \texttt{cache}[c] := \texttt{exclusive}\ ;\ \texttt{chan2}[c] := \texttt{empty}$

So instead of the `cache` array we have two sets `cache_a` and `cache_b`. For example, $c \notin$ `cache_a` $\cup$ `cache_b` corresponds to `cache`$[c]$ = `invalid`. The other values for the channels are encoded similarly, as well as the other array variables.

```
## transition guard: cache[c] = invalid, chan1[c] = empty
c notin cache_a union cache_b & c notin chan1_a union chan1_b

## transition assignment: chan1[c] := req_shared
& chan1_a' = chan1_a union {c} & chan1_b' = chan1_b \ {c}

## the other variables are not modified
& (excl_granted' <=> excl_granted)
& (curr_cmd_a' <=> curr_cmd_a)
& (curr_cmd_b' <=> curr_cmd_b)
& curr_client' = curr_client
& chan2_a' = chan2_a & chan2_b' = chan2_b
& chan3' = chan3
& cache_a' = cache_a & cache_b' = cache_b
& sharer_list' = sharer_list
& invalidate_list' = invalidate_list
```

8.1.4 Properties of Interest

The protocol should ensure *coherence* between the clients, that is, whenever there is a client in `exclusive` state, then all the other clients are in state `invalid`.

The second kind of properties we are interested in are liveness (response) properties, namely, that requests of a process will be eventually granted. These liveness properties are only valid under further fairness assumptions, e.g., that the home process will eventually read the channel content of each process.

We start with verifying the coherence property, which is a safety property, more precisely, an invariance property.

8.2 Coherence Property

We apply an incremental verification process to verify the coherence property that whenever there is a process in exclusive mode, then there is no other process having any access. Successively, we prove invariants of the system,

which are then used to strengthen the system to establish further invariants, as explained in Section 5.3.

8.2.1 Step 1

In a first step, we want to show that the property

$$\begin{array}{l}\texttt{excl\_granted} \Rightarrow \\ (\forall i : \neg\texttt{sharer\_list}[i] \\ \quad \vee (\exists i : \texttt{sharer\_list}[i] \wedge \forall j : j \neq i \Rightarrow \neg\texttt{sharer\_list}[j]))\end{array} \tag{8.1}$$

which states that whenever variable `excl_granted` is set, then there is at most one process in `sharer_list`, is an invariant for the protocol.

It turned out that this property is an *inductive invariant* of the system. Hence, it is true for initial states and preserved by every system step.

To prove such inductive invariants, one can use the simplest abstraction relation, with only one abstraction predicate, let us say ϕ_1, that is given by Formula (8.1). So, the abstraction relation used is

$$\texttt{a\_inv1} \Leftrightarrow \phi_1 \ ,$$

expressing that the (only) abstract variable `a_inv1` is true whenever (8.1) holds on the concrete level.

Example 8.2
Translated to the PAX/MONA input language, the abstraction relation looks as follows:

```
## abstract variable a_inv1:
(a_inv1 <=>
  (exclusive_granted =>
    (sharer_list = empty | ex1 i: {i} = sharer_list)))
```

Computing the abstract system then proves the invariance of this property nearly directly. The abstract system has only two possible states, one for each boolean value of the abstract variable for ϕ_1, `a_inv1`. Moreover, the only initial state is the one with `a_inv1` $= tt$, and every abstract transition starting in the state `a_inv1` $= tt$ leads to same post-state.

8.2.2 Step 2

Invariant (8.1) can now be used to strengthen the concrete system. This is done by simply adding the invariant to the system description, ruling out any concrete states not fulfilling the invariant. This enables us to establish simultaneously seven new invariants.

Together, these properties further determine the behavior of the system. We first give some intuitions about them.

- Formula (8.2) and (8.3) state that during the invalidation process, processes are not in `invalidate_list`.
- Formula (8.4) describes that the home process does not give grants to a client which it wants to invalidate.
- Formula (8.5) states that the `sharer_list` contains at least all processes which have shared or exclusive access, or have such a grant in their input channel, together with all processes which are not fully invalidated.
- Formula (8.6) specifies that no process is invalidated without a request that enforces invalidation.
- Formula (8.7) is similar to (8.1), and specifies the same property for `invalidate_list` instead of `sharer_list`.
- The last formula states that every `invalidate_ack` received by the home process is correct because the corresponding cache is indeed invalid.

Now these properties are formalized.

$$\forall i : \neg(\texttt{invalidate\_list}[i] \wedge \texttt{chan2}[i] = \texttt{invalidate}) \tag{8.2}$$

$$\forall i : \neg(\texttt{invalidate\_list}[i] \wedge \texttt{chan3}[i] = \texttt{invalidate\_ack}) \tag{8.3}$$

$$\forall i : \neg(\texttt{chan2}[i] \neq \texttt{empty} \wedge \texttt{chan3}[i] = \texttt{invalidate\_ack}) \tag{8.4}$$

$$\begin{aligned} &\forall i : (\texttt{invalidate\_list}[i] \vee \texttt{chan2}[i] \neq \texttt{empty} \\ &\vee \texttt{chan3}[i] = \texttt{invalidate\_ack} \vee \texttt{cache}[i] \neq \texttt{invalid}) \\ &\Rightarrow \texttt{sharer\_list}[i] \end{aligned} \tag{8.5}$$

$$
\begin{array}{l}
(\exists i : \texttt{chan2}[i] = \texttt{invalidate} \vee \texttt{chan3}[i] = \texttt{invalidate\_ack}) \Rightarrow \\
(\exists i : \texttt{sharer\_list}[i] \wedge \\
((\texttt{command} = \texttt{req\_shared} \wedge \texttt{excl\_granted}) \\
\vee \texttt{command} = \texttt{req\_exclusive}))
\end{array}
\tag{8.6}
$$

$$
\begin{array}{l}
\texttt{excl\_granted} \Rightarrow \\
(\forall i : \neg\texttt{invalidate\_list}[i] \\
\vee \exists i : \texttt{invalidate\_list}[i] \\
\wedge (\forall j : j \neq i \Rightarrow \neg\texttt{invalidate\_list}[j]))
\end{array}
\tag{8.7}
$$

$$
\forall i : \neg(\texttt{chan3}[i] = \texttt{invalidate\_ack} \wedge \texttt{cache}[i] \neq \texttt{invalid}) \tag{8.8}
$$

In order to verify the property using our approach, we have to define an abstraction relation to be able to compute an abstract system. We use an abstraction relation, which is built up by a set of abstraction predicates ϕ_i as described in Section 5.7. Each of the following items corresponds to one (or more, also depending on the encoding of the protocol into WS1S logic) of these abstraction predicates ϕ_i:

- One abstract boolean variable for the truth value of each of Formulae (8.2)-(8.8). This gives seven abstract variables a_2, ... , a_8.
- Two abstract boolean variables a_command_a and a_command_b that encode the values of the home process variable command.
- One abstract variable a_excl_granted for the value of the controller variable excl_granted.

PAX is able to automatically compute the abstraction, given these abstraction predicates and the concrete system description.

The resulting finite abstract system can then be translated by the PAX tool to the input language of some model-checker. The model-checker can easily establish the invariants, since the abstract system is quite small, having only a couple of boolean variables.

Counterexamples. Most of these invariants were found during the verification process by examining *counterexamples*. Whenever the abstraction was too weak to show the properties so far, the analysis of the counterexample led to a new property which was violated by the example, but seemed to be an invariant of the system. Therefore, we added a new formula ϕ_i to the abstraction relation and recomputed the abstraction. Since the construction of

the abstract system is fully automatically, there is very little user interaction necessary for the reconstruction.

Nevertheless, human ingenuity is needed to analyze the counterexample and find the new "missing" property. See also Section 5.3.

8.2.3 Step 3

In a third step we strengthen the system with all properties verified so far. We are now able to prove simultaneously two more invariance properties, the second being the coherence property.

We want to show invariance of the following properties *for each arbitrary process* p. The first one specifies that the home process is always aware of processes having exclusive access, the second one is the coherence property.

$$\texttt{cache}[p] = \texttt{exclusive} \Rightarrow \texttt{excl\_granted} \tag{8.9}$$

$$\texttt{cache}[p] = \texttt{exclusive} \Rightarrow (\forall j : j \neq i \Rightarrow \texttt{cache} \neq \texttt{shared}) \tag{8.10}$$

Since these properties are *universal properties* as explained in Section 5.8.3, we now use an abstraction relation which *focuses* on one arbitrary but fixed process p. This allows us to generalize the property to an invariant for all processes. The abstraction is based on abstraction predicates ϕ_i describing

- the truth value of the Formula (8.9) and (8.10),
- the value of the home variable `excl_granted`,
- the content of p's input channel `chan2`$[p]$,
- whether p is in the sharer list (`sharer_list`$[p]$), and
- p's cache status.

Computing the abstract system corresponding to this abstraction relation, it can easily be used to establish the Properties (8.9) and (8.10).

8.3 Liveness: Exclusive Access Response

Our goal is to verify that for each process p, whenever p requests for an exclusive access, then this access will eventually be granted by the home process. This is again a *universal property*, so we use again an abstraction relation which focuses on one arbitrary but fixed process p, built on abstraction predicates observing all information relevant for that process. So we introduce abstract variables for

- `cache`$[p]$,
- `chan1`$[p]$-`chan3`$[p]$,
- whether `sharer_list`$[p] = tt$,
- whether $p =$ `current_client`,
- the values of the home process variables `command` and `excl_granted`.

Moreover, we add to the abstract system boolean variables $taken_\tau$ encoding which transition was taken in the last step, as described in Section 6.3.

As presented in Section 6.2 we use the following ranking predicates:

$$\chi_1(i) \stackrel{\text{def}}{=} \texttt{invalidate\_list}[i]$$
$$\chi_2(i) \stackrel{\text{def}}{=} \texttt{sharer\_list}[i]$$
$$\chi_3(i) \stackrel{\text{def}}{=} \texttt{chan1}[i] = \texttt{empty}$$
$$\chi_4(i) \stackrel{\text{def}}{=} \texttt{chan2}[i] \neq \texttt{empty}$$
$$\chi_5(i) \stackrel{\text{def}}{=} \texttt{chan3}[i] = \texttt{empty}$$

Our tool computes for each transition τ and predicate χ_j, whether τ will definitely decrease the set of processes i for which $\chi_j(i)$ holds, or potentially increase this set, as explained in Chapter 6. Thus, for each predicate χ_j we build two sets of transitions D_j and I_j. Since all the sets appearing in the formulae describing the system are finite for each instance of the parameterized network, no such instance can have a computation containing infinitely many decreasing transitions from D_j and only finitely many transitions from I_j.

These fairness constraints can be added as a new premises to the liveness property that is to prove.

Choosing ranking predicates. These predicates were easily found, since it is sufficient to examine each transition locally, searching for predicates that decrease for this transition without knowing the behavior of the overall system! This can easily be done, because all the behavior of the system is encoded in the manipulation of sets, and usually it is fairly easy to find one of these sets M that decreases (so one can choose the ranking predicate $\chi(i) \stackrel{\text{def}}{=} i \in M$) or which increases (one can use $\chi(i) \stackrel{\text{def}}{=} i \in P \wedge i \notin M$, where P is a set containing all process indices) when the transition is taken.

In principle, since the computed fairness constraints are always guaranteed to hold for the concrete system, one could also try to choose arbitrary

sub-formulae appearing in the system description, and calculate their ranking behavior.

If the result of such a formula is a useful fairness constraint, one can add it to the system. It is guaranteed, that the computed fairness constraint will be satisfied by every instance of the parameterized network, so it is safe to add the constraint to the abstraction. In the worst case, the fairness constraint does not rule out any behavior, but it can never be harmful.

In our case we get the following results:

$$\begin{array}{ll} D_1 = \{h_3\} & I_1 = \{h_2\} \\ D_2 = \{h_4\} & I_2 = \{h_0, h_1\} \\ D_3 = \{c_1, c_2\} & I_3 = \{h_2\} \\ D_4 = \{c_3, c_4, c_5\} & I_4 = \{h_0, h_1, h_3\} \\ D_5 = \{c_3\} & I_5 = \{h_4\} \end{array} \tag{8.11}$$

For each predicate χ_i the following fairness constraint can be added as premise to the liveness property

$$\psi_i \stackrel{\text{def}}{=} (\Box\Diamond(\bigvee_{\tau \in D_i} taken_\tau)) \Rightarrow (\Box\Diamond(\bigvee_{\tau \in I_i} taken_\tau)) \ .$$

For example, from D_4 and I_4 we can derive the following fairness constraint:

$$\begin{aligned} \psi_4 \stackrel{\text{def}}{=} & (\Box\Diamond(taken_{c_3} \vee taken_{c_4} \vee taken_{c_5})) \\ & \Rightarrow (\Box\Diamond(taken_{h_0} \vee taken_{h_1} \vee taken_{h_3})) \end{aligned}$$

Obviously, we also have to rule out taking transition c_0 forever, this can be done with another simple fairness constraint.

It turns out that the computed fairness constraints are too weak to show the liveness property, since it is possible that the home process just never reads the request of a single client, only processing requests of the other clients. Therefore, we have to assume $\texttt{chan1}[p]$ to be fair, such that if home has infinitely often the possibility to read the request of process p, then eventually this will be done.

Assuming these fairness constraints, we can easily prove the exclusive response property.

8.4 Liveness: Shared Access Response

We now prove that also requests for *shared* access will eventually be granted. The same abstraction relation as in Section 8.3 can be used. It turns out, that

even with all fairness constraints given in Section 8.3, the property cannot be proven. Examining a counterexample, we found a further ranking predicate missing, namely

$$\chi_6(i) \stackrel{\text{def}}{=} \texttt{chan2}[i] = \texttt{empty} \ .$$

The fairness constraint derived from this predicate allows to rule out some traces where h_0, h_1, or h_3 are taken infinitely often and neither c_3, c_4, nor c_5 are.

Even this is not sufficient: assuming this fairness constraint enables us to show that eventually a shared grant will be sent from home, but it is possible that this grant will never be read from process p. But if we additionally assume $\texttt{chan2}[p]$ to be fair (or we assume that each process will eventually make a step), then shared access response can be established.

Conclusions and Future Work

In this thesis we have addressed the problem of verification of parameterized systems. This problem is known to be undecidable in general.

We have introduced a uniform framework for modeling parameterized systems. By imposing a number of constraints on this model, a number of parameterized system classes was derived. Among them are classical system classes consisting of a parameterized number of identical user processes which run in parallel synchronously or asynchronously.

The verification problem for these classes has been investigated, exploring the boundary of decidability by weakening undecidable classes by adding more constraints to these system classes.

Furthermore, we have presented a verification method for parameterized networks based on abstraction. Our approach is based on modeling the infinite family of finite state systems as one single higher-order WS1S transition system. This system is then automatically finitely abstracted and model-checked. The user has to provide abstraction predicates from which the abstraction relation is built, hence, our method is not fully automatic.

To verify liveness properties, we have presented an algorithm which computes fairness conditions which are guaranteed to be valid in the concrete system. Since the abstract systems so constructed correspond closely to the concrete system, these fairness constraints can be lifted to the abstract level, removing abstract behavior without concrete counterpart. These fairness constraints are derived from ranking predicates which must be supplied by the user.

This verification method has been implemented in the PAX tool. This PAX tool uses MONA to compute automata which represent the set of all models for WS1S formulae.

We have demonstrated applicability of this verification method by applying the method to a non-trivial example of a cache-coherence protocol. Both safety and liveness properties of this protocol were proven.

Future Work

The classification of parameterized systems given in Chapter 3 can be investigated further. There are combinations of constraints not yet considered, for example, other forms of observability than the ones presented. On the other hand, known results indicate that, for instance, by loosening the constraints on index terms by also allowing indices of the form $i + k$ for constants k will most often lead to undecidable system classes.

Moreover, all the system classes which we have investigated correspond to systems displaying shared variable concurrency. Hence, another interesting class of parameterized systems to study are systems with other forms of communication. There already is some work on parameterized broadcast protocols [EN98, EFM99].

Other interesting system classes may be derived by changing the network topology. For instance, systems organized in a tree structure can be modeled by changing the parameter domain.

Future work with respect to the verification method consists of searching for new system classes which fit in our framework of WS1S systems. For example, systems which are parameterized in two dimensions fall in this category. For instance, systems where every user process has an array of size of the number of user processes, cannot be translated to WS1S systems in a direct way. In this case we already have successfully applied an abstraction to reduce the verification problem to a system with only finitely many arrays.

List of Figures

List of Tables

Bibliography

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In N. Halbwachs and D. Peled, editors, *CAV '99*, volume 1633 of *LNCS*, pages 134–145. Springer, 1999.

[AČJT96] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[AK86] K. Apt and D. Kozen. Limits for Automatic Verification of Finit-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 1986.

[APR+01] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. *Lecture Notes in Computer Science*, 2102:221–234, 2001.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AS87] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Comp.*, 2:117–126, 1987.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 1989.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BD02a] M. Bozzano and G. Delzanno. Algorithmic verification of invalidation-based protocols. In *Proceedings of CAV 2002*, volume 2404 of *LNCS*, page 295 ff, 2002.

[BD02b] M. Bozzano and G. Delzanno. Beyond parameterized verification. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 221–235, 2002.

[BL02] Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, July 27–31 2002.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. "computing abstractions of infinite state systems automatically and compositionally". In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.

[BLS00a] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. In M. Joseph, editor, *FTRTFT'00*, volume 1926, pages 291 – 304. Springer, 2000.

[BLS00b] K. Baukus, Y. Lakhnech, and K. Stahl. Verifying Universal Properties of Parameterized Networks. Technical Report TR-ST-00-4, CAU Kiel, 2000.

[BLS02] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI 2002*, volume 2294 of *LNCS 2294*, pages 317–330, 2002.

[BM02] A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In *Proc. of 7th Intern. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*, 2002.

[Bry85] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 688–694, Los Alamitos, Ca., USA, June 1985. IEEE Computer Society Press.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Büc60] J.R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In CAV [BL02], pages 359–363.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[CCK+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. *Lecture Notes in Computer Science*, 2517:33ff, 2002.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of Synchronisation Skeletons for Branching Time Temporal Logic. In D. Kozen, editor, *Workshop on Logic of Programs 1981*, volume 131 of *LNCS*. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. An early version appeared in *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983.

[CGJ95] E. Clarke, O. Grumberg, and S. Jha. Verifying Parameterized Networks using Abstraction and Regular Languages. In I. Lee

and S. Smolka, editors, *CONCUR '95: Concurrency Theory*, LNCS. Springer, 1995.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Emerson and Sistla [ES00], pages 154–169.

[CGKS02] Edmund M. Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In Brinksma and Larsen [BL02], pages 265–279.

[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, December 1999.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):77–104, August 1996.

[CR99a] S.J. Creese and A.W. Roscoe. Formal verication of arbitrary network topologies. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1033–1039, 1999.

[CR99b] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *Proceedings of FORTE/PSTV'99*, 1999.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00)*, June 2000.

[DGG94] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In E.-R. Olderog, editor, *Proceedings of PROCOMET '94*. North-Holland, 1994.

[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*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 352–359. IEEE Comp. Soc. Press, 1999.

[EK00] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE 2000*, pages 236–254, 2000.

[Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[Eme91] E. A. Emerson. *Handbook of Theoretical Computer Science Vol. B*, chapter Temporal and Modal Logic, pages 997–1072. Elsevier/North Holland, 1991.

[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 *8th Conference on Computer Aided Verification*, LNCS 1102, pages 87–98, 1996.

[EN98] E. Allen Emerson and Kedar S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Logic in Computer Science*, pages 70–80, 1998.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th International Computer Aided Verification Conference*, pages 463–478, 1993.

[ES96] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[ES00] E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[ET99] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHARME'99*, pages 142–156, 1999.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[Fra86] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The temporal analysis of fairness. In *Conference Record of the Seventh annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, ACM, January 1980.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[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 *TACAS '95*, volume 1019 of *LNCS*. Springer, 1996.

[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.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, 2nd edition edition, 2001.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *TACAS'00*, volume 1785. Lecture Notes in Computer Science, 2000.

[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.

[KM01] N. Klarlund and A. Møller. MONA Version 1.4 User Manual. BRICS Notes Series NS-01-1, 2001.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In O. Grumberg, editor, *Proceedings of CAV '97*, volume 1256 of *LNCS*, pages 424–435. Springer, 1997.

[KMS02] Klarlund, Moller, and Schwatzbach. MONA implementation secrets. *IJFCS: International Journal of Foundations of Computer Science*, 13, 2002.

[KV95] Orna Kupferman and Moshe Y. Vardi. On the complexity of branching modular model checking (extended abstract). In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 408–422, Philadelphia, Pennsylvania, 21–24 August 1995. Springer-Verlag.

[KV99] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proc. 11th International Computer Aided Verification Conference*, pages 172–183, 1999.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS 2001*, volume 2031 of *LNCS*, 2001.

[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.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97*, Paris, 1997.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In R. Parikh, editor, *Proceedings 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, Berlin, 1985.

[Mai01] Monika Maidl. A unifying model checking approach for safety properties of parameterized systems. *Lecture Notes in Computer Science*, 2102:311–327, 2001.

[McM92] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* Technical report, CMU, 1992.

[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.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artifitial Intelligence*, pages 481–489, 1971.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. De Bakker, W. P. De Roover, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Lecture Notes on Computer Science, pages 201–284. Springer Verlag, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[PRZ01] Pnueli, Ruah, and Zuck. Automatic deductive verification with invisible invariants. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 2001.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In Emerson and Sistla [ES00], pages 328–343.

[PXZ02] A. Pnueli, J. Xu, , and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *14th International Conference on Computer Aided Verification, CAV 2002*, 2002.

[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.

[Rei85] Wolfgang Reisig. Petri nets. An Introduction. In W. Brauer, G. Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer-Verlag, Berlin, Germany, 1985.

[SBLS99] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In *Proceedings of the 5th International SPIN Workshop on Theoretical Aspects of Model Checking*, volume 1680 of *LNCS*. Springer, 1999.

[SC82] A. P. Sistla and E. Clarke. The complexity of propositional temporal logic. In *14th ACM Symposium on Theory of Computing*, pages 159–167, 1982.

[SC85] A. P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of Assoc. Comput. Mach.*, 32(3):733–749, July 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.

[Sis94] A Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

[Sis97] A. P. Sistla. Parametrized verification of linear networks using automata as invariants. In *Proc. 9th International Computer Aided Verification Conference*, pages 412–423, 1997.

[Suz88] Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Information Processing Letters*, 28(4):213–214, July 1988.

[TA95] W. Thomas and M. Ackermann. *Informatik IV, Skriptum zur Vorlesung.* University Kiel, 1995.

[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.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.

[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*, LNCS 407, pages 68–80, 1989.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM, ACM, January 1986.