Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 97
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 97
Espoo 2005 HUT-TCS-A97

ON BOUNDED MODEL CHECKING OF ASYNCHRONOUS SYSTEMS

Toni Jussila

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 97
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 97
Espoo 2005 HUT-TCS-A97

ON BOUNDED MODEL CHECKING OF ASYNCHRONOUS SYSTEMS

Toni Jussila

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium T1 at Helsinki University of Technology (Espoo, Finland) on the 18th of November, 2005, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Toni Jussila

ISBN 951-22-7903-7
ISSN 1457-7615

Otamedia Oy
Espoo 2005

ABSTRACT: This dissertation studies the verification of reachability properties of concurrent systems where the components of the system are Labeled Transition Systems (LTSs) using a symbolic model checking technique called Bounded Model Checking (BMC). BMC is a technique that seeks to answer the question whether among the system's executions shorter than some given number of steps there is one (or more) violating a given property. Answering this question is reduced to propositional satisfiability, i.e., to a propositional formula that is satisfiable iff there is such a violating execution. The translation from a system to a formula is polynomial in the size of the system but the running time of the propositional solver can be exponential in the number of atomic propositions in the formula. This number, on the other hand, correlates directly with the number of execution steps that the formula models.

Traditionally, LTSs are model checked by composing the components into a synchronized product and then applying a model checking algorithm on this product. The executions of the synchronized product are called *interleaving executions*. The research hypothesis of this work is that by using other composition operators than the one yielding the synchronized product, more efficient BMC algorithms can be obtained. The added efficiency comes from the fact that with these operators, propositional formulas with fewer atomic propositions are obtained. The reduction in the number of atomic propositions follows from the fact that fewer execution steps are needed to cover the same state space than when the synchronized product is used.

Three techniques to create composition operators are presented, namely (i) partial-order semantics, (ii) on-the-fly determinization, and (iii) local transition merging. These techniques can be combined in many ways.

The dissertation demonstrates that given a system of LTSs and a bound, a BMC formula modeling the executions of the products applying partial-order semantics and on-the-fly determinization can be created efficiently. That means that the translation effort is polynomial and the size of the resulting formula is linear in the size of the system and the bound.

The third of the applied techniques, local transition merging, provides potentially dramatic reductions to the bound needed to detect a violation of a reachability property. The size of the BMC formula modeling this execution model is no more linear, though, since a complicated constraint is needed.

The dissertation concludes with some experimental results comparing the products against each other and two state-of-the-art model checking tools.

KEYWORDS: verification, symbolic methods, bounded model checking, labeled transition systems, partial order methods

CONTENTS

1 Introduction **1**
1.1 This Dissertation 5

2 Labeled Transition Systems **9**
2.1 Composition Operators 11
2.1.1 Composition with Partial Order Semantics 12
2.1.2 Composition with On-the-fly Determinization 22
2.1.3 Composition with Local Transition Merging 25

3 Bounded Model Checking **29**
3.1 Completeness 30

4 Encoding Algorithms **34**
4.1 General Conventions 34
4.2 Interleaving Executions 35
4.3 Step Executions 42
4.4 Process Executions 46
4.5 Including On-the-fly Determinization 48
4.5.1 Determinized Interleaving Executions 49
4.5.2 Determinized Step Executions 56
4.5.3 Determinized Process Executions 60
4.6 Reachability Properties 61
4.7 Completeness 64

5 Encoding Algorithm with Local Transition Merging **67**
5.1 Encoding Algorithm for Limited Path Product 70
5.1.1 Consistency Check 73
5.1.2 An Alternative Approach for Consistency Check . . . 88
5.1.3 Iterative Approach 88
5.2 Lifting Constraints from Limited Path Product 89
5.2.1 Limitations on the Structure of the Path 89
5.2.2 Limitations on the Repetition of Same Action 90
5.3 Determinized Limited Path Product 91

6 Implementation **93**
6.1 Optimizations Based on Static Analysis 93
6.2 Translation to a Boolean Formula 98

7 Test Cases **101**

8 Conclusions **114**
8.1 Future Work 116

Bibliography **117**

List of Figures

2.1 Running Example 12
2.2 Synchronized Product 13
2.3 Two Components (left) and their Synchronized Product (right) 13
2.4 Step Product 16
2.5 Process Product (partial state vector) 21
2.6 Determinized Synchronized Product 25
2.7 Determinized Step Product 25
2.8 Determinized Process Product 26
2.9 Determinization and Composition vs. Determinized Composition 28
4.1 Purpose of Preprocessing 51
4.2 The Literal $fs(s_i, \mathfrak{t})$ 63
5.1 Preprocessing Algorithm 69
5.2 Running Example 73
5.3 Inconsistent Executions and the Ordering Graph 73
5.4 Unreachable Transition Cycles and the Ordering Graph . . . 74
5.5 Literal $ltr(t_k, \mathfrak{t})$. 75
5.6 Cyclic Ordering Graph 77
5.7 A Hard Example for Iterative Strengthening 89
5.8 Paths Not Allowed 90
5.9 Repetition of Actions 91
5.10 On-the-fly Determinization 92
6.1 Optimization Algorithm (Beginning) 94
6.2 Optimization Algorithm (End) 94
6.3 Example System 96
6.4 Optimization Algorithm after Step 2, Limited Path Model . . 97
6.5 Completed Optimization Algorithm, Limited Path Model . . 98
6.6 Example Circuits 99
6.7 Schematic Diagram 100
7.1 NuSMV BDD command file 109
7.2 NuSMV BMC command file 110

List of Tables

1.1 Composition Operators Studied in This Work 7
7.1 Test Results of Interleaving, Step and Process Models 103
7.2 Test Results of Interleaving, Step and Process Models (Static Analysis) 104
7.3 Test Results of Interleaving, Step and Process Models (Static Analysis and On-the-fly Determinization) 105
7.4 Comparison between BCzChaff and siege_v4 106
7.5 Comparison between BCzChaff and siege_v4 (Static Analysis) 107
7.6 Comparison between BCzChaff and siege_v4 (Static Analysis and On-the-Fly Determinization) 107
7.7 Test Results of Local Transition Merging Model 108
7.8 Test Results of Local Transition Merging Model (Static Analysis) 108
7.9 Test Results of Comparison Against NuSMV 111
7.10 Test Results of Iterative Encoding against NuSMV BDD . . . 112
7.11 Test Results of Comparison Against SPIN 112

PREFACE

This dissertation is the result of studies and research at the Laboratory for Theoretical Computer Science from 2000 to 2005. I wish to thank Prof. Ilkka Niemelä for supervising this work and for his support and advice. I would also like to thank my instructor, Dr. Keijo Heljanko for reading the preliminary versions of this work and for his advice in improving it.

From January to March 2004, I visited Prof. Javier Esparza's research group at the University of Stuttgart. I would like to thank him and the entire group for the opportunity to this visit.

I would also like to thank Dr. Tommi Junttila for his work on the propositional logic solvers `BCSat` and `BCzChaff`. Both of these solvers have been used to solve the Boolean circuits resulting from this work.

This work has been funded by Helsinki Graduate School for Computer Science and Engineering (HeCSE) and Academy of Finland (project SA-53695). Furthermore, the financial support from Nokia Oyj Foundation is gratefully acknowledged.

Finally, I would like to thank my family for their financial and mental support, and encouragement.

1 INTRODUCTION

Hardware and software systems are nowadays used in applications where failure is not an option, for instance, in air traffic control and medical equipment. Therefore, a growing amount of effort is invested in order to ensure correctness of these systems. The functionality of hardware and software systems is in most cases too complex to be analyzed manually and the complexity tends to grow at a high rate. This is due to added features and concurrency, i.e., the system is composed of several independently executing components. Therefore there is a growing need for computer aided techniques.

Computer systems can be analyzed by simulating and testing their behavior. However, these techniques typically address situations of standard operation whereas experience has shown that errors may arise from a sequence of highly unlikely events. The central problem is that simulation and testing are in general not exhaustive. Therefore, *computer aided verification* has been suggested to complement these methods. The term refers to methods where computational power is used to mathematically establish that the system under study fulfills the given specification or state the conditions under which this is the case.

One of the most widely studied method, also applied in this work, is *model checking* first proposed in the early 1980's independently by two groups [21, 77]. The central idea is to model the system under study in such a way that it is possible to generate a *reachability graph* whose vertices represent the states the system can reach and the edges the possible transitions from a state to another. Each state of the graph is some unique value combination of the system's *state variables*. Based on the values of these state variables, each state is labeled with a set of atomic propositions that hold in that particular state.

The obtained object is often referred to as a *Kripke structure*. The term model checking refers to the fact that the specification to be verified is given as a *temporal logic formula*. By employing a *model checking algorithm*, one can verify that the Kripke structure is a *model* of that formula. If this is the case, then the system fulfills the specification. Otherwise, a counterexample demonstrating the violation of the specification is generated.

To perform model checking, the verifier has to model a concrete system using some specification language. Several possibilities have been proposed depending on the domain. Synchronous digital hardware is typically modeled using a hardware description language whereas for asynchronous systems diverse formalisms exist. A simple but very general formalism is called *transition systems* [3]. The transitions of the system can be associated with some action that is used as a transition label. Such a system model is referred to as a *labeled transition system* (LTS). Some of the other possibilities for modeling asynchronous systems include Petri nets [31], process algebras [68, 69], extended finite state machines (supported by the SPIN tool [42]), protocol specification languages like SDL [45] and various programming languages like Java [4]. However, in many cases the other formalisms can also be seen as describing transition systems. In [3], the relation of both Petri nets and process algebras to transition systems is studied in more detail. In this work, computer aided verification is studied using labeled tran-

sition systems as the modeling formalism.

Applying temporal logic to specifying properties of concurrent programs was first suggested by Pnueli [76]. The term temporal logic does not refer to a single formal language but to a family of them. Two of the mostly applied ones are the computation tree logic (CTL) and the linear temporal logic (LTL). In [22] the semantics of both of them are presented as well as references to the vast literature discussing the merits of each of them.

The properties to be verified are typically characterized as belonging to one of the following three categories: (i) reachability properties, (ii) safety properties, and (iii) liveness properties. Reachability properties require that a state where some particular combination of the state variables occur can be reached. Intuitively, safety properties present claims like: "after a state where p holds is reached a state where q holds can not be reached" (nothing bad happens). Liveness properties state claims that something good happens, e.g., "when a state where p holds is reached, then eventually the system reaches a state where q holds."

Liveness properties are different from the two other categories in that their counterexamples are infinite executions. With reachability and safety properties, a finite amount of execution steps suffices to demonstrate a property violation. Furthermore, safety properties can often be reduced to reachability properties by introducing additional state variables, referred to as *history variables* [17]. This work concentrates on the verification of reachability properties.

The application of model checking is impeded by the fact that the number of states a concurrent system can reach tends to grow rapidly when the number of components grow. This due to concurrency and a combinatorial explosion due to combinations of values of state variables. Therefore a straightforward application of model checking results in huge reachability graphs and the approach of creating the graph and storing it in the memory of a computer thus quickly runs out of steam with larger systems. Several techniques have been developed to alleviate this *state explosion problem* [86] and apply model checking to larger systems. Some of these include:

- Representing sets of states compactly by using e.g. ordered binary decision diagrams (OBDDs) [63, 13, 16].
- Constructing a reduced reachability graph by identifying symmetries in the system [32, 47].
- Replacing the system with an abstraction that has a smaller reachability graph [59, 38].
- Bounded Model Checking, considering only executions of some limited length [8, 7, 9, 11]. This dissertation concentrates on this technique.
- Considering only a subset of the possible paths in the reachability graph by exploiting the independence of transitions (partial order reduction methods) [36, 54, 85].

The first item above is often called *symbolic model checking* referring to the fact that the state space of the system is encoded in a symbolic form

rather than explicitly constructing the reachability graph. The data structure usually employed, ordered binary decision diagram, is a canonical form for representing Boolean functions. Its benefits are that there are efficient algorithms for computing basic Boolean operations using OBDDs and that an OBDD is typically more compact than the corresponding explicit graph. Symbolic model checking has thus been applied to larger systems than possible using an explicit representation of the graph [16]. However, there are also computational problems in applying OBDDs. Namely, different OBDDs modeling the same Boolean function can be constructed by ordering the Boolean variables of the function differently. The size of the OBDD may vary heavily depending on the chosen ordering and the problem of checking that a given ordering yields the smallest OBDD is **NP**-complete [22, 15].[1] Furthermore, there are Boolean functions (e.g., integer multiplication) for which no compact OBDD exists [14, 15].

The symbolic representation of a reachability graph is not limited to OBDDs. The development of propositional logic solvers have brought about proposals to represent sets of states with a propositional formula rather than an OBDD [1, 64, 10, 65]. Hybrid methods combining propositional logic and OBDDs have also been proposed [90, 67, 18, 5]. Results in these papers show that these methods are in some cases able to solve quickly problems hard for the BDD based approach.

Propositional satisfiability is also used in a technique coined as bounded model checking (BMC). BMC encodes the reachability graph symbolically, however, only up to some limited depth [8]. The idea bears close resemblance to that of SAT planning [55]. BMC seeks to answer the question of whether there are executions shorter than this depth that violate the desired property. The graph is encoded as a propositional formula that "unrolls" the transition relation to the given depth. In general, this procedure is not complete, i.e., if no violations are found, the results are inconclusive and the search depth has to be increased. The process is only complete if it is possible to prove e.g., that all the paths in the graph are bound to loop within n steps [57]. Implementations of the technique have compared favorably both in terms of the running time and the memory used [9, 11, 25] when the counterexamples are relatively short. However, in [2] it is argued that shallow counterexamples can also be found effectively using explicit state model checking using randomization.

This dissertation applies BMC to concurrent systems where the components are given as synchronizing LTSs. The goal is to make bounded model checking more efficient. Namely, the BMC procedure consists of two phases. Firstly, the system model is translated to a propositional formula. Secondly, a propositional logic (SAT) solver is used to detect whether that formula has a model. The first phase is polynomial in the size of the system whereas the second can be exponential in the number of atomic propositions in the formula.

Due to this second fact, the following research hypothesis is suggested. The fewer atomic propositions the formula contains, the shorter time it takes from a SAT solver to solve it. Such a hypothesis is, of course, not universally

[1] This dissertation refers to complexity classes in the way that they are defined in [73].

true but should be considered as a rule of thumb for formulas having a "similar structure." In the context of BMC, the number of atomic propositions in the formula directly correlates with the number of execution steps the formula models, i.e., the bound. This dissertation aims to reduce the bound needed to detect a violation of a reachability property of a system given as LTSs. If this is achieved, then simpler (containing fewer atomic propositions) BMC formulas can be used. This reduction in the bound is obtained by taking an alternative view on the transition relation of the concurrent system, yet without losing reachable states. The dissertation presents three techniques improve the performance of BMC. These are described briefly below with references to related work. The contributions are presented in more detail in Section 1.1.

The first technique to reduce the needed bound is to replace the standard interleaving execution model with non-standard models allowing the execution of several visible actions simultaneously. This kind of work can be seen as a subclass of *partial order methods*, more precisely *partial order semantics*. The intuition is that the components forming a concurrent system may execute local transitions and then communicate with their peers. From the perspective of the global property, it is irrelevant in which order the local actions occur. Indeed, they may even occur simultaneously. Thus, there is an order between dependant events but since there is independence in the system, this order is not total but partial.

Partial order semantics has been previously applied in the context of Petri nets. In [71], a technique called *net unfoldings* is presented as a partial order semantics for Petri nets. An unfolding of a Petri net is an acyclic net modeling the behavior of the original. However, in the general case the unfolding can be infinite. McMillan [63, 62] is the first to apply net unfoldings to verification. In [63], an algorithm is provided to compute a *finite complete prefix* of a net unfolding. This prefix contains the full information of the behavior of the Petri net but it can be exponentially more succinct than the reachability graph of the net. Later improvements have been proposed to the original unfolding algorithm (see, e.g. [34]). As shown in [56], the unfolding can also be computed when the Petri Net is extended with integer variables with a finite domain. Finally, in [33], Esparza and Römer demonstrate that the same formalism used in this work, labeled transition systems, can be verified using the net unfolding approach.

Another partial order semantics for Petri nets is *processes* (see, e.g. [6]). A Petri net induces several processes each of which can be seen as a partial order version of an execution. A single process can correspond to exponentially many (in the length of the execution) interleaving executions. Heljanko [39] applies processes to verify reachability properties of 1-safe Petri nets using BMC.

As stated above, the benefit of partial order semantics in the context of bounded model checking is that if simultaneous execution of independent events is allowed, the potential counterexamples demonstrating the violation of a property are shorter. Put another way, if a BMC formula is created for the standard interleaving semantics and a partial order semantics both using the same bound, the latter covers at least as large and often a larger part of the reachable states of the system than the former.

The simplest partial order execution model presented in this work is called *step semantics*, the actual executions being referred to as *step executions*. It is possible to limit the number of executions of the system by only considering executions in a certain normal form, called *process executions*, also presented in this work. The idea behind this normal form is similar to the process semantics for Petri nets.

Even though some of the presented methods can be characterized under the concept of partial order methods, they should not be identified as applying *partial order reduction*. Partial order reduction is also an important technique for alleviating the state explosion problem. The technique uses the independence of transitions to build a reduced (smaller) reachability graph. However, if the original system violates the property to be verified, then even the reduced reachability graph contains a counterexample execution. Therefore, the application of partial order reduction is also referred to as *model checking using representatives* [22]. The executions of the reduced reachability graph, however, still adhere to the interleaving semantics, as contrast to the presented work. More information on partial order reduction methods can be found in [36, 88, 86, 54].

The second technique to possibly improve BMC performance is to model the potentially non-deterministic LTSs as deterministic. It is well-known that a finite-state automaton (FSA) can be determinized using a standard algorithm (see e.g. [74]). An LTS, on the other hand, can be seen as a FSA. However, the standard algorithm is potentially computationally expensive. This dissertation demonstrates that it is possible to work around this by creating a propositional formula whose models correspond to the executions of the determinized equivalents of the components. However, these determinized equivalents are never actually constructed. This can help in reducing the bound since transitions internal to a particular component are "compressed" away from the executions.

The final technique considered in this work to make BMC more efficient, is called *local transition merging*. In this approach, additional transitions are added to the components forming the concurrent system. If a transition is added, then the component has to contain a path between the transition's source and target states. However, the length of the path can be greater than one, i.e., some intermediate states can be skipped. Similar ideas to skip intermediate states (called *path compression*) have been presented in for instance [91, 53, 60]. In those papers, however, the goal is to create a *reduced* reachability graph by replacing the original transitions with a smaller set where several transitions of the original system are compressed to a single transition. In this work, however, no transition of the original system is removed.

The most comprehensive treatment on model checking is probably [22]. The state explosion problem is analyzed in more detail in [86].

1.1 THIS DISSERTATION

This dissertation studies bounded model checking of systems where the components are given as labeled transition systems. Traditionally, model check-

ing of such systems is performed by composing the system's components to a *synchronous product* on which a model checking algorithm is applied. The research question studied is whether it is possible to define other composition operators such that:

1. the counterexamples are potentially shorter and
2. the BMC encoding to a propositional formula is simple.

It is assumed that if a non-standard composition operator fulfills the first criterion, then a BMC procedure based on this operator is more efficient than a BMC encoding using the synchronized product. This is the case since the propositional formulas resulting from the BMC encoding of the non-standard composition operator typically contain fewer atomic propositions than those from the synchronized product to cover the same state space.

The second criterion, simplicity of the encoding algorithm, is also important. In the extreme case, it is possible to conceive an encoding scheme that, given the system and a bound, solves the BMC problem by some other model checking algorithm and produces a propositional formula $\top$, if the verified property holds and $\bot$ otherwise. In this dissertation, the criterion for simplicity is that a comparable effort is needed to create a formula modeling non-standard executions than creating a formula modeling interleaving executions.

The dissertation presents three mutually independent techniques that can in principle be applied in any combination to create a non-standard composition operator. The applied techniques are as follows:

1. partial order semantics (two possibilities),
2. on-the-fly determinization, and
3. merging of local transitions.

The first of the techniques, partial order semantics, allows the simultaneous execution of independent actions resulting in potentially shorter counterexamples to violating states. Two variants of this technique are presented. Besides (i) allowing the simultaneous execution of independent actions it is possible to limit the search space of the SAT solver by (ii) only considering executions in a certain normal form. The former model is called *step semantics* and the latter *process semantics*.

The second of the techniques, on-the-fly determinization, determinizes the components during their composition following the standard subset construction [74]. The potential benefits that this brings about are the following:

1. the number of executions is reduced since non-determinism is removed and
2. the counterexamples are shortened since the internal transitions are removed from the components.

The third of the techniques, local transition merging, can be seen as introducing additional transitions to the components. These transitions correspond to the execution of a *sequence* of local actions. These modified

components can then be composed respecting the standard synchronization rules.

Having introduced these composition operators formally, the dissertation proceeds by presenting for some of them an encoding to a propositional formula. Thus, given a system of LTSs consisting of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is possible to create a formula ϕ such that the models of ϕ are the non-standard executions of the concurrent system formed by $\mathcal{L}_1, \ldots, \mathcal{L}_n$. Table 1.1 presents all possible combinations of the presented techniques (12 in total). It should be noted that the first one is the standard interleaving semantics. Those that are considered in this work are marked with $*$, those that are not are marked with $-$.

The columns in the table are as follows:

- p o sem.: Partial order semantics is applied. The possible values are I for interleaving (partial order semantics not applied), S for step semantics and P for process semantics.
- on-the-fly det.: If marked with a $*$, then on-the-fly determinization is applied.
- trans. merging.: If marked with a $*$, then local transitions are merged.
- studied: Status in this work, if marked with a $*$, then studied and if marked with a $-$, then not studied.

Table 1.1: Composition Operators Studied in This Work

p o sem.	on-the-fly det.	trans. merging	studied
I	-	-	$*$
S	-	-	$*$
P	-	-	$*$
I	$*$	-	$*$
S	$*$	-	$*$
P	$*$	-	$*$
I	-	$*$	$-$
S	-	$*$	$*$
P	-	$*$	$-$
I	$*$	$*$	$-$
S	$*$	$*$	$*$
P	$*$	$*$	$-$

The four composition operators that are not studied are all such that they apply local transition merging. In addition, they apply either interleaving or process semantics. The reason for leaving out composition operators with interleaving semantics and local transition merging is that from the experience from preliminary work with execution models without local transition merging, step semantics seemed to perform better than interleaving semantics.

The reason for leaving out composition operators with process semantics and local transition merging is that so far, no suitable criterion has been

defined to limit the step executions with local transition merging to a sensible normal form. This provides an interesting aspect for future work, though.

Most of the studied composition operators can be encoded compactly using propositional logic. However, for the composition operator applying step semantics, on-the-fly determinization, and local transition merging, this seems not to be the case. This is unfortunate since this product could be useful in BMC. The difficulties found when trying to encode this composition operation are later studied in more detail. Finally, after presenting the operators some experimental results are reported. The different composition operators are compared against each other and also against established model checking tools. Finally, it is analyzed whether the benefits given in the original research hypothesis are obtained.

The dissertation extends the material previously presented in the following publications: Firstly, in [51, 52], the non-standard execution models applying partial order semantics and on-the-fly determinization is presented. Secondly, the paper [49] presents the execution model applying local transition merging. Finally, an iterative encoding of that composition operator is presented in [50].

The structure of the dissertation is as follows. In Chapter 2, the formal theory of labeled transition systems as well as the used non-standard composition operators are presented. Chapter 3 contains a survey of the literature of bounded model checking. In Chapter 4, the propositional encodings for the execution models applying partial order semantics and on-the-fly determinization are presented. Chapter 5 presents the encoding for a version of the execution model applying local transition merging. Then, in Chapter 6, an implementation of the propositional encodings is described together with optimization algorithms based on static analysis. Chapter 7 presents expermental results of this implementation. Finally, Chapter 8 contains the conclusions and some ideas for further work.

2 LABELED TRANSITION SYSTEMS

The BMC methods presented in Chapters 4 and 5 assume that the system model is given as a set of synchronizing labeled transition systems (LTS). This chapter presents a formal definition of an LTS as well as different composition operations for forming larger LTSs in such a way that their executions correspond to the presented non-standard execution models.

Definition 1 *A* labeled transition system $\mathcal{L}$ *is the 4-tuple* $\langle S, I, \Sigma, \Delta \rangle$ *where*

- S *is a finite non-empty set of states,*
- $I \subseteq S$ *is a finite non-empty set of initial states,*
- Σ *is a non-empty set of actions containing the special internal action* τ*, and*
- $\Delta \subseteq S \times \Sigma \times S$*, is the transition relation, the elements of which are called transitions of* $\mathcal{L}$*.*

Let $t = (s, a, s')$ be a transition of LTS $\mathcal{L}$. Then s is the *source* state of t and s' its *target* state. Furthermore t is an *outgoing* transition of state s and an *incoming* transition of state s'. All the actions $a \in \Sigma \setminus \{\tau\}$ are *visible*. The transitions whose middle component is τ are called *internal* (or alternatively *invisible*). A transition is visible iff its middle component is a visible action. An action a is *executable* or *enabled* in some state s of LTS $\mathcal{L}$ iff there is a transition $(s, a, s') \in \Delta$.

In order to be able to compactly refer to the elements presented above, the following definitions are used:

Definition 2 *Let* $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ *be an LTS,* $s \in S$ *its local state, and* $t \in \Delta$ *its local transition. Then*

1. $pr(s)$ *is the set of* incoming transitions *of* s*,*
2. $pt(s)$ *is the set of* outgoing transitions *of* s*,*
3. $sr(t)$ *is the* source state *of* t*, and*
4. $tar(t)$ *is the* target state *of* t*.*

LTSs are typically represented graphically as graphs where the vertices are the states (i.e., the elements of S in the definition above) and the edges are the transitions (i.e., the elements of Δ). The edges are labeled with the action associated with the transition.

The semantics of an LTS is defined in terms of its *executions*. The execution of a single transition of an LTS corresponds intuitively to the execution of an atomic action of the system being modeled. An execution of an LTS is then a connected sequence of transitions $t_1, t_2, \ldots$ such that the source state of t_1 is an initial state of the LTS. In general, executions can be finite or infinite.

Definition 3 *Let σ be a sequence of transitions. The* length *of σ, denoted $|\sigma|$ is an element of the set $\mathbb{N} \cup \{\omega\}$, where $\omega \notin \mathbb{N}$, such that:*

- *if σ is finite, i.e., $\sigma = t_1, \ldots, t_k$, then $|\sigma| = k$ and*
- *if σ is infinite, then $|\sigma| = \omega$.*

The standard operation $<$ for natural numbers is extended to $\mathbb{N} \cup \{\omega\}$ by defining $n < \omega$ for all $n \in \mathbb{N}$. Furthermore, it holds that $n + \omega = \omega + n = \omega$. A formal definition of an execution is then:

Definition 4 *Let $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ be an LTS. Its* execution *σ is the sequence of transitions such that*

- *if $|\sigma| = k, k < \omega$, then*

  $$\sigma = (s_1, a_1, s_2), (s_2, a_2, s_3), \ldots, (s_k, a_k, s_{k+1})$$

  such that

  1. *$s_1 \in I$ and*
  2. *for every $1 \leq i \leq k$, $(s_i, a_i, s_{i+1}) \in \Delta$.*

- *if $|\sigma| = \omega$, then*

  $$\sigma = (s_1, a_1, s_2), (s_2, a_2, s_3), \ldots$$

  such that

  1. *$s_1 \in I$ and*
  2. *for every $i \geq 1$, $(s_i, a_i, s_{i+1}) \in \Delta$.*

In this work, bounded model checking is performed on finite executions. In order to avoid the repetition of states $s_2, \ldots, s_k$ when presenting a finite execution, a finite connected sequence of transitions is in the rest of the work presented in the form:

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_k} s_{k+1}.$$

The execution above *reaches* state s_{k+1}, also called the *final* state of the execution.

Definition 5 *Let σ be an execution of $\mathcal{L}$. Then if σ is finite, i.e., of the form $s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_k} s_{k+1}$, then $\ell(\sigma) = a_1 \ldots a_k$. If σ is infinite, i.e., of the form $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$, then $\ell(\sigma) = a_1 a_2 \ldots$. Let $vis(\sigma)$ be the sequence of labels obtained by removing from $\ell(\sigma)$ the labels τ.*

Definition 6 *Let $\sigma = s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_k} s_{k+1}$ be a finite execution of LTS $\mathcal{L}$. The execution σ is a* deadlock execution *iff the state s_{k+1} does not have any outgoing transitions.*

Given an execution σ of $\mathcal{L}$, its *ith step* or *step i* is $s_i \xrightarrow{a_i} s_{i+1}$. Any connected sequence of transitions from an LTS is called an *execution segment*, or a segment for short. If the last state of a segment is the same as the first state of another segment, they can be combined as follows:

Definition 7 *Let $\sigma = s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_i} s_{i+1}$ and $\sigma' = s_{i+1} \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_k} s_{k+1}$ be finite execution segments. Then, the* concatenation *of σ' to σ, denoted $\sigma\sigma'$, is the execution segment:*

$$s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_k} s_{k+1}$$

The length *of $\sigma\sigma'$ is $|\sigma| + |\sigma'|$.*

When the intermediate states of an execution segment σ are not of great interest, the convention of presenting σ in the form $s \xrightarrow{\ell(\sigma)} s'$ is applied if σ is finite. The notation means that there is an execution from s to s' such that the labels of the execution are $\ell(\sigma)$. In case of infinite executions, the notation $s \xrightarrow{\ell(\sigma)}$ is used.

2.1 COMPOSITION OPERATORS

The treatment proceeds by defining an operation with which a tuple of LTSs can be combined to a single LTS (called their synchronized product), whose executions model the behavior of the components as a synchronizing concurrent system. The intuition of the definition below is that if a visible action a is executed in the product, then every component with a in its alphabet has to execute a transition labeled a. The internal action τ has the special role that any component can execute a τ transition in isolation. The execution of a transition labeled τ in the product corresponds to the case that exactly one component executes a τ transition.

Definition 8 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, such that each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Their* synchronized product*, denoted $\mathcal{L}_1 \parallel \mathcal{L}_2 \parallel \cdots \parallel \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = S_1 \times S_2 \times \cdots \times S_n$,
- $I = I_1 \times I_2 \times \cdots \times I_n$,
- $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n$, *and*
- $\Delta = \{(\langle s_1, \ldots, s_n \rangle, l, \langle s'_1, \ldots, s'_n \rangle) \in S \times \Sigma \setminus \{\tau\} \times S \mid$ *for all* $1 \leq i \leq n$, *if* $l \in \Sigma_i$, *then* $(s_i, l, s'_i) \in \Delta_i$, *otherwise* $s'_i = s_i$.$\} \cup$ $\{(\langle s_1, \ldots, s_n \rangle, \tau, \langle s'_1, \ldots, s'_n \rangle) \in S \times \{\tau\} \times S \mid$ *there is* $1 \leq i \leq n$ *such that* $(s_i, \tau, s'_i) \in \Delta_i$ *and for all* $\mathcal{L}_j$, *if* $i \neq j$, *then* $s'_j = s_j$.$\}$

The composition operation above defines the *interleaving* model of execution, i.e., in each time step at most one action is executed from the components. For that reason, an execution of the synchronized product of components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ is also called their *interleaving execution*. The states s_i forming the state tuples $\langle s_1, \ldots, s_n \rangle \in S$ are referred to as *local states* whereas the complete tuple is a *global state* or a *state vector*. Similarly, a transition $t \in \Delta_i$ for some component $\mathcal{L}_i$ is a *local transition* whereas the

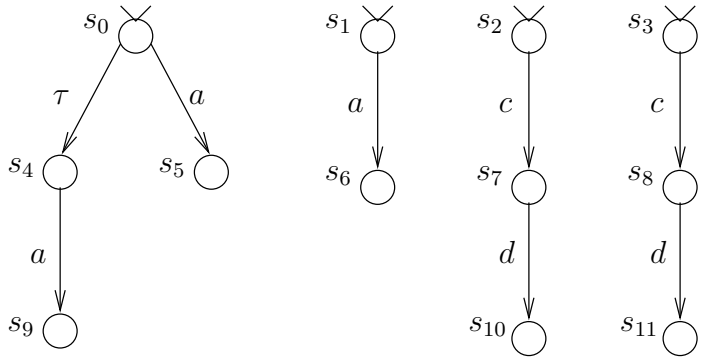

Figure 2.1: Running Example

transitions $t' \in \Delta$ are *global transitions*. The transition relation of the synchronized product is defined in such a way that if a visible action a is executed, then every component having a in its alphabet executes a transition labeled a. This fact is referred to as the *synchronization requirement*.

If the executed action is $l \in \Sigma \setminus \{\tau\}$ in a particular global transition t, then every component i such that $l \in \Sigma_i$ is *scheduled* in t. That is also the case for a component $\mathcal{L}_i$ if $l = \tau$ and $\mathcal{L}_i$ is the one component that executes a local transition. If neither of the cases above apply for a component $\mathcal{L}_i$ and a global transition t, then $\mathcal{L}_i$ is *idle* in t.

Figure 2.1 presents 4 LTSs that are used as a running example. Their synchronized product is presented in Figure 2.2. Following the definition, the states of the synchronized product are 4-tuples, in which the elements are states of the components. Every transition is labeled with either a visible action or the internal action τ. The synchronized product in Figure 2.2 has for instance the following interleaving execution:

$$\langle s_0, s_1, s_2, s_3\rangle \xrightarrow{a} \langle s_5, s_6, s_2, s_3\rangle \xrightarrow{c} \langle s_5, s_6, s_7, s_8\rangle \xrightarrow{d} \langle s_5, s_6, s_{10}, s_{11}\rangle$$

The definition of the synchronized product allows a case where it is not possible to infer from a transition of the product which local transition is executed from the components. The situation is illustrated in Figure 2.3. On the left-hand-side there are two components consisting of a single state and one τ transition that is a self-loop. Their synchronized product on the right-hand side consists of precisely the same elements. However, from an execution of the product, it is not possible to say which local transition is executed in each step. The definition of the synchronized product says that it can be either one. However, intuitively, the definition does not allow both of them to be executed concurrently in a single step.

2.1.1 Composition with Partial Order Semantics

Concurrent systems typically have local transitions that can be considered independent. The precise definition of independence uses the following concept.

Definition 9 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $a \in \Sigma_1 \cup \cdots \cup \Sigma_n$. Then the set C_a is defined by $C_a = \{1 \leq j \leq n \mid a \in \Sigma_j\}$.*

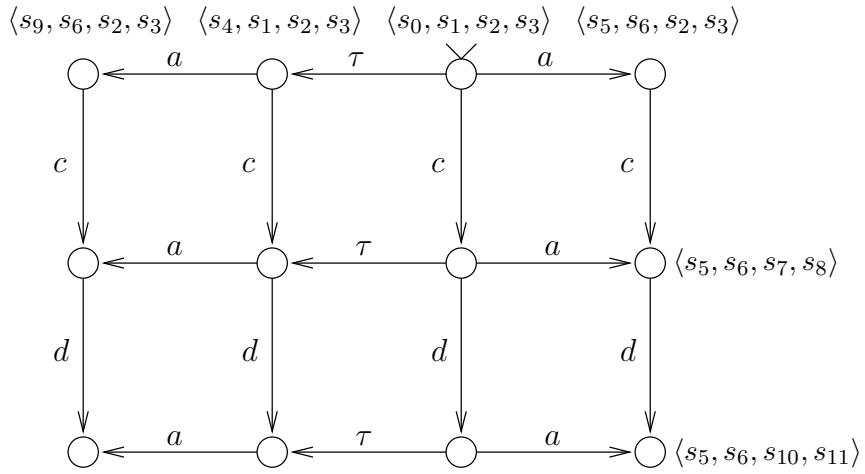

Figure 2.2: Synchronized Product

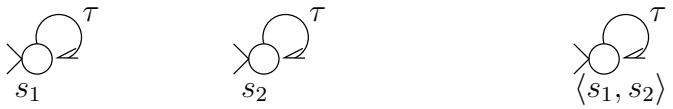

Figure 2.3: Two Components (left) and their Synchronized Product (right)

In this work, the criterion for independence is as follows:

Definition 10 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, $\mathcal{L}$ their synchronized product, and for some $1 \leq i, j \leq n$, $t_1 \in \Delta_i$ and $t_2 \in \Delta_j$ be local transitions. Assume the labels of t_1 and t_2 to be a_1 and a_2, respectively. The transitions t_1 and t_2 are* independent *iff the following conditions hold:*

1. *if $a_1 = \tau$ or $a_2 = \tau$, then $i \neq j$ or*
2. *if $a_1, a_2 \in \Sigma \setminus \{\tau\}$, then $C_{a_1} \cap C_{a_2} = \emptyset$.*

It is possible to define other composition operations utilizing this independence. This can be especially beneficial in the context of bounded model checking since it is possible to prove that with carefully defined non-standard execution models, the set of reachable states remains the same but in most cases shorter executions are needed to reach a particular state.

The simplest of these non-standard execution models is the *step* execution model. It allows, provided that the synchronization requirement is respected, simultaneous execution of independent local transitions. The formal definition is as follows:

Definition 11 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, such that each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Their* step product, *denoted $\mathcal{L}_1 \parallel_{st} \mathcal{L}_2 \parallel_{st} \cdots \parallel_{st} \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = S_1 \times S_2 \times \cdots \times S_n$,

- $I = I_1 \times I_2 \times \cdots \times I_n$,
- $\Sigma = (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times \cdots \times (\Sigma_n \cup \{\epsilon\})$, *and*
- $\Delta = \{(\langle s_1, \ldots, s_n\rangle, \langle l_1, \ldots, l_n\rangle, \langle s'_1, \ldots, s'_n\rangle) \in$
  $(S \times \Sigma \times S) \mid \langle l_1, \ldots, l_n\rangle \neq \langle \epsilon, \ldots, \epsilon\rangle$, *and*
  for all $1 \leq i \leq n$,
  1. *if* $l_i \in \Sigma_i \setminus \{\tau\}$, *then for all* $\mathcal{L}_j$ *such that* $l_i \in \Sigma_j$, *it holds that* $l_j = l_i$ *and there is a transition* $(s_j, l_j, s'_j) \in \Delta_j$,
  2. *if* $l_i = \tau$, *then there is a transition* $(s_i, \tau, s'_i) \in \Delta_i$, *or*
  3. *if* $l_i = \epsilon$, *then* $s'_i = s_i$.$\}$

The executions of the step product of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ are also called *step executions*. The convention is applied that an action a is *executable* or *enabled* in the step (and later process) product in its state $s = \langle s_1, \ldots, s_n\rangle$ iff there is a transition $(s, \langle a_1, \ldots, a_n\rangle, s') \in \Delta$ such that for all $1 \leq i \leq n$, if $a \in \Sigma_i$, then $a_i = a$, else $a_i = \epsilon$. The internal action τ is executable in s iff there is a transition $(s, \langle a_1, \ldots, a_n\rangle, s') \in \Delta$ such that for some $1 \leq i \leq n$, $a_i = \tau$. Furthermore, action a is *executed* iff the executed transition is labeled $\langle a_1, \ldots, a_n\rangle$ so that for some $1 \leq i \leq n$, $a_i = a$. Finally, the action ϵ is called *idling action* whereas an action $a \in \Sigma_1 \cup \cdots \cup \Sigma_n$ is a *non-idling* action.

The step product is an extension of the synchronized product. Namely, it holds that a whenever an action a is executable in the synchronized product, it is also executable in the same state in the step product. In addition, the same state is reached. A given interleaving execution can be seen as a step execution where in each step all the executed local transitions share the same label. However, the set of step executions contains more elements. These executions are characterized by the fact that in some steps several independent transitions can occur simultaneously. The difference between the interleaving and step execution models is illustrated in Figures 2.1, 2.2, and 2.4 that give an example system, its synchronized, and step products, respectively.

In order to be usable, the step product, given the property to be verified, has to give the same answer to a verification question as the synchronized product. For reachability properties this is established as follows:

Theorem 1 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs. Let* σ_{I} *be an execution of the synchronized product reaching the state* $\langle s_1, \ldots, s_n\rangle$. *Then there is an execution* σ_{S} *of the step product reaching the same state.*

Proof. This follows directly from the fact that if action a is executable in the synchronized product, it is executable in the step product and the same state is reached. The step execution is obtained by replacing in each step of the interleaving execution the executed action a with:

- if a is visible, the tuple $\langle a_1, \ldots, a_n\rangle$ such that for all $1 \leq i \leq n$, if $a \in \Sigma_i$, then $a_i = a$, otherwise $a_i = \epsilon$ and
- if $a = \tau$, the tuple $\langle a_1, \ldots, a_n\rangle$ such that $a_i = \tau$ for the component executing a local τ transition and for all $j \neq i$, $a_j = \epsilon$.

□

Theorem 2 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Let σ_{S} be an execution of the step product reaching the state $\langle s_1, \ldots, s_n \rangle$. Then there is an interleaving execution σ_{I} reaching the same state.*

Proof. The proof is by induction over the states of σ_{S} establishing the stronger fact that for any state s of σ_{S}, an interleaving execution σ_{I} reaching s can be constructed.

1. **Base Case.** The initial states in both the synchronized and the step products are the same.

2. **Induction Hypothesis.** Assume σ_{I} can be constructed up to the lth state $\langle s_1^l, \ldots, s_n^l \rangle$ of σ_{S}.

3. **Induction Step.** Consider the step

$$\langle s_1^l, \ldots, s_n^l \rangle \stackrel{\langle a_1, \ldots, a_n \rangle}{\longrightarrow} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$$

in σ_{S}. The interleaving execution σ_{I} is extended with an execution sequence σ'_{I} starting from $\langle s_1^l, \ldots, s_n^l \rangle$ that can contain several steps. Let $\langle t_1, \ldots, t_n \rangle$ be a temporary state variable, initially $\langle s_1^l, \ldots, s_n^l \rangle$. Similarly, let σ'_{I} be the empty execution segment. The algorithm is as follows:

   1. If $\langle a_1, \ldots, a_n \rangle = \langle \epsilon, \ldots, \epsilon \rangle$ return the execution obtained by concatenating σ'_{I} to σ_{I}.
   2. Pick any symbol a_j except ϵ from $\langle a_1, \ldots, a_n \rangle$. If a_j is a visible action, replace a_j with ϵ in all the positions k such that $a_k = a_j$. If $a_j = \tau$, replace a_j with ϵ.
   3. Add the step $\langle t_1, \ldots, t_n \rangle \stackrel{a_j}{\rightarrow} \langle t'_1, \ldots, t'_n \rangle$ to σ'_{I} where
      (i) if a_j is visible and $a_j \in \Sigma_k$, then $t'_k = s_k^{l+1}$,
      (ii) if $a_j = \tau$, then $t'_j = s_j^{l+1}$, or
      (iii) if neither of the cases apply for a component k, then $t'_k = t_k$.
      Let $\langle t_1, \ldots, t_n \rangle = \langle t'_1, \ldots, t'_n \rangle$. Goto step 1.

   It should be justified that the step $\langle t_1, \ldots, t_n \rangle \stackrel{a_j}{\rightarrow} \langle t'_1, \ldots, t'_n \rangle$ added in step 3 of the algorithm above is a transition of the synchronized product. This follows from the fact that initially $\langle t_1, \ldots, t_n \rangle$ is $\langle s_1^l, \ldots, s_n^l \rangle$. From that state, all the local actions forming the tuple $\langle a_1, \ldots, a_n \rangle$ can be executed in the synchronized product. Then, when an action a_j is processed, the reached state $\langle t'_1, \ldots, t'_n \rangle$ is such that if a_j is visible, only the components with a_j in their alphabet move to a new state whereas the others remain in the same state. This is also the case if $a_j = \tau$ for other components than $\mathcal{L}_j$. Therefore, the remaining actions in the tuple $\langle a_1, \ldots, a_n \rangle$ are still executable. Indeed, any order for processing the actions can be chosen. When every symbol has been processed, $\sigma_{\mathrm{I}}\sigma'_{\mathrm{I}}$ reaches the state $\langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$. □

Corollary 1 *The reachable states of the synchronized product and the step product coincide.*

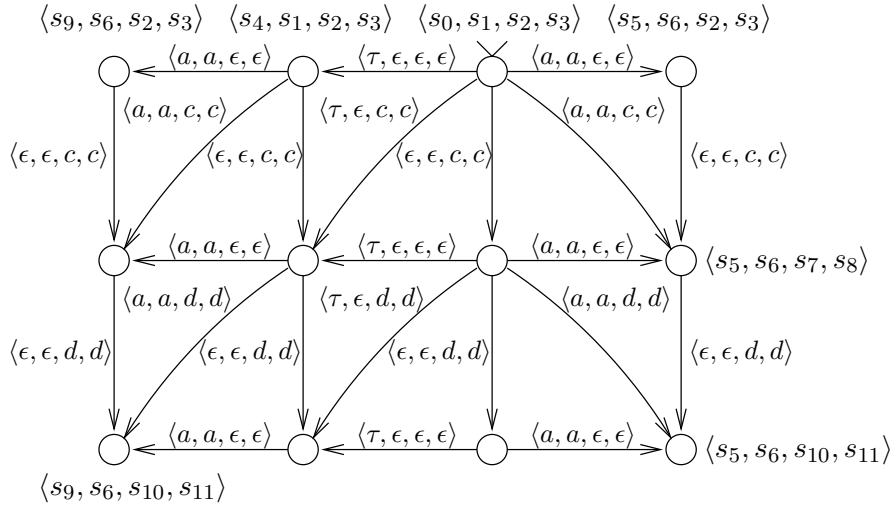

Figure 2.4: Step Product

The step product of the system in Figure 2.1 is presented in Figure 2.4. Since the step executions defined by this product are a superset of the interleaving executions, every transition of the synchronized product is also present in the step product. However, the labels are different corresponding to the different alphabet. For instance, the label a is replaced by the tuple $\langle a, a, \epsilon, \epsilon\rangle$ since a is only in the alphabet of the first two components. The system formed by the components in Figure 2.1 has for instance the following step execution that is not an interleaving execution:

$$\langle s_0, s_1, s_2, s_3\rangle \overset{\langle a,a,c,c\rangle}{\longrightarrow} \langle s_5, s_6, s_7, s_8\rangle \overset{\langle \epsilon,\epsilon,d,d\rangle}{\longrightarrow} \langle s_5, s_6, s_{10}, s_{11}\rangle \tag{2.1}$$

In order to convert the step execution in (2.1) to an interleaving execution, the algorithm given in the proof of Theorem 2 has to be applied. The first tuple of the step execution is $\langle a, a, c, c\rangle$. If action a is processed first, the sequence σ'_1 is:

$$\langle s_0, s_1, s_2, s_3\rangle \overset{a}{\longrightarrow} \langle s_5, s_6, s_2, s_3\rangle \overset{c}{\longrightarrow} \langle s_5, s_6, s_7, s_8\rangle \tag{2.2}$$

For the second step, the tuple is $\langle \epsilon, \epsilon, d, d\rangle$. The interleaving execution segment for this step is obtained by merely replacing the tuple with the single action d as follows:

$$\langle s_5, s_6, s_7, s_8\rangle \overset{d}{\longrightarrow} \langle s_5, s_6, s_{10}, s_{11}\rangle \tag{2.3}$$

The step executions of an LTS typically contain several executions that intuitively correspond to the same concurrent behavior. Keeping the application area, bounded model checking, in mind, it could be sensible to try to limit the search space of the solver so that for each execution only some normal form is considered. This is the reasoning behind the following definition that gives a product whose transition relation is more restrictive than in the case of the step product. However, the product might contain more states. The purpose of the additional restriction is to disallow certain executions.

These executions are characterized by the fact that an action a becomes executable in some step of the execution. It is not, though, immediately executed but the components that are scheduled when a is executed idle for some steps and only then execute a.

Definition 12 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, such that each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Their* process product, *denoted $\mathcal{L}_1 \parallel_{pr} \mathcal{L}_2 \parallel_{pr} \cdots \parallel_{pr} \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = S_1 \times \{\top, \bot\} \times S_2 \times \{\top, \bot\} \times \cdots \times S_n \times \{\top, \bot\}$,
- $I = I_1 \times \{\top\} \times I_2 \times \{\top\} \times \cdots \times I_n \times \{\top\}$,
- $\Sigma = (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times \cdots \times (\Sigma_n \cup \{\epsilon\})$, *and*
- $\Delta = \{(\langle s_1, b_1, \ldots, s_n, b_n \rangle, \langle l_1, \ldots, l_n \rangle, \langle s'_1, b'_1 \ldots, s'_n, b'_n \rangle) \in (S \times \Sigma \times S) \mid \langle l_1, \ldots, l_n \rangle \neq \langle \epsilon, \ldots, \epsilon \rangle.$
  For all $1 \leq i \leq n$,
  1. *if $l_i \in \Sigma_i \setminus \{\tau\}$, then there is a $1 \leq j \leq n$ such that $l_i \in \Sigma_j, b_j = \top$ and for every $\mathcal{L}_k$ such that $l_i \in \Sigma_k$, it holds that $l_k = l_i$, there is a transition $(s_k, l_k, s'_k) \in \Delta_k$, and $b'_k = \top$,*
  2. *if $l_i = \tau$, there is a transition $(s_i, \tau, s'_i) \in \Delta_i$, $b_i = \top$ and $b'_i = \top$, or*
  3. *if $l_i = \epsilon$, then $s'_i = s_i$ and $b'_i = \bot$.}*

Given a tuple of components, the executions of their process product are also called *process executions*. To be able to verify reachability properties, the process product should also preserve the set of reachable states. This is established by proving that the reachable states of the step and process products coincide when the truth values present in the state vectors of the process product are omitted. [1] Thus, by Corollary 1, the reachable states are preserved.

The proof makes use of the following concept:

Definition 13 *Let*

$$\langle s_1^1, \ldots, s_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\rightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\rightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

be an execution σ_S of the step product of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$. A non-idling action l_i^j fulfills the process condition *iff one of the following three cases holds:*

1. *the value of j is 1, i.e., the action is executed in the first step,*
2. *if $j > 1$ and l_i^j is visible, then some component k such that $l_i^j \in \Sigma_k$ is scheduled in step $j - 1$, or*
3. *if $j > 1$ and $l_i^j = \tau$, then i is scheduled in step $j - 1$.*

[1] The reachability properties are defined in terms of component state combinations, so nothing is lost in the process.

The execution σ_{S} fulfills the process condition iff every non-idling action in its every step fulfills the process condition.

Lemma 1 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $\mathcal{L}_{pr}$ and $\mathcal{L}_{st}$ be their process and step products, respectively. An execution of the form*

$$\langle s_1^1, b_1^1, \ldots, s_n^1, b_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\longrightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\longrightarrow} \langle s_1^{k+1}, b_1^{k+1}, \ldots, s_n^{k+1}, b_n^{k+1} \rangle$$

is an execution of $\mathcal{L}_{pr}$ iff

$$\langle s_1^1, \ldots, s_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\longrightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\longrightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

is an execution of $\mathcal{L}_{st}$ fulfilling the process condition.

Proof. The intuition of the lemma above is that if the truth values b_i^j are omitted from the state vectors of an execution of the process product, then an execution of the step product of the same components is obtained. Furthermore, that step execution fulfills the process condition. Conversely, a step execution fulfilling the process condition can be converted to a process execution by inserting the truth values to the state vector of the step execution in such a way that the definition of the transition relation of $\mathcal{L}_{pr}$ is respected (elaborated below).

Firstly, the transition relations of $\mathcal{L}_{pr}$ and $\mathcal{L}_{st}$ are otherwise the same, but Δ_{pr} imposes an additional condition on the truth values in the state vector of the transition's source state. Thus, if

$$\langle s_1^i, b_1^i, \ldots, s_n^i, b_n^i \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\longrightarrow} \langle s_1^{i+1}, b_1^{i+1}, \ldots, s_n^{i+1}, b_n^{i+1} \rangle \in \Delta_{pr},$$

then

$$\langle s_1^i, \ldots, s_n^i \rangle \overset{\langle l_1^i, \ldots, l_n^i \rangle}{\longrightarrow} \langle s_1^{i+1}, \ldots, s_n^{i+1} \rangle \in \Delta_{st}.$$

The proof in the first direction (from $\mathcal{L}_{pr}$ to $\mathcal{L}_{st}$) is by induction over the steps of the given process execution. Let the step and process executions in question be denoted σ_{S} and σ_{P}, respectively.

> **Base Case.** Firstly, by definition of the products $\mathcal{L}_{st}$ and $\mathcal{L}_{pr}$, the state $\langle s_1^1, \ldots, s_n^1 \rangle \in I_{st}$. Secondly, all the actions in the first transition fulfill the process condition.
>
> **Induction Hypothesis.** Assume that up to some state l, σ_{S} is an execution of $\mathcal{L}_{st}$ fulfilling the process condition.
>
> **Induction Step.** Consider the step
>
> $$\langle s_1^l, b_1^l, \ldots, s_n^l, b_n^l \rangle \overset{\langle l_1^l, \ldots, l_n^l \rangle}{\longrightarrow} \langle s_1^{l+1}, b_1^{l+1}, \ldots, s_n^{l+1}, b_n^{l+1} \rangle$$
>
> of σ_{P}. If action l_i^l is visible, then by definition of $\mathcal{L}_{pr}$, in the process product, the truth value $b_j^l = \top$ for at least one component j such that $l_i^l \in \Sigma_j$. However, this implies that the component j is scheduled in step l and thus l_i^l fulfills the process condition. If, on the other hand, $l_i^l = \tau$, then by definition of $\mathcal{L}_{pr}$, $b_i^l = \top$. However, this implies

that component i is scheduled in step l and thus l_i^l fulfills the process condition.

Thus, in step

$$\langle s_1^l, \ldots, s_n^l \rangle \overset{\langle l_1^l, \ldots, l_n^l \rangle}{\longrightarrow} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$$

every action fulfills the process condition.

The other direction of the proof states that if σ_{S} is an execution of $\mathcal{L}_{st}$ fulfilling the process condition, then an execution σ_{P} of $\mathcal{L}_{pr}$ can be constructed by inserting the truth values $\top$ for the values b_i^1. For the remaining steps, if component i executes an action in step j, then $b_i^{j+1} = \top$, otherwise $b_i^{j+1} = \bot$. That the execution σ_{P} obtained in this way is an execution of $\mathcal{L}_{pr}$, is established by induction over the transitions of σ_{S}.

Base Case. By definition of the products $\mathcal{L}_{st}$ and $\mathcal{L}_{pr}$, if $\langle s_1^1, \ldots, s_1^n \rangle \in I_{st}$, then $\langle s_1^1, \top, \ldots, s_1^n, \top \rangle \in I_{pr}$ Secondly, since every $b_1^l = \top$, the first transition of σ_{P} is a transition of Δ_{pr}.

Induction Hypothesis. Assume that up to some state l, σ_{P} is an execution of $\mathcal{L}_{pr}$.

Induction Step. Consider the step

$$\langle s_1^l, \ldots, s_n^l \rangle \overset{\langle l_1^l, \ldots, l_n^l \rangle}{\longrightarrow} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$$

of σ_{S}. By definition of the process condition, if l_i^l is a visible action, then some component j such that $l_i^l \in \Sigma_j$ is scheduled in step l. However, by induction hypothesis, then $b_j^l = \top$ and thus it is possible to execute l_i^l in step l.

Secondly, if $l_i^l = \tau$, since σ_{S} fulfills the process condition, that component is scheduled in step l. However, by induction hypothesis, then $b_i^l = \top$ and it is possible to execute τ in that particular component in step l. Therefore, the lth transition of σ_{P} is also a transition of $\mathcal{L}_{pr}$. □

Lemma 2 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $\mathcal{L}_{st}$ their step product. If*

$$\langle s_1^1, \ldots, s_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\longrightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\longrightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

is an execution σ_{S} of $\mathcal{L}_{st}$ but some action l in some step $\mathfrak{t}$ violates the process condition, then σ_{S} can be converted to another step execution σ'_{S} of $\mathcal{L}_{st}$ such that:

1. *l no longer violates the process condition,*
2. *$|\sigma'_{\mathrm{S}}| \leq |\sigma_{\mathrm{S}}|$,*
3. *σ'_{S} reaches the same state as σ_{S}, and*
4. *for every action l' executed in steps $1 \leq k < \mathfrak{t}$ in σ_{S}, the process condition for l' holds in σ'_{S} iff it holds in σ_{S}.*

Proof. The conversion process is as follows. Initially, let $\sigma'_{\mathrm{S}} = \sigma_{\mathrm{S}}$. Since l in step $\mathfrak{t}$ violates the process condition, then one of the following two cases is true:

- if l is visible, then no component k such that $l \in \Sigma_k$ is scheduled in step $\mathfrak{t}-1$, or
- if $l = \tau$ and the local transition is executed from $\mathcal{L}_i$, then $\mathcal{L}_i$ is not scheduled in step $\mathfrak{t}-1$.

However, this implies in the former case that for all the components $\mathcal{L}_k$ where $l \in \Sigma_k$, $s_k^{\mathfrak{t}} = s_k^{\mathfrak{t}-1}$ and in the latter case for the single component $\mathcal{L}_i$ that $s_i^{\mathfrak{t}} = s_i^{\mathfrak{t}-1}$. Since these local states are the same in both global states $\mathfrak{t}-1$ and $\mathfrak{t}$, action l can be moved to step $\mathfrak{t}-1$. More precisely, the modifications to σ'_{S} are:

- if the violating action l is visible, then set $l_k^{\mathfrak{t}-1} = l$ and $l_k^{\mathfrak{t}} = \epsilon$ in all the components k such that $l \in \Sigma_k$. The component states $s_k^{\mathfrak{t}}$ and $s_k^{\mathfrak{t}+1}$ of σ'_{S} both become $s_k^{\mathfrak{t}+1}$ of σ_{S}, or
- if the violating action $l = \tau$, then set $l_i^{\mathfrak{t}-1} = \tau$ and $l_i^{\mathfrak{t}} = \epsilon$. The component states $s_i^{\mathfrak{t}}$ and $s_i^{\mathfrak{t}+1}$ of σ'_{S} both become $s_i^{\mathfrak{t}+1}$ of σ_{S}.

It may be the case that after the above transformation the same action l, this time in step $j-1$, does still not fulfill the process condition. However, then the above procedure can be repeated. Action l is eventually bound satisfy the process condition since it is always pushed to the direction of the beginning of σ_{S} and if it reaches the first step, then it fulfills the process condition.

If it is the case that after removing l from a particular action tuple, the tuple becomes $\langle \epsilon, \ldots, \epsilon \rangle$, then the tuple and the following global state are removed from σ'_{S}. Therefore, the execution σ'_{S} can only become shorter than σ_{S}.

Since in both executions σ_{S} and σ'_{S} all the components execute the same local transitions in the same order, in both of them all the components reach the same local states. Finally, proposition 4 in the lemma states that action l can not alter the process condition of any executed actions in steps prior to j. This is due to the fact that the process condition on actions executed in any step $\mathfrak{t}$ (besides the first one) is defined based on scheduling of components in step $\mathfrak{t}-1$. If l is moved from step $\mathfrak{t}$ to $\mathfrak{t}-1$, it does not change the scheduling of any component prior to step $\mathfrak{t}-1$ and thus can not alter the process condition of any executed action in steps up to $\mathfrak{t}-1$. □

Now the following theorem can be established:

Theorem 3 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. If σ_{S} is an execution of their step product reaching the state $\langle s_1, \ldots, s_n \rangle$, then there is an execution σ_{P} of their process product reaching the state $\langle s_1, b_1, \ldots, s_n, b_n \rangle$ for some Boolean values $b_1, \ldots, b_n$.*

Proof. By repeated application of the construct in Lemma 2, σ_{S} can be converted to another step execution that reaches the same state and fulfills the

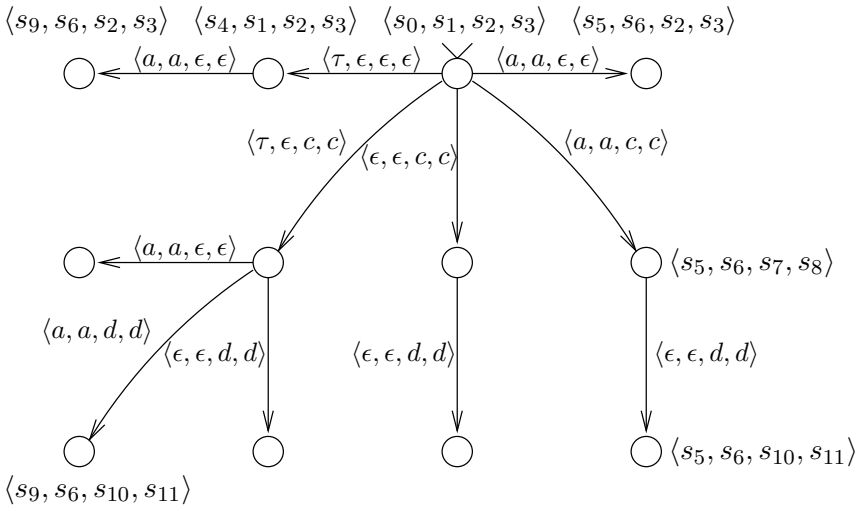

Figure 2.5: Process Product (partial state vector)

process condition. Since the number of executed actions in σ_{S} is finite, this process is bound to terminate. Then, by Lemma 1, for any such execution there is a process execution σ_{P} such that in the final states of σ_{S} and σ_{P}, the component states are the same. □

Corollary 2 *Given n components, the reachable states of their synchronized and process products coincide.*

Figure 2.5 presents the process product of the components presented in Figure 2.1. Let s be any state of Figure 2.5. It holds that the corresponding state in Figure 2.4 has at least as many outgoing transitions as s. However, the reachable global states are preserved.

Due to space restrictions, Figure 2.5 applies the convention of only presenting the state vector partially, omitting the Boolean values associated with the states. As stated above, the Boolean value of some component $\mathcal{L}_i$ is $\top$ iff (i) the global state is an initial state or (ii) along the incoming edges to the global state, $\mathcal{L}_i$ is scheduled. The step execution in (2.1) fulfills the process condition and can thus be converted to a process execution as follows.

$$\langle s_0, \top, s_1, \top, s_2, \top, s_3, \top\rangle \overset{\langle a,a,c,c\rangle}{\longrightarrow} \langle s_5, \top, s_6, \top, s_7, \top, s_8, \top\rangle \overset{\langle \epsilon,\epsilon,d,d\rangle}{\longrightarrow} \langle s_5, \bot, s_6, \bot, s_{10}, \top, s_{11}, \top\rangle \tag{2.4}$$

If a step execution is to be converted to a process execution, the construct given in the proof of Lemma 2 has to be applied. Consider the following execution of the step product where the action c in the second step does not fulfill the process condition:

$$\langle s_0, s_1, s_2, s_3\rangle \overset{\langle a,a,\epsilon,\epsilon\rangle}{\longrightarrow} \langle s_5, s_6, s_2, s_3\rangle \overset{\langle \epsilon,\epsilon,c,c\rangle}{\longrightarrow} \langle s_5, s_6, s_7, s_8\rangle \overset{\langle \epsilon,\epsilon,d,d\rangle}{\longrightarrow} \langle s_5, s_6, s_{10}, s_{11}\rangle$$

Now, pushing action c back one step leads to the following:

$$\langle s_0, s_1, s_2, s_3 \rangle \overset{\langle a,a,c,c \rangle}{\longrightarrow} \langle s_5, s_6, s_7, s_8 \rangle \overset{\langle \epsilon,\epsilon,\epsilon,\epsilon \rangle}{\longrightarrow}$$
$$\langle s_5, s_6, s_7, s_8 \rangle \overset{\langle \epsilon,\epsilon,d,d \rangle}{\longrightarrow} \langle s_5, s_6, s_{10}, s_{11} \rangle$$

The execution above contains the label $\langle \epsilon, \epsilon, \epsilon, \epsilon \rangle$ that has to be removed. Removing it yields:

$$\langle s_0, s_1, s_2, s_3 \rangle \overset{\langle a,a,c,c \rangle}{\longrightarrow} \langle s_5, s_6, s_7, s_8 \rangle \overset{\langle \epsilon,\epsilon,d,d \rangle}{\longrightarrow} \langle s_5, s_6, s_{10}, s_{11} \rangle$$

Now, every action fulfills the process condition and the execution can be converted to an execution of the process product. That process execution is already presented in (2.4).

It should be noted that even though process executions are characterized by saying that actions occur as early as possible, the execution model is not such that starting from the initial state, a maximal set of independent actions has to be executed. Indeed, such an execution model would lose some reachable states. Rather, the execution model is such that executions where if some action a is possible in the ith step, the components with a in their alphabet can not idle only to execute a some steps later.

2.1.2 Composition with On-the-fly Determinization

So far, no limitations have been imposed on the structure of the component LTSs. In general, an LTS can be seen as a finite state automaton (FSA) [74], where every state of the LTS is an accepting state. One way of characterizing FSAs is by dividing them to deterministic and non-deterministic ones. Deterministic automata are such that they have only one initial state, no internal transitions exist and from every state there is precisely one transition for every action in their alphabet. Non-deterministic automata, on the other hand, can be determinized so that the accepting sequences of visible actions are the same.

This fact could be beneficial in the context of bounded model checking since eliminating internal τ transitions yields product LTSs with potentially shorter counterexamples. Furthermore, it is a working hypothesis that the search space of the SAT solver is smaller since given a sequence of visible actions, a deterministic system has precisely one execution (sequence of reached states) with that action sequence. In the non-deterministic system, on the other hand, this number can be exponential (in the length of the sequence).

Thus, replacing potentially non-deterministic components with their deterministic equivalents before applying a composition operator may lead to gains in the running time of the verification task. Non-deterministic automata can be determinized by using the standard construct presented in textbooks, e.g. [74]. In the construct, the states in the determinized automaton represent sets of states in the original, non-deterministic automaton. Thus, the algorithm is potentially expensive since if the original automaton has n states, the resulting deterministic equivalent can have as many as 2^n states.

There is also another problem in applying the standard determinization procedure. That is the fact that the transition relation is a *function* from the Cartesian product of the states and the alphabet and thus defined for every state-action pair. If there is no state reachable in the non-deterministic automaton by executing an action from a particular state, the deterministic counterpart contains an "error" state that has a self-loop for every action in the alphabet.

Applying the standard determinization procedure thus has the undesired effect that an interesting reachability property, namely a deadlock, a global state where no transition is possible, is lost. Therefore, in order to reach the desired improvement in bounded model checking, the determinization algorithm has to allow the transition relation to be partial. This determinization construct applies the following concepts:

Definition 14 *Let $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ be an LTS and $S' \subseteq S$. The τ-*closure *of S', denoted $\tau(S')$, is the maximal set of states S'' such that $S' \subseteq S''$ and $s' \in S''$ iff there is an execution segment from a state $s \in S'$ to s' labeled with only τ transitions.*

Definition 15 *Let $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ be an LTS, $S' \subseteq S$ and $a \in \Sigma$. Then $\Delta^a_{S'} = \{(s, a, s') \in \Delta \mid s \in S'\}$, i.e., the set of transitions labeled a whose source state is in the state set S'.*

Definition 16 *Let $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ be an LTS and $\Delta' \subseteq \Delta$. Then $pt(\Delta') = \{s' \in S \mid (s, l, s') \in \Delta'\}$, i.e., the set of target states of the transitions in Δ'.*

The determinization construct suitable for deadlock checking is then as follows:

Definition 17 *Let $\mathcal{L} = (S, I, \Sigma, \Delta)$ be an LTS. Its* determinized equivalent *is the LTS $\mathcal{L}^d = (S^d, I^d, \Sigma^d, \Delta^d)$ where:*

- $S^d = 2^S$,
- $I^d = \tau(I)$,
- $\Sigma^d = \Sigma$, *and*
- $\Delta^d = \{(S', l, S'') \in S^d \times \Sigma^d \setminus \{\tau\} \times S^d \mid \Delta^l_{S'} \neq \emptyset$ *and* $S'' = \tau(pt(\Delta^l_{S'}))$.$\}$

However, it is not necessary to require deterministic components nor to determinize then using the modified construct above. It is possible to define composition operations that determinize the components "on-the-fly" so that the benefits of deterministic components are obtained.

Definition 18 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, such that each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Their* determinized synchronized product, *denoted $\mathcal{L}_1 \parallel^d \mathcal{L}_2 \parallel^d \cdots \parallel^d \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = 2^{S_1} \times 2^{S_2} \times \cdots \times 2^{S_n}$,

- $I = \tau(I_1) \times \tau(I_2) \times \cdots \times \tau(I_n)$,
- $\Sigma = (\Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n)$, *and*
- $\Delta = \{(\langle T_1, \ldots, T_n \rangle, l, \langle T'_1, \ldots, T'_n \rangle) \in S \times \Sigma \setminus \{\tau\} \times S \mid$
  For all $1 \leq i \leq n$, *if* $l \in \Sigma_i$, *then* $\Delta^l_{T_i} \neq \emptyset$ *and*
  $T'_i = \tau(pt(\Delta^l_{T_i}))$. *Otherwise* $T'_i = T_i$.$\}$

As noticed from the definition above, the transition relation of the determinized synchronized product does not contain any transitions labeled τ. The same idea can be similarly applied with the composition operators applying partial order semantics.

Definition 19 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs, such that each* $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. *Their* determinized step product, *denoted* $\mathcal{L}_1 \parallel^d_{st} \mathcal{L}_2 \parallel^d_{st} \cdots \parallel^d_{st} \mathcal{L}_n$, *is the LTS* $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ *such that:*

- $S = 2^{S_1} \times 2^{S_2} \times \cdots \times 2^{S_n}$,
- $I = \tau(I_1) \times \tau(I_2) \times \cdots \times \tau(I_n)$,
- $\Sigma = ((\Sigma_1 \setminus \{\tau\}) \cup \{\epsilon\}) \times ((\Sigma_2 \setminus \{\tau\}) \cup \{\epsilon\}) \times \cdots \times ((\Sigma_n \setminus \{\tau\}) \cup \{\epsilon\})$, *and*
- $\Delta = \{(\langle T_1, \ldots, T_n \rangle, \langle l_1, \ldots, l_n \rangle, \langle T'_1, \ldots, T'_n \rangle) \in S \times \Sigma \setminus \{\tau\} \times S \mid$
  $\langle l_1, \ldots, l_n \rangle \neq \langle \epsilon, \ldots, \epsilon \rangle$, *and for all* $1 \leq i \leq n$,
  1. if $l_i \neq \epsilon$, *then for all* $\mathcal{L}_j$ *such that* $l_i \in \Sigma_j$, *it holds that*
  $l_j = l_i$, $\Delta^{l_j}_{T_j} \neq \emptyset$ *and* $T'_j = \tau(pt(\Delta^{l_j}_{T_j}))$, *or*
  2. if $l_i = \epsilon$, *then* $T'_i = T_i$.$\}$

Definition 20 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs, such that each* $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. *Their* determinized process product, *denoted* $\mathcal{L}_1 \parallel^d_{pr} \mathcal{L}_2 \parallel^d_{pr} \cdots \parallel^d_{pr} \mathcal{L}_n$, *is the LTS* $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ *such that:*

- $S = 2^{S_1} \times \{\top, \bot\} \times 2^{S_2} \times \{\top, \bot\} \times \cdots \times 2^{S_n} \times \{\top, \bot\}$,
- $I = \tau(I_1) \times \top \times \tau(I_2) \times \top \times \cdots \times \tau(I_n) \times \top$,
- $\Sigma = ((\Sigma_1 \setminus \{\tau\}) \cup \{\epsilon\}) \times ((\Sigma_2 \setminus \{\tau\}) \cup \{\epsilon\}) \times \cdots \times ((\Sigma_n \setminus \{\tau\}) \cup \{\epsilon\})$, *and*
- $\Delta = \{(\langle T_1, b_1, \ldots, T_n, b_n \rangle, \langle l_1, \ldots, l_n \rangle, \langle T'_1, b'_1 \ldots, T'_n, b'_n \rangle) \in$
  $S \times \Sigma \setminus \{\tau\} \times S \mid \langle l_1, \ldots, l_n \rangle \neq \langle \epsilon, \ldots, \epsilon \rangle$, *and for all* $1 \leq i \leq n$,
  1. if $l_i \neq \epsilon$, *then there is a* $1 \leq j \leq n$ *such that*
  $l_i \in \Sigma_j, b_j = \top$, *and for all* $\mathcal{L}_k$ *such that* $l_i \in \Sigma_k$, *it holds that*
  $l_k = l_i$, $\Delta^{l_i}_{T_k} \neq \emptyset$, $T'_k = \tau(pt(\Delta^{l_i}_{T_k}))$ *and* $b'_k = \top$, *or*
  2. if $l_i = \epsilon$, *then* $T'_i = T_i$ *and* $b'_i = \bot$.$\}$

As can be seen in the definitions above, the number of states is potentially exponential compared to the non-determinizing products. However, as can be seen later, this does not adversely affect the size of the propositional formula resulting from the BMC encoding procedure. Namely, given n components it is possible to create a formula whose size is linear in the size of the

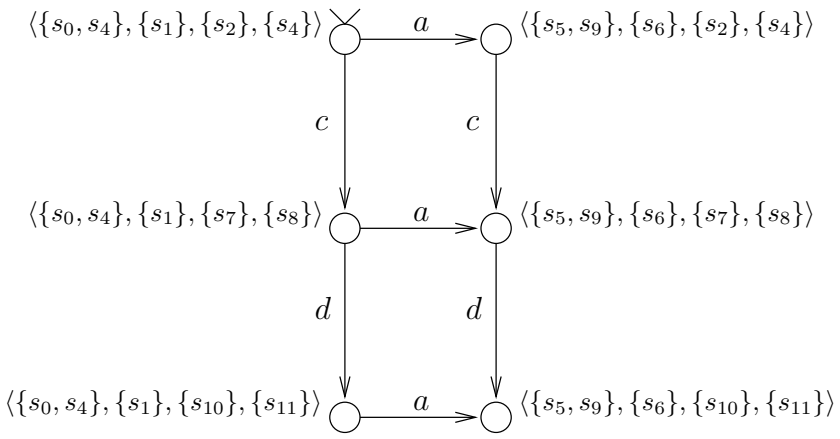

Figure 2.6: Determinized Synchronized Product

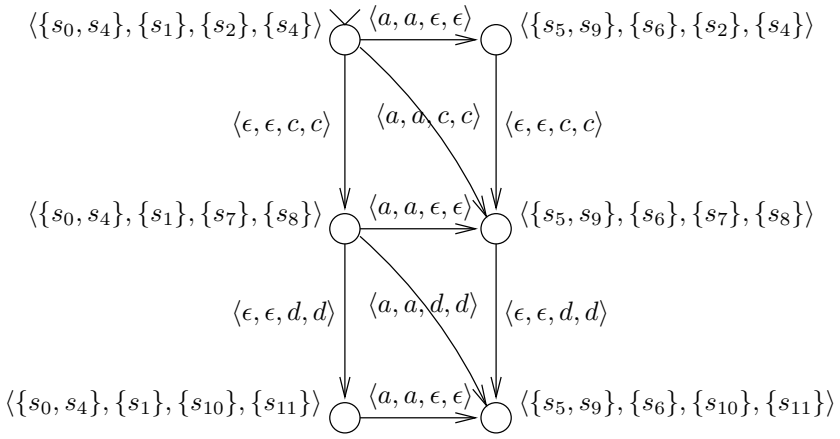

Figure 2.7: Determinized Step Product

system but whose models correspond to the executions of the *determinized* products above.

Figures 2.6, 2.7, and 2.8 are the determinized synchronized, determinized step, and determinized process products of the running example given in Figure 2.1. From these products it can be seen that with this example, removing non-determinism reduces the number of states, for instance the step product (given in Figure 2.2) contains 12 states whereas the determinized step product (given in Figure 2.7) contains only 6 states. The state vectors are more complicated, though, since each component can be in a set of states.

2.1.3 Composition with Local Transition Merging

Both step and process products are such that due to the additional edges executing several independent actions, the paths to undesirable states are potentially shorter than in the LTS resulting from composing components using the standard synchronized product. However, step and process products are by no means the most general of such composition operators. Indeed, it is

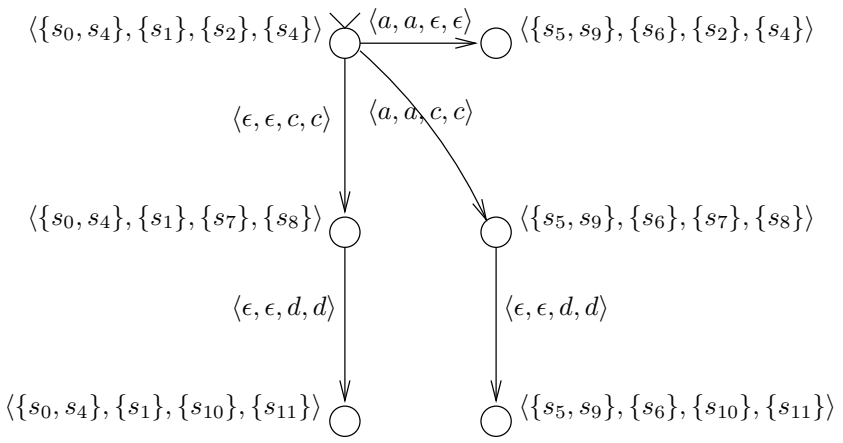

Figure 2.8: Determinized Process Product

possible to define a composition operator such that any violation of a reachability property can be detected by an execution of length one. However, as elaborated below, the BMC encoding of this product is probably not polynomial in the size of the system.

Definition 21 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i\rangle$. Let $\mathcal{L}_{st} = (S_{st}, I_{st}, \Sigma_{st}, \Delta_{st})$ be the step product $\mathcal{L}_1 \parallel_{st} \cdots \parallel_{st} \mathcal{L}_n$. The* path product *of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, denoted $\mathcal{L}_1 \parallel_{pt} \mathcal{L}_2 \parallel_{pt} \cdots \parallel_{pt} \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta\rangle$ such that:*

- $S = S_1 \times S_2 \times \cdots \times S_n$,
- $I = I_1 \times I_2 \times \cdots \times I_n$,
- $\Sigma = \{r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle} \mid \langle s_1, \ldots, s_n\rangle \in S, \langle s'_1, \ldots, s'_n\rangle \in S\}$, *and*
- $\Delta = \{\langle s_1, \ldots, s_n\rangle, r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle}, \langle s'_1, \ldots, s'_n\rangle \mid$ *there is an execution from $\langle s_1, \ldots, s_n\rangle$ to $\langle s'_1, \ldots, s'_n\rangle$ in $\mathcal{L}_{st}$.*$\}$

The definition of the transition relation Δ above can be intuitively characterized by saying that it is obtained from the transition relation of the corresponding step product by adding all the transitive edges. The label of a transition should somehow characterize the execution segments in $\mathcal{L}_{st}$ from its source state $\langle s_1, \ldots, s_n\rangle$ to its target state $\langle s'_1, \ldots, s'_n\rangle$. However, the language formed by all such label sequences is necessarily regular. This follows from the fact that it is possible to construct an automaton from $\mathcal{L}_{st}$ whose initial state is $\langle s_1, \ldots, s_n\rangle$ and the only final state is $\langle s'_1, \ldots, s'_n\rangle$. Clearly, the accepting of runs of this automaton are execution segments from $\langle s_1, \ldots, s_n\rangle$ to $\langle s'_1, \ldots, s'_n\rangle$ in $\mathcal{L}_{st}$ and all of these executions form a regular language [74]. Thus, labels of the form $r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle}$ are employed.

If a reachability property holds, then the step product has an execution that is a witness of the property. By definition of the transition relation of the path product, it has an edge between the initial and final states of the witness. From this it follows that if a reachability property holds, then the path product has a witness of length one. Therefore, it is assumed that given n components

it is hard to conceive a bounded model checking scheme encoding the executions of the path product so that the resulting formula is polynomial in the size of the system. Namely, if this was the case, a **PSPACE**-complete problem (reachability of a state in the synchronized product of LTSs [58]) would be solved in **NP**.

Due to the theoretical result above, the approach taken in this dissertation is to consider a limited version of the path product. This product employs the idea of merging local transition to a single step. However, a local loop can be executed at most once and action repetition is disallowed. This construction, its BMC encoding (whose size is polynomial in the size of the system), and reasons for the chosen limitations are discussed in Chapter 5.

Lemma 3 *Given n LTSs, the reachable states of their synchronized and path products coincide.*

Proof. This follows from the definition of the transition relation of the path product. A path product has a transition between two states iff there is an execution in the step product between those states. Thus, the reachable states of the step and path products are the same. By Corollary 1, the reachable states of the step and the synchronized product are the same, which proves the lemma. □

Similarly as in the case for composition operators applying partial order semantics, local transition merging can also be combined with on-the-fly determinization. The composition operator is as follows:

Definition 22 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs, where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Let $\mathcal{L}_{st}^d$ be the determinized step product $\mathcal{L}_1 \parallel_{st}^d \cdots \parallel_{st}^d \mathcal{L}_n$. The* determinized path product *of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, denoted $\mathcal{L}_1 \parallel_{pt}^d \cdots \parallel_{pt}^d \mathcal{L}_n$, is the LTS $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = 2^{S_1} \times 2^{S_2} \times \cdots \times 2^{S_n}$,
- $I = \tau(I_1) \times \tau(I_2) \times \cdots \times \tau(I_n)$,
- $\Sigma = \{r_{\langle T_1,\ldots,T_n\rangle,\langle T'_1,\ldots,T'_n\rangle} |\ \langle T_1, \ldots, T_n \rangle \in S, \langle T'_1, \ldots, T'_n \rangle \in S\}$, *and*
- $\Delta = \{\langle T_1, \ldots, T_n \rangle, r_{\langle T_1,\ldots,T_n\rangle,\langle T'_1,\ldots,T'_n\rangle}, \langle T'_1, \ldots, T'_n \rangle |$ *there is an execution from* $\langle T_1, \ldots, T_n \rangle$ *to* $\langle T'_1, \ldots, T'_n \rangle$ *in* $\mathcal{L}_{st}^d$.$\}$

Similarly as in the case of the path product, it is assumed that the determinized path product can not be encoded compactly (to a BMC formula whose size of polynomial in the size of the system). However, with the determinized path product even the limited version, for which a polynomial encoding exists for the path product, the encoding seems hard. These difficulties are discussed in more detail in Section 5.3.

Finally, the term on-the-fly determinization is justified, i.e., it is shown that the same LTSs is obtained with the determinized products as with a technique that first determinizes the system's components (given in Definition 17) and then composes them with the standard composition operators. The situation is illustrated in Figure 2.9. This is proved as follows:

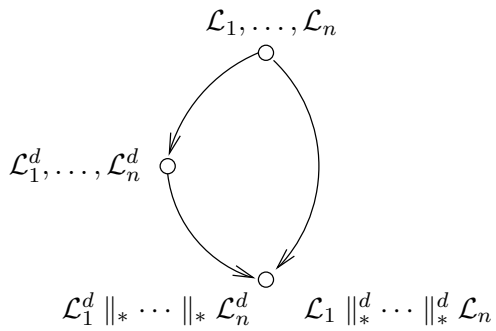

Figure 2.9: Determinization and Composition vs. Determinized Composition

Theorem 4 *Let $\parallel_*$ be one of the composition operators above, i.e., yielding the synchronized, step, process, or path product and let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Then the reachable parts of the LTSs $\mathcal{L}_*^{d'} = \mathcal{L}_1^d \parallel_* \cdots \parallel_* \mathcal{L}_n^d$ and $\mathcal{L}_*^d = \mathcal{L}_1 \parallel_*^d \cdots \parallel_*^d \mathcal{L}_n$ are the same.*

Proof. The proof is similar for any of the composition operators. Consider the case for the process product. Thus $\mathcal{L}_*^{d'}$ and $\mathcal{L}_*^d$ become $\mathcal{L}_{pr}^{d'}$ and $\mathcal{L}_{pr}^d$, respectively. Firstly, the state spaces, initial states and the alphabet for both $\mathcal{L}_{pr}^{d'}$ and $\mathcal{L}_{pr}^d$ are precisely the same.

It needs to be shown that the transition relations are the same in both LTSs. Firstly, let $t = (\langle S_1, b_1, \ldots, S_n, b_n\rangle, \langle l_1, \ldots, l_n\rangle, \langle S_1', b_1', \ldots, S_n', b_n'\rangle)$ be a transition from $\Delta_{pr}^{d'}$.

If $l_i \neq \epsilon$, then there has to be a transition $(S_i, l_i, S_i') \in \Delta_i^d$. This implies that Δ_i contains a transition from a state $s \in S_i$ labeled with l_i and S_i' is the τ-closure of all the states reachable via such transitions. However, that is also the definition of S_i' in $\mathcal{L}_{pr}^d$. If $l_i = \epsilon$, then $S_i' = S_i$. This is also the case in $\mathcal{L}_{pr}^d$. Therefore $t \in \Delta_{pr}^d$. The proof to the other direction (from Δ_{pr}^d to $\Delta_{pr}^{d'}$) is similar. Finally, since all the elements of $\mathcal{L}_{pr}^{d'}$ and $\mathcal{L}_{pr}^d$ are the same, their reachable parts are also the same. □

Corollary 3 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ the reachable states of their determinized synchronized, step, process and path products coincide.*

3 BOUNDED MODEL CHECKING

The previous chapter presents the formal theory of the system modeling formalism needed in this work. The purpose of the dissertation is to apply bounded model checking on the LTSs obtained with the presented composition operators.

Bounded model checking is a technique that seeks to answer the question of whether among the executions up to some fixed length k, there is one (or more) that violates a given specification. If such an execution is found, it is given as a counterexample. If no such execution is found, the verification engineer knows that the system behaves correctly up to the bound k.

Even though a fixed bound k is necessarily finite, it should be noted that BMC can be used to reason about some infinite executions, namely those that loop within the bound k. In such executions only k steps suffice to provide complete information of its executed actions and its reached states.

The concept of BMC as presented above does not in general limit the way the executions are represented. One could, for instance, answer the posed question of whether among the execution of length k, a violating one exists by simulating them one by one and halting if a violating one is found.

In the seminal paper [8], the executions of length k are represented symbolically as a propositional logic formula. At least two reasons for this can be identified:

- such a formula is easy to create from the transition relation of the system and its size is linear in the bound and size of the system and
- propositional satisfiability is the most studied **NP**-complete problem and there are several efficient tools for solving SAT formulas.

The ideas presented in [8] provide several interesting possibilities for research. Firstly, it is possible to cover more of the system's state space by conceiving the system's transition relation differently, i.e. applying non-standard execution semantics. In [39, 41, 40], this idea is applied to 1-safe Petri Nets, in [41] using a logic programming approach instead of propositional satisfiability. In these papers, the presented non-standard execution models correspond to the step and process semantics presented in this dissertation. However, no on-the-fly determinization and no local transition merging is applied. A non-standard execution model is also presented in [72] for Petri nets. In this paper, the presented execution model bears some resemblence to the idea of local transition merging. It is namely possible to execute several local transitions in the same time step with the goal of reducing the needed bound. However, the executed transitions do not necessarily form paths and the performance of the technique relies on a preprocessing algorithm computing a total order on the transitions of the Petri net. In [78], non-standard execution models for artificial intelligence planning problems are presented. In addition to step and process semantics, another semantics (called 1-linearization) bearing some resemblance to local transition merging is presented. However, the domain and the details of the semantics are different. The ideas presented in the papers [39, 41, 40, 72, 78] are the closest related work to this dissertation that the author is familiar with.

In addition to applying non-standard execution models, it is possible to improve on the way the properties to be verified are encoded. In [8], an encoding for LTL is presented where a temporal formula is translated to a propositional formula whose size is quadratic in the bound. That encoding is improved in [20] but it is still of the same complexity in the worst case. In [61], it is shown that a linear size translation of LTL to propositional satisfiability is possible (the translation is inspired by the translation with logic programming presented in [41]). All of these translations are based on encodings that use LTL syntax to devise a direct translation of the problem. Originally, however, model checking LTL properties is done using an algorithm where the temporal formula is translated to a Büchi automaton [87] and model checking is reduced to the question of checking language emptiness of an automaton. In the context of bounded model checking, an LTL property can also be encoded using the corresponding Büchi automaton [30, 24]. Encodings for other temporal logics than LTL have also been studied. For instance, in [75], a translation of the universal fragment of CTL is presented.

Research has also been conducted in the area of creating SAT solvers that are especially tuned for bounded model checking problems [43, 82]. Most SAT solvers (see e.g. [70, 37]) are based on variations of the Davis-Putnam procedure [29]. A propositional formula is typically solved by traversing a search tree and at each node of the tree, the solver heuristically chooses a variable and assigns to it a Boolean value determining the next subtree to be traversed. In [82, 84, 44, 89, 92] it is demonstrated that the performance of the SAT solver can be improved if the default solver heuristics is overridden with other methods based on the knowledge that the propositional formula to be solved encodes a BMC problem.

Most SAT solvers also add clauses (called *conflict clauses*) to the propositional instance to be solved during the traversal of the search tree based on the choices made so far. In [82, 84] it is shown that the symmetric nature of a BMC formula allows the default algorithm of a SAT solver to be extended so that more clauses are inferred. These additional constraints may help the solver to solve the instance faster.

When applying bounded model checking, the verifier seeks counterexamples of increasing lenght. This involves solving a sequence of propositional formulas whose structure is similar. Therefore, some of the clauses learned during one SAT check can also be used when solving successive instances [83]. This *incremental* SAT solving is applied for instance in [44, 35] with promising results.

Compared to other model checking techniques, BMC has compared favorably both in terms of the running time and memory usage even in industrial examples [9, 11, 25].

3.1 COMPLETENESS

Bounded model checking has the undesirable property that it is not complete. Since the length of the executions is limited to some bound k, it is in general not the case that all the violations of a property are caught. The BMC procedure is complete only if the verification engineer can compute a

sufficienty small bound within which a counterexample is guaranteed to be found if one exists. In general, this number depends on the property. In [8], several upper bounds are presented for different kinds of temporal formulas. For instance, for reachability properties it suffices to unroll the transition relation up to the *diameter* of the Kripke structure. The formal definition of a Kripke structure is as follows: Let $\mathcal{A}$ be a finite, non-empty set of atomic propositions.

Definition 23 *A* Kripke structure *is a tuple* $M = \langle S, I, T, \ell \rangle$ *where*

- S *is a set of states,*
- $I \subset S$ *is a set of initial states,*
- $T \subseteq S \times S$ *is a transition relation between state, and*
- $\ell : S \to \mathcal{P}(\mathcal{A})$ *is a labeling of the states with atomic propositions* $\mathcal{A}$ *which hold in that state.*

Then, the diameter of a Kripke structure is defined as follows [8]:

Definition 24 *Given a Kripke structure* M*, the diameter of* M *is the minimal number* $d \in \mathbb{N}$ *with the following property. For every sequence of states* $s_0, \cdots, s_{d+1}$ *with* $(s_i, s_{i+1}) \in T$ *for* $i \leq d$*, there is a sequence of states* $t_0, \ldots, t_l$ *where* $l \leq d$ *such that* $t_0 = s_0$ *and* $t_l = s_{d+1}$ *and* $(t_j, t_{j+1}) \in T$ *for* $j < l$*.*

Intuitively, the diameter d of a Kripke structure is the longest shortest path of between any two states. This bound is sufficient to find a counterexample of a reachability property. However, it can be too large since there is no need to consider the longest shortest path from *any* state to another state but it is sufficient for the starting state to be an initial state (called the *initialized diameter*).

However, given a symbolic representation of the Kripke structure, the diameter is not easy to compute. In [8] it is shown that given a Kripke structure and a number k, finding whether $k \leq d$ can be done by solving a Quantified Boolean Formula (QBF). Solving a sequence of QBFs for different values of k is computationally very expensive [57].

It is possible to overapproximate the diameter by another value that is easier to compute, namely the *recurrence diameter* r [8, 7]. A recurrence diameter of a graph is its longest loop-free path between any two states. Obviously, a shortest path from one state to another is loop-free, thus it holds that $d \leq r$. In the case of BMC, also with the recurrence diameter, it is sound to limit the starting states to initial states, i.e., to compute the *initialized recurrence diameter*. Unfortunately, there are graphs for which the recurrence diameter is much larger than the diameter. Consider, for example, a fully connected graph with $|S|$ states. Its diameter is one and its recurrence diameter $|S| - 1$. Biere et al. [8] show that given a number k it is possible to determine whether $k \leq r$ by solving a propositional formula whose size is quadratic in k.

The idea to compute the initialized recurrence diameter in [8] is as follows. Firstly, a formula that is a conjunction containing the constraint on initial states and k copies of the transition relation of the system is created.

However, this time an additional constraint is added. It requires that all the states along the path are distinct. When k is increased one by one, eventually this formula becomes unsatisfiable. This fact implies that all paths of length k are bound to visit some state twice, thus $k - 1$ is the recurrence diameter. Therefore, the obtained bound is sufficient for a complete SAT-based model checking procedure for reachability properties. As it is shown in Section 4.7, this technique can be easily implemented to the presented framework. However, the resulting formula is quadratic in the bound since $\mathcal{O}(k^2)$ state comparisons are performed. In [57], it is shown that the size of the formula can be reduced to $\mathcal{O}(k \log k)$ using *sorting networks*.

Another technique that can be seen as an extension of the procedure above is called *temporal induction* [81]. The goal is to prove for some k that (i) all the paths from the initial states of length k do not violate the invariant property to be verified and that (ii) along any path of length k where the invariant property holds in all the states, it is not possible from the last state of the path to reach a state where the invariant property does not hold. The first statement above can be seen as the base case of an inductive proof and the second as the induction step. The model checking algorithm increments the value of k until either a counterexample is found or both of the claims above can be proved. As stated in [81], to guarantee that the proof can be established for some k (completeness), the base case of the temporal induction has to contain the constraint that all the states of the path are distinct. In [35], it is shown that the performance of temporal induction can be increased by applying incremental satisfiability.

McMillan has also studied complete SAT-based model checking tehniques in, for instance [64, 65, 66]. An especially promising technique close to standard BMC seems to be the model checking procedure for reachability properties in [65, 66]. This approach is based on the observation that it is possible to compute a symbolic overapproximation of the states of a system reachable in one step as the *Craig interpolant* [28] of two formulas, (i) a formula encoding the initial states of the system and its transition relation one step and (ii) a formula unrolling the transition relation k steps together with the reachability property. Thus, the former formula represents executions of length one from an initial state and the latter formula execution segments of lenght k (not necessarily from an initial state) where the reachability property holds in at least one state. The paper [65] presents an iterative procedure and proves that if k is equal to the diameter of the system's state space, the procedure terminates either proving or disproving the property.

Even though the diameter suffices to guarantee completeness also with standard BMC, as discussed above, starting from symbolic representations its value is hard to compute. Indeed, if a verification engineer is unable to compute the diameter, he may have to proceed up to the potentially much larger recurrence diameter. Compared to standard BMC, McMillan's algorithm has the additional benefit that experiments have shown that it can terminate with much smaller values of k than the diameter [65].

It is also possible to use BMC together with some complete model checking procedure (for instance BDD based) to create a complete model checking algorithm. The techniques presented in [67, 18] use a SAT solver to produce from unsatisfiable propositional instances a resolution proof. The basic

idea in [67] is as follows. (i) Create a BMC formula of the system to some depth k. (ii) If this formula is satisfiable, return a counterexample, otherwise return the resolution proof of its unsatisfiability. (iii) Use the resolution proof to create an *abstraction* of the original system to which an unbounded model checking procedure is applied.

The proof of unsatisfiability of the BMC formula contains only clauses that are relevant to the property to be verified. Thus, this proof is a good source for creating the abstraction. Obviously, phase (iii) above can return a (possibly spurious) counterexample, however, necessarily longer than the initial bound k. Then, the procedure can be restarted with a larger bound.

In [18], the applied procedure is intuitively as follows: (i) Create an abstraction of the system based on the property to be verified. (ii) Apply an unbounded model checking procedure to this abstraction. (iii) If no counterexample is returned, then terminate. (iv) If a counterexample is returned, simulate it symbolically using a BMC formula.

In this simulation phase, if the counterexample turns out to be spurious, then the BMC formula together with this counterexample is unsatisfiable. The proof of this can be used to *refine* the original abstraction created in phase (i) of the algorithm above and the procedure started with this refined system. In both papers, experiments demonstrate that the results compare favorably to a standard BDD based model checker.

4 ENCODING ALGORITHMS

Chapter 2 presents the formal theory of the non-standard composition operators studied in this work. This chapter presents the translation algorithms of some of these operators, namely those that apply partial order semantics and on-the-fly determinization, to a BMC formula. The treatment of the operator applying local transition merging is postponed to Chapter 5.

The chapter starts by giving the encoding of the simplest model, interleaving executions without on-the-fly determinization (standard interleaving semantics). When the execution model is changed some of the constraints remain the same (or similar). Thus, the presentation is in such cases limited to giving the required changes. The encodings are supplemented with theorems of soundness and completeness.

4.1 GENERAL CONVENTIONS

The work presents several encodings of execution models to propositional formulas. In these presentations, certain conventions are applied. Firstly, the syntax and semantics of propositional logic are as expected. A propositional formula consists of atomic propositions and connectives. The truth value of a propositional formula f can be evaluated based on a *valuation* (in this work denoted $\mathcal{V}$) that assigns to every atomic proposition of f a truth value. A valuation $\mathcal{V}$ of f is *satisfying* or a *model* of f iff f evaluates to true when the atomic propositions are assigned the truth values from $\mathcal{V}$.

In this work, all of the Boolean formulas share the following atomic propositions:

- $in(s, \mathfrak{t})$ that is true iff local state s is reached in global state $\mathfrak{t}$,
- $ex(t, \mathfrak{t})$ that is true iff transition t is executed in execution step $\mathfrak{t}$,
- $ex(a, i, \mathfrak{t})$ that is true iff action a is executed in component $\mathcal{L}_i$ in execution step $\mathfrak{t}$,
- $ex(a, \mathfrak{t})$ that is true iff action a is executed in execution step $\mathfrak{t}$, and
- $sc(i, \mathfrak{t})$ that is true iff some transition is executed from component $\mathcal{L}_i$ in execution step $\mathfrak{t}$.

In order to be able to infer an execution of a product of components from a model of the corresponding BMC formula, only the first and second of the propositions above suffice. The additional literals are used for compactness and readability. All the encodings are given by presenting the propositional constraints one by one and in the end the structure of the complete formula (a conjunction of the constraints) is described. As seen above in the list of literals, the constraints contain propositions of the form $ex(t, \mathfrak{t})$ that in spite of the fact that they look like predicates with arguments are to be viewed as propositional atomic formulas. In the complete formula, these are grounded to actual instances of the system, for instance $ex(1, 2)$, that is on the implementation level fed to the SAT solver as a literal of the form `ex_1_2`.

In the presented constraints, every variable is bound except for the one denoting the execution step ($\mathfrak{t}$). This is due to the fact that in all the constraints, $\mathfrak{t}$ is instantiated from the first step 1 to the bound k. This grounding is further illustrated when the entire formula for a particular execution model is given.

The constraints contain conjunctions and disjunctions with instantiation rules like $\bigvee_{a_j \in \Sigma_i} ex(a_j, i, \mathfrak{t})$. In general, it might be the case that such a conjunction or disjunction turns out to be empty. The convention is applied that an empty conjunction is always true and an empty disjunction is always false.

Certain conditions are compactly encoded using a formula of the type:

$$\text{card}_0^1\{a_1, \ldots, a_n\}$$
$$\text{card}_1^1\{a_1, \ldots, a_n\}$$

The formulas above are instances of *cardinality constraints* and they are not part of standard propositional logic. Their semantics is that the former evaluates to true iff at most one of the literals $a_1, \ldots, a_n$ are true. The latter is more restrictive, it evaluates to true iff precisely one of the literals $a_1, \ldots, a_n$ evaluates to true. These cardinality constraints do not adversely affect the complexity of the resulting formula since they can be simulated using $\mathcal{O}(n)$ new variables and connectives using traditional propositional logic.

Finally, the following notational conventions are introduced for readability:

- $\Delta_i^a = \{(s, a, s') \in \Delta_i\}$, i.e., it is the set of transitions in component $\mathcal{L}_i$ labeled with a and
- $C_a = \{1 \leq i \leq n \mid a \in \Sigma_i\}$, i.e., it is the set of component indexes i such that the alphabet of component $\mathcal{L}_i$ contains a.

4.2 INTERLEAVING EXECUTIONS

This section presents an encoding of interleaving executions. It provides the basic case to which the encodings of the non-standard execution models are compared both in terms of the complexity of the encoding and the performance.

To guarantee that the execution starts from an initial state in each component, the following constraints are needed:

$$\bigwedge_{1 \leq i \leq n} \text{card}_1^1\{in(s_j, 1) \mid s_j \in I_i\}. \tag{4.1}$$

Secondly, no other states may be reached in execution state 1:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i \setminus I_i} \neg in(s_j, 1) \Big). \tag{4.2}$$

In order to obtain a sound encoding, the executed transitions have to start from a reached state. Formally:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{t_j \in \Delta_i} (ex(t_j, \mathfrak{t}) \rightarrow in(sr(t_j), \mathfrak{t})) \Big) \tag{4.3}$$

where

- $sr(t_j)$ is the source state of t_j.

At most one transition can be executed from each component. The constraint to implement this is as follows:

$$\bigwedge_{1 \leq i \leq n} \text{card}_0^1\{ex(t_j, \mathfrak{t}) \mid t_j \in \Delta_i\}. \tag{4.4}$$

In order to be able to respect the synchronization requirement in the definition of the synchronized product, a literal $ex(a_i, j, \mathfrak{t})$ is needed. It encodes the fact that action a_i is executed in component $\mathcal{L}_j$ in execution step $\mathfrak{t}$. It is defined based on the executed transitions as follows:

$$\bigwedge_{a_i \in \Sigma} \Big(\bigwedge_{j \in C_{a_i}} (ex(a_i, j, \mathfrak{t}) \leftrightarrow \bigvee_{t_k \in \Delta_j^{a_i}} ex(t_j, \mathfrak{t})) \Big) \tag{4.5}$$

where

- $C_{a_i} = \{1 \leq j \leq n \mid a_i \in \Sigma_j\}$, i.e., it is the set of component indexes j such that the alphabet of component $\mathcal{L}_j$ contains a_i and
- $\Delta_j^{a_i} = \{(s, a_i, s') \in \Delta_j\}$, i.e., it is the set of transitions in component $\mathcal{L}_j$ labeled with a_i.

The actual implementation of the synchronization requirement states that all the $ex(a_i, j, \mathfrak{t})$ literals have to have the same truth value when the action a_i is fixed but the component j is different. This is guaranteed by setting them all equivalent to the literal $ex(a_i, \mathfrak{t})$:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \Big(\bigwedge_{j \in C_{a_i}} (ex(a_i, \mathfrak{t}) \leftrightarrow ex(a_i, j, \mathfrak{t})) \Big). \tag{4.6}$$

The internal transition τ requires a different treatment since it does not have any synchronization requirements:

$$ex(\tau, \mathfrak{t}) \leftrightarrow \bigvee_{i \in C_\tau} ex(\tau, i, \mathfrak{t}). \tag{4.7}$$

At most one component can execute a τ transition:

$$\text{card}_0^1\{ex(\tau, i, \mathfrak{t}) \mid i \in C_\tau\}. \tag{4.8}$$

In the interleaving execution model, it is only possible to execute precisely one action in each step. Restricting the number of executed actions is achieved by the following cardinality constraint:

$$\text{card}_1^1\{ex(a_i, \mathfrak{t}) \mid a_i \in \Sigma\}. \tag{4.9}$$

Finally, the reached state in each component has to be correctly updated. This is managed using the following constraint:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i} (in(s_j, \mathfrak{t}+1) \leftrightarrow (in(s_j, \mathfrak{t}) \wedge \neg sc(i, \mathfrak{t})) \vee \bigvee_{t_k \in pr(s_j)} ex(t_k, \mathfrak{t})) \Big) \tag{4.10}$$

where

- $pr(s_j)$ is the set of incoming transitions to s_j.

The formula above uses the literal $sc(i, \mathfrak{t})$ that encodes the fact that some action is executed from the component $\mathcal{L}_i$. Its definition is then as follows:

$$\bigwedge_{1 \leq i \leq n} (sc(i, \mathfrak{t}) \leftrightarrow \bigvee_{a_j \in \Sigma_i} ex(a_j, i, \mathfrak{t})). \tag{4.11}$$

The complete formula is created by instantiating the constraints (4.1) to (4.11) above. Given n components and a bound k, the complete formula is denoted $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ and is of the following form:

$$IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$

where

- $IS(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ encodes the allowed initial states i.e., it is the conjunction of constraints (4.1) and (4.2) and
- $TR(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ represents the unrolling of the transition relation from state i to $i + 1$, i.e., it is the conjunction of the constraints (4.3) to (4.11) where the time variable $\mathfrak{t}$ is instantiated to the value i.

Thus, the first two constraints are used to encode the initial states of the synchronized product and the rest to describe its transition relation. In the following, the soundness of the encoding is established. As stated above, the $in(s_i, \mathfrak{t})$ literal encodes the reached component states and the $ex(a_i, \mathfrak{t})$ the executed actions. The soundness proof states that if from any satisfying valuation of $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, the true $in(s_i, \mathfrak{t})$ and $ex(a_i, \mathfrak{t})$ literals are used to construct a sequence of states and actions, an interleaving execution is obtained.

In every encoding presented in this work, the soundness proof is constructed in two phases. For the interleaving case, it is firstly shown that from satisfying valuations of $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, state-action sequences of a certain type are obtained. Then, it is shown that every step in such a sequence is a transition of the synchronized product.

In order to map the models of the formula to the executions of the system, the following definition is needed:

Definition 25 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and an integer k, let $\mathcal{V}$ be a valuation of any formula containing the following atomic propositions:*

- *for all $s_j \in S_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k + 1$, the proposition $in(s_j, \mathfrak{t})$ and*
- *for all actions $a_i \in \Sigma_1 \cup \cdots \cup \Sigma_n$ and all $1 \leq \mathfrak{t} \leq k$, the proposition $ex(a_i, \mathfrak{t})$.*

*Then, the $\mathcal{V}$-*sequence *corresponding to $\mathcal{V}$ is the sequence:*

$$\langle T_1^1, \ldots, T_n^1 \rangle \xrightarrow{L_1} \cdots \xrightarrow{L_k} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

such that each $T_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k + 1$, is a set of states and a component state $s_l \in T_i^{\mathfrak{t}}$ iff $s_l \in S_i$ and $\mathcal{V}(in(s_l, \mathfrak{t})) = true$. Each $L_{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k$, is a set of actions and an action a_l is in $L_{\mathfrak{t}}$ iff $\mathcal{V}(ex(a_l, \mathfrak{t})) = true$.

Definition 26 *Given the $\mathcal{V}$-sequence corresponding to a satisfying valuation $\mathcal{V}$ of the formula $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a valid* interleaving sequence *iff the following conditions hold:*

1. *for all* $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k+1, |T_i^{\mathfrak{t}}| = 1$ *and*
2. *for all* $1 \leq \mathfrak{t} \leq k, |L_{\mathfrak{t}}| = 1$.

Lemma 4 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}$ of the formula $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}$-sequence is an interleaving sequence.*

Proof. By definition of an interleaving sequence (Definition 26), it has to be shown that in each step of the $\mathcal{V}$-sequence, the $in(s_j, \mathfrak{t})$ literal is true for precisely one state from each component. Secondly, the $ex(a, \mathfrak{t})$ literal is true for precisely one action a.

The former condition can be established by induction over the states of the $\mathcal{V}$-sequence.

1. **Base Case.** It is true in state $\langle T_1^1, \ldots, T_n^1 \rangle$ by constraints (4.1) and (4.2).
2. **Induction Hypothesis.** Assume that the number of true $in(s_j, \mathfrak{t})$ literals is true for precisely one state from each component up to some state l.
3. **Induction Step.** Consider the state $\langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$. The desired condition follows from the fact that (4.4) limits the number of true $ex(t_j, l)$ literals to at most one in each component.

   Namely, if the number of $ex(t_j, l)$ literals is zero in component $\mathcal{L}_i$, then by constraint (4.5), the number of true $ex(a_j, i, l)$ literals is zero and thus, by constraint (4.11), the literal $sc(i, l)$ is false. Then however, constraint (4.10) reduces to:

   $$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow in(s_j, l)).$$

   By the formula above, the $in(s_j, l+1)$ literal is true for precisely the same states than in the state l, the number of which is one by induction hypothesis.

   If the number of true $ex(t_j, l)$ literals is one in component $\mathcal{L}_i$, then by constraint (4.5), there is one true $ex(a_j, i, l)$ literal. Then, by constraint (4.11), the literal $sc(i, l)$ is true and constraint (4.10) reduces to:

   $$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow \bigvee_{t_k \in pr(s_j)} ex(t_k, l)).$$

   Since $ex(t_k, l)$ literal is true for precisely one transition, the $in(s_j, l+1)$ literal is true for precisely the unique target state of that transition.

The latter condition (that the $ex(a, \mathfrak{t})$ literal is true for precisely one action a) follows directly from constraint (4.9). □

Given an interleaving sequence, let for all $1 \leq i \leq n$ and $1 \leq \mathfrak{t} \leq k+1$, $s_i^{\mathfrak{t}}$ be the single element of $T_i^{\mathfrak{t}}$. Furthermore, for all $1 \leq i \leq k$, let l_i be the single element of L_i.

Definition 27 *An interleaving sequence of formula $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the synchronized product $\mathcal{L}_{il}$ of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, iff the following conditions hold:*

1. *for all components $1 \leq i \leq n$, $s_i^1 \in I_i$ and*
2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle s_1^{\mathfrak{t}}, \ldots, s_n^{\mathfrak{t}} \rangle \xrightarrow{l_{\mathfrak{t}}} \langle s_1^{\mathfrak{t}+1}, \ldots, s_n^{\mathfrak{t}+1} \rangle \in \Delta_{il}$.*

The soundness of the encoding is then established as follows:

Theorem 5 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}$ of the formula $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}$-sequence is an execution of the synchronized product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$.*

Proof. By Lemma 4, any $\mathcal{V}$-sequence of $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an interleaving sequence. Now it has to be established that the interleaving sequence starts from an initial state and that every one of its steps is a transition of the synchronized product. To justify that a particular step in the $\mathcal{V}$-sequence is a transition of the synchronized product, knowledge about the executed transitions is needed. These are encoded using the $ex(t_j, \mathfrak{t})$ literals ($\mathcal{V}(ex(t_j, \mathfrak{t})) =$ true iff transition t_j is executed in time step $\mathfrak{t}$). The proof is by induction over the states of the interleaving sequence.

1. **Base Case.** Due to the element $IS(\mathcal{L}_1, \ldots, \mathcal{L}_k)$ (conjunction of constraints (4.1) and (4.2)) in $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, in any $\mathcal{V}$-sequence the state in component $\mathcal{L}_i$ such that $\mathcal{V}(in(s_j, 1)) =$ true has to be an initial state. Thus, the tuple of states obtained is an initial state of the synchronized product.

2. **Induction Hypothesis.** Assume that up to some state l, all the steps in the interleaving sequence are transitions of the synchronized product.

3. **Induction Step.** Consider the step $\langle s_1^l, \ldots, s_n^l \rangle \xrightarrow{l_l} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$. It is shown that:

   (a) if l_l is visible, then every component i such that $l_l \in \Sigma_i$ executes exactly one transition labeled l_l,

   (b) if $l_l = \tau$, then exactly one component executes exactly one transition labeled τ,

   (c) all the executed transitions are labeled l_l,

   (d) the executed transitions start from reached component states, and

   (e) the state $\langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$ is a state of the synchronized product that is reached from the state $\langle s_1^l, \ldots, s_n^l \rangle$ by executing the action l_l.

The first condition concerning visible actions is established as follows. Firstly, constraint (4.6) states that if $\mathcal{V}(ex(l_l, l)) = \text{true}$, then it holds that $\mathcal{V}(ex(l_l, i, l)) = \text{true}$ for all the components $\mathcal{L}_i$ having l_l in their alphabet. However, then by constraint (4.5) every such component has to execute a transition labeled l_l.

The second condition concerning the internal action τ is established by constraint (4.8) limiting the number of true literals in the literal set $\{ex(\tau, i, l) \mid i \in C_\tau\}$ to at most one. Constraint (4.5), on the other hand, allows an execution of a transition labeled τ in component $\mathcal{L}_i$ in step l iff $ex(\tau, i, l)$ is true. In both cases (visible / internal action) the fact that exactly one transition is executed is guaranteed by constraint (4.4).

The third condition is guaranteed by constraint (4.5) since it requires that if a transition labeled $a_i \neq l_l$ is executed, then the literal $ex(a_i, j, l)$ has to be true for the component $\mathcal{L}_j$ containing the transition. Then, however, by constraint (4.6), the literal $ex(a_i, l)$ has to be true and constraint (4.9) is violated.

The fourth condition that the executed transitions start from reached states is guaranteed by (4.3).

Finally, it needs to be established that the state $\langle s_1^{l+1}, \ldots, s_n^{l+1}\rangle$ is of correct form, i.e., that constraints (4.10) and (4.11) do not allow unintended models. The treatment is divided to two cases, components that are idle and components that execute a transition in step l. Keeping in mind the definition of the $sc(i, \mathfrak{t})$ literal, then for the former the constraint encoding control flow becomes:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow in(s_j, l)).$$

Therefore, idle components remain in the same state. This adheres to the definition of the synchronized product. For components executing a transition the constraint becomes:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow \bigvee_{t_k \in pr(s_j)} ex(t_k, l)).$$

Thus, the control flow in the global state $l+1$ reaches local states that are the target states of executed transitions. Thus, the reached global state is reachable in the synchronized product from $\langle s_1^l, \ldots, s_n^l\rangle$ by executing action l_l. □

Theorem 6 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the synchronized product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is a $\mathcal{V}$-sequence of some satisfying valuation $\mathcal{V}$ of $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. The proof presents a mapping from an execution to a valuation and shows that if the truth values of the literals are assigned according to this mapping, all the constraints of $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ are satisfied. Any execution of the synchronized product can be presented in the form:

$$\langle s_1^1, \ldots, s_n^1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_k} \langle s_1^{k+1}, \ldots, s_n^{k+1}\rangle.$$

The mapping $\mathcal{V}$ is as follows:

- $\mathcal{V}(in(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $s_i^{\mathfrak{t}} = s_j$,
- $\mathcal{V}(ex(a_i, \mathfrak{t})) =$ true iff $l_{\mathfrak{t}} = a_i$,
- If $l_{\mathfrak{t}} \in \Sigma \setminus \{\tau\}$, the following conditions hold:
  1. $\mathcal{V}(ex(t_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $(s_i^{\mathfrak{t}}, l_{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) = t_j$,
  2. $\mathcal{V}(ex(a_i, j, \mathfrak{t})) =$ true iff $l_{\mathfrak{t}} = a_i$, and
  3. $\mathcal{V}(sc(i, \mathfrak{t})) =$ true iff $l_{\mathfrak{t}} \in \Sigma_i$.
- If $l_{\mathfrak{t}} = \tau$, the following conditions hold:
  1. $\mathcal{V}(ex(t_j, \mathfrak{t})) =$ true iff $(s_i^{\mathfrak{t}}, l_{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) = t_j$ and there is no $m < i$ such that $(s_m^{\mathfrak{t}}, l_{\mathfrak{t}}, s_m^{\mathfrak{t}+1}) \in \Delta_m$, [1]
  2. $\mathcal{V}(ex(\tau, i, \mathfrak{t})) =$ true iff $(s_i^{\mathfrak{t}}, l_{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) \in \Delta_i$ and there is no $m < i$ such that $(s_m^{\mathfrak{t}}, l_{\mathfrak{t}}, s_m^{\mathfrak{t}+1}) \in \Delta_m$, and
  3. $\mathcal{V}(sc(i, \mathfrak{t})) =$ true iff $(s_i^{\mathfrak{t}}, l_{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) \in \Delta_i$ and there is no $m < i$ such that $(s_m^{\mathfrak{t}}, l_{\mathfrak{t}}, s_m^{\mathfrak{t}+1}) \in \Delta_m$.

The proof is by induction over the steps $\mathfrak{t}$ in $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.

1. **Base Case.** For all $1 \leq i \leq n$, the states s_i^1 are elements of I_i. Thus, the constraints (4.1) and (4.2) are satisfied.

2. **Induction Hypothesis.** Assume that every conjunct is satisfied in formula $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ up to the state l.

3. **Induction Step.** Consider the step $\langle s_1^l, \ldots, s_n^l \rangle \xrightarrow{l_l} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$. The goal is to show that mapping $\mathcal{V}$ satisfies all the conjuncts of the formula $TR(\mathcal{L}_1, \ldots, \mathcal{L}_n, l)$. By definition of the synchronized product, if a component executes a transition t_j, then its source state is reached and constraint (4.3) is satisfied. In any interleaving execution, each component can execute at most one transition. Therefore, constraint (4.4) is satisfied. Component $\mathcal{L}_i$ executes an action a iff it executes a transition labeled a. Therefore, constraint (4.5) is also satisfied.

   By definition of the synchronized product, if a visible action is executed, then all components having that action in their alphabet have to execute one transition labeled with that action. Therefore, constraint (4.6) is satisfied. The internal action τ is executed iff exactly one component executes a τ transition. Constraint (4.7) is therefore satisfied. In any interleaving execution in any step, at most one component may execute a τ transition and constraint (4.8) is satisfied. The number of executed actions is precisely one in every interleaving execution. Thus, constraint (4.9) is satisfied.

[1] In general, it can be the case that there are several possible choices for the executed τ transition. This is a design choice of setting the $ex(t_j, l)$ literal true for a transition in a component with the smallest index. Another choice could be made as well.

If a component does not execute any transitions, the control flow in that component remains in the same state. Thus, constraint (4.10) is satisfied for such idle components. If a component executes a transition, then the control flow moves to the target state of that transition and constraint (4.10) is also satisfied for these scheduled components. Finally, the literal $sc(i, \mathfrak{t})$ is true iff component $\mathcal{L}_i$ executes a transition. Thus, constraint (4.11) is satisfied. □

The encoding presented above is thus sound and complete with respect to the interleaving execution model. When the encoding for step and process models is considered, it is seen that they are obtained from the encoding of interleaving executions with small modifications.

4.3 STEP EXECUTIONS

The difference between the executions of the step product and the synchronized product is that the former is more general, it contains additional executions characterized by the fact that several independent actions are executed simultaneously. Therefore, given n components and a bound k, the encoding formula for the step product, denoted $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, has to be weaker than $IL(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, i.e., have more models.

The simplest way to obtain more models is to omit some constraints. Indeed, in the encoding of the interleaving model, constraint (4.9) limits the executed actions in each step to precisely one. If this is omitted, then several actions can be executed in each step but the system is also able to idle. The step model is obtained when the cardinality constraint in constraint (4.9) is replaced with constraint (4.12), a disjunction:

$$\bigvee_{a_i \in \Sigma} ex(a_i, \mathfrak{t}). \tag{4.12}$$

In addition, the step product allows several components to execute an internal τ action. To allow this is easy, constraint (4.8) is omitted. Given n components and a bound k, the formula encoding the executions of their step product of length k is of the form:

$$ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR_s(\mathcal{L}_1, \ldots, \mathcal{L}_n, i).$$

The first conjunct, $IS(\mathcal{L}_1, \ldots, \mathcal{L}_n)$, is precisely the same as in the case of the synchronized product. The latter part, $TR_s(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$, encodes again the transition relation from state i to $i + 1$. It is otherwise the same as $TR(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ but constraint (4.9) is replaced by constraint (4.12) and constraint (4.8) is omitted.

The formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ can be proved sound and complete with respect to the executions of the step product. The proofs apply the concept of a $\mathcal{V}_s$-sequence defined below. The definition is a modified version of the one used in interleaving executions recognizing the fact that the executed actions in the step product are tuples.

Definition 28 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and an integer k, let $\mathcal{V}_s$ be a valuation of any formula containing the following atomic propositions:*

- *for all $s_j \in S_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k+1$, the proposition $in(s_j, \mathfrak{t})$ and*
- *for all action–component pairs $(a, i) \in \bigcup_{1 \leq i \leq n}(\Sigma_i \times \{i\})$ and all $1 \leq \mathfrak{t} \leq k$, the proposition $ex(a, i, \mathfrak{t})$.*

*Then, the $\mathcal{V}_s$-*sequence *corresponding to $\mathcal{V}_s$ is the sequence:*

$$\langle T_1^1, \ldots, T_n^1 \rangle \overset{\langle L_1^1, \ldots, L_n^1 \rangle}{\rightarrow} \cdots \overset{\langle L_1^k, \ldots, L_n^k \rangle}{\rightarrow} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

such that each $T_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k+1$, is a set of states and a component state $s_l \in T_i^{\mathfrak{t}}$ iff $s_l \in S_i$ and $\mathcal{V}(in(s_l, \mathfrak{t})) = true$. Each $L_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k$, is a set of actions and an action a_j is in $L_i^{\mathfrak{t}}$ iff $\mathcal{V}_s(ex(a_j, i, \mathfrak{t})) = true$.

The proof of the soundness of the encoding is established in two phases similarly as in the case of interleaving executions. The proof uses the following concept:

Definition 29 *Given the $\mathcal{V}_s$-sequence corresponding to a valuation $\mathcal{V}_s$ of the formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a valid* step-sequence *iff the following conditions hold:*

1. *for all $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k+1$, $|T_i^{\mathfrak{t}}| = 1$ and*
2. *for all $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k$, $|L_i^{\mathfrak{t}}| \leq 1$.*

Lemma 5 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}_s$ of the formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}_s$-sequence is a step-sequence.*

Proof. The proof is similar than in the case of interleaving executions. It has to be shown that in each step of the $\mathcal{V}_s$-sequence, the $in(s_j, \mathfrak{t})$ literal is true for precisely one state from each component. Secondly, the $ex(a, i, \mathfrak{t})$ literal is true for precisely one action a in the alphabet of component $\mathcal{L}_i$.

The proof of the former condition is precisely the same as in Theorem 4 for the interleaving model.

The latter condition on the cardinality of the sets L_i^j is guaranteed by the fact that constraint (4.4) allows the number of true $ex(t_m, \mathfrak{t})$ literals from any fixed component to be at most one. If every one of those literals is false, then by constraint (4.5) every $ex(a, i, \mathfrak{t})$ literal for that component is also false. If precisely one of the $ex(t_m, \mathfrak{t})$ literals is true, then by the constraint (4.5), there is precisely one $ex(a, i, \mathfrak{t})$ literal for that component that is true. □

Given a step-sequence of length k, let for all $1 \leq i \leq n$ and $1 \leq \mathfrak{t} \leq k+1$, $s_i^{\mathfrak{t}}$ be the single element of $T_i^{\mathfrak{t}}$. Furthermore, for all $1 \leq i \leq n$ and $1 \leq \mathfrak{t} \leq k$, since $|L_i^{\mathfrak{t}}| \leq 1$, let $l_i^{\mathfrak{t}}$ denote either the single element of $L_i^{\mathfrak{t}}$ or if $L_i^{\mathfrak{t}} = \emptyset$, let $l_i^{\mathfrak{t}}$ denote ϵ.

Definition 30 *A step-sequence of formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the step product $\mathcal{L}_{st}$ of $\mathcal{L}_1, \ldots, \mathcal{L}_n$ iff the following conditions hold:*

1. *for all components* $1 \leq i \leq n$, $s_i^1 \in I_i$ *and*

2. *for all* $1 \leq \mathfrak{t} \leq k$, $\langle s_1^{\mathfrak{t}}, \ldots, s_n^{\mathfrak{t}} \rangle \stackrel{\langle l_1^{\mathfrak{t}}, \ldots, l_n^{\mathfrak{t}} \rangle}{\longrightarrow} \langle s_1^{\mathfrak{t}+1}, \ldots, s_n^{\mathfrak{t}+1} \rangle \in \Delta_{st}$.

Theorem 7 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs. Given any satisfying valuation* $\mathcal{V}_s$ *of the formula* $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$*, then the corresponding* $\mathcal{V}_s$*-sequence is an execution of the step product of* $\mathcal{L}_1, \ldots, \mathcal{L}_n$.

Proof. By Lemma 5, given any $\mathcal{V}_s$-sequence of $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a step-sequence. Now it has to be established that the step-sequence starts from an initial state and that every one of its steps is a transition of the step product. To justify that a particular step in the $\mathcal{V}_s$-sequence is a transition of the step product, knowledge about the executed transitions is needed. These are encoded using the $ex(t_j, \mathfrak{t})$ literals ($\mathcal{V}_s(ex(t_j, \mathfrak{t})) =$ true iff transition t_j is executed in time step $\mathfrak{t}$). The proof is by induction over the states of the step-sequence.

1. **Base Case.** Due to the element $IS(\mathcal{L}_1, \ldots, \mathcal{L}_k)$ (conjunction of constraints (4.1) and (4.2)) in $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, in any $\mathcal{V}_s$-sequence the state in component $\mathcal{L}_i$ such that $\mathcal{V}_s(in(s_j, 1)) =$ true has to be an initial state. Thus, the tuple of states obtained is an initial state of the step product.

2. **Induction Hypothesis.** Assume that up to some state l, all the steps in the step-sequence are transitions of the step product.

3. **Induction Step.** Consider the step

$$\langle s_1^l, \ldots, s_n^l \rangle \stackrel{\langle l_1^l, \ldots, l_n^l \rangle}{\longrightarrow} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle.$$

   It is shown that:

   (a) the tuple $\langle l_1^l, \ldots, l_n^l \rangle$ can not be $\langle \epsilon, \ldots, \epsilon \rangle$,

   (b) if l_i^l is visible, then for every component $\mathcal{L}_j$ such that $l_i^l \in \Sigma_j, l_j^l = l_i^l$,

   (c) if $l_i^l \neq \epsilon$, then component $\mathcal{L}_i$ executes exactly one transition and that transition is labeled l_i^l,

   (d) if $l_i^l = \epsilon$, then the component $\mathcal{L}_i$ executes no transition,

   (e) the executed transitions start from reached component states, and

   (f) the state $\langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$ is a state of the step product reached from the state $\langle s_1^l, \ldots, s_n^l \rangle$ by executing the action $\langle l_1^l, \ldots, l_n^l \rangle$.

   The first condition, disabling idling in the step product, is guaranteed by the new constraint (4.12). Since at least one $ex(a, l)$ has to be true, at least one $ex(a, i, l)$ literal has to be true.

   The second condition is guaranteed by constraint (4.6) implementing the synchronization requirement.

   The third condition in guaranteed by constraint (4.5) forcing some transition with the correct label to be executed and constraint (4.4) limiting the number of executed transition to at most one.

The fourth condition is guaranteed solely by constraint (4.5).

The fifth condition is guaranteed by constraint (4.3).

The final condition concerning the state $\langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle$ is established by analysis of constraint (4.10) in precisely the same way as in the proof of Condition 3e in Theorem 5. The control flow stays in the same local state for components that do not execute any actions and if a component executes an action, then the control flow moves to the target state the executed local transition. □

Theorem 8 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the step product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is a $\mathcal{V}_s$-execution of some satisfying valuation $\mathcal{V}_s$ of $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. The proof is similar as in the case of interleaving executions. Any step execution of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ of length k is of the form:

$$\langle s_1^1, \ldots, s_n^1 \rangle \stackrel{\langle l_1^1, \ldots, l_n^1 \rangle}{\longrightarrow} \cdots \stackrel{\langle l_1^k, \ldots, l_n^k \rangle}{\longrightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

The literals of $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ are mapped using a mapping $\mathcal{V}_s$ in a similar fashion as in the proof of Theorem 6, namely:

- $\mathcal{V}_s(in(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $s_i^{\mathfrak{t}} = s_j$,
- $\mathcal{V}_s(ex(t_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $(s_i^{\mathfrak{t}}, l_i^{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) = t_j$,
- $\mathcal{V}_s(ex(a_i, j, \mathfrak{t})) =$ true iff $l_i^{\mathfrak{t}} = a_i$,
- $\mathcal{V}_s(ex(a_i, \mathfrak{t})) =$ true iff for some $1 \leq j \leq n$, $l_j^{\mathfrak{t}} = a_i$, and
- $\mathcal{V}_s(sc(i, \mathfrak{t})) =$ true iff $l_i^{\mathfrak{t}} \neq \epsilon$.

The proof is by induction over the steps $\mathfrak{t}$ in $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.

1. **Base Case.** Any execution starts from some initial state. Thus, constraints (4.1) and (4.2) are satisfied.

2. **Induction Hypothesis.** Assume that no conjunct is violated in formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ up to the state l.

3. **Induction Step.** Consider the step

$$\langle s_1^l, \ldots, s_n^l \rangle \stackrel{\langle l_1^l, \ldots, l_n^l \rangle}{\longrightarrow} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle.$$

The goal is to show that mapping $\mathcal{V}_s$ satisfies all the conjuncts of the formula $TR_s(\mathcal{L}_1, \ldots, \mathcal{L}_n, l)$. By definition of the step product, if a component executes a transition t_i, then its source state is reached and condition (4.3) is satisfied. In any step execution, each component can execute at most one transition. Hence, constraint (4.4) is satisfied. Component $\mathcal{L}_i$ executes action a iff it executes a transition labeled a. Therefore, constraint (4.5) is also satisfied.

By definition of the step product, if a visible action is executed, then all components with that action in their alphabet have to execute a transition labeled with that action. Therefore, constraint (4.6) is satisfied. The internal action τ is executed iff at least one component executes a τ transition. Constraint (4.7) is therefore satisfied.

The definition of the step product does not allow the executed tuple of actions to be $\langle \epsilon, \ldots, \epsilon \rangle$. Thus, all the components of the system can not be idle and constraint (4.12) is satisfied.

If a component does not execute any transitions, the control flow in that component remains in the same state. Thus, constraint (4.10) is satisfied for such idle components. If a component executes a transition, then the control flow moves to the target state of that transition and constraint (4.10) is also satisfied for these scheduled components. Finally, the literal $sc(i, \mathfrak{t})$ is true iff component $\mathcal{L}_i$ executes some transition. Thus, constraint (4.11) is satisfied. □

4.4 PROCESS EXECUTIONS

Even though the process product of a system can have more states than the step product, its executions (when the truth values associated with the states are omitted) are a subset of the executions of the step product. Namely those executions that fulfill the process condition given in Definition 13. Thus, when the bounded executions of length k of the process product of a system with n components are encoded, the resulting formula, which is denoted $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, has to be stronger than $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. Indeed, the correct encoding is obtained by adding two constraints.

The additional constraints limit the models of $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ to executions that fulfill the process condition. For visible actions, the condition states that if an action is executed in step $\mathfrak{t}+1$, then some component having that action in its alphabet has to be scheduled in step $\mathfrak{t}$. The encoding for visible actions is as follows:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} (ex(a_i, \mathfrak{t}+1) \rightarrow \bigvee_{j \in C_{a_i}} sc(j, \mathfrak{t})). \tag{4.13}$$

For τ transitions the treatment is different. The definition of the process condition states that if a τ transition is executed in the step $\mathfrak{t}+1$, the component containing the transition is scheduled in step $\mathfrak{t}$:

$$\bigwedge_{i \in C_\tau} (ex(\tau, i, \mathfrak{t}+1) \rightarrow sc(i, \mathfrak{t})). \tag{4.14}$$

Given n components and a bound k, the formula to encode process executions of length k, denoted $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, is of the following form:

$$PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR_s(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$
$$\bigwedge_{1 \leq i < k} PRC(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$

where

- $IS(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is the constraint on the initial state,
- $TR_s(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ encodes the transition relation from state i to $i+1$, and
- $PRC(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ is the constraint to encode the process condition, i.e., it is the conjunction of constraints (4.13) and (4.14) with the variable $\mathfrak{t}$ instantiated to i.

Notice that in $PRC(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$, the maximum value of i is $k-1$.

Soundness and completeness are established as follows:

Definition 31 *Let $\mathcal{L}_{st}$ be the step product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$. A $\mathcal{V}_s$-sequence of formula $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the process product of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ iff the following conditions hold:*

1. *for all components $1 \leq i \leq n$, $s_i^1 \in I_i$,*
2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle s_1^{\mathfrak{t}}, \ldots, s_n^{\mathfrak{t}} \rangle \overset{\langle l_1^{\mathfrak{t}}, \ldots, l_n^{\mathfrak{t}} \rangle}{\rightarrow} \langle s_1^{\mathfrak{t}+1}, \ldots, s_n^{\mathfrak{t}+1} \rangle \in \Delta_{st}$, and*
3. *this step execution fulfills the process condition.*

Theorem 9 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}_s$ of the formula $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}_s$-sequence is an execution of the process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$.*

Proof. Firstly, since the formula $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is stronger than the formula $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ every one of its $\mathcal{V}_s$-sequences is a step execution. However, the additional constraints (4.13) and (4.14) are verbatim translations of the the process condition given in Definition 13. Thus, a model of $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is a step executions that fulfills the process condition and by Lemma 1 an execution of the process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$. □

Completeness is established in the same way:

Theorem 10 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, the corresponding step execution is a $\mathcal{V}_s$-sequence of some satisfying valuation $\mathcal{V}_s$ of $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. Given any execution of the process product, an execution of the step product is obtained by omitting the truth values from the global states. If the truth values of the literals of $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ are defined using the mapping of Theorem 8 a valuation $\mathcal{V}_s$ is obtained. This valuation satisfies all the constraints in $ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. These constraints are also present in $PR(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ and are thus satisfied.

Remains to show that $\mathcal{V}_s$ also satisfies constraints (4.13) and (4.14). This follows from the fact that the step executions obtained from executions of the process product fulfill the process condition (Definition 13). Keeping in mind the fact that the literal $sc(i, \mathfrak{t})$ is true iff component $\mathcal{L}_i$ executes a transition in step $\mathfrak{t}$, the new constraints for step $\mathfrak{t}$ evaluate to true iff the process condition for that step holds. Thus, these constraints are satisfied. □

4.5 INCLUDING ON-THE-FLY DETERMINIZATION

This section presents the encoding for the products presented in Sections 4.2, 4.3 and 4.4 when on-the-fly determinization is added. They key differences brought about by on-the-fly determinization are the following:

- the execution of an action corresponds to the execution of *every* transition that is labeled with the action and whose source state is in the reached set of states in a particular component and
- the τ transitions of the components are compressed away.

As a consequence of on-the-fly determinization, each component can after a sequence of actions be in a set of states instead of being in a single state. The encoding assumes that the LTSs do not have loops containing only τ transitions involving more than one state (called *non-trivial τ loops*). The reason for this assertion is rather technical, however necessary, and it is elaborated after the encoding for determinized interleaving executions is presented. To guarantee the non-existence of non-trivial τ loops, a preprocessing algorithm is performed on all the components of the system. The algorithm uses the concept of a maximal strongly connected component defined as follows:

Definition 32 *Let $G = (V, E)$ be a directed graph with vertices V and edges $E \subseteq V \times V$. A* strongly connected component *C of G is a set of vertices $C \subseteq V$ such that for any two vertices $v, v' \in C$, there is a path from v to v' in G.*

Definition 33 *A strongly connected component C of $G = (V, E)$ is* maximal *iff for any vertice $v \in V \setminus C$, the set $C \cup \{v\}$ is not a strongly connected component.*

The preprocessing algorithm is then as follows. Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be the components of the analyzed system. For all $1 \leq i \leq n$:

1. Compute all the maximal strongly connected components $C_1, \ldots, C_j$ of $\mathcal{L}_i$ restricted to τ transitions.
2. Replace $\mathcal{L}_i = (S_i, I_i, \Sigma_i, \Delta_i)$ with $\mathcal{L}'_i = (S'_i, I'_i, \Sigma'_i, \Delta'_i)$ where
   - $S'_i = \{s'_1, \ldots, s'_j\}$,
   - $I'_i = \{s'_k \in S'_i \mid C_k \cap I_i \neq \emptyset\}$,
   - $\Sigma'_i = \Sigma_i$, and
   - $\Delta'_i = \{(s'_k, a, s'_l) \in S'_i \times \Sigma'_i \times S'_i \mid$ there are $s \in C_k, a \in \Sigma_i,$ and $s' \in C_l$ such that $(s, a, s') \in \Delta_i.\}$

A strongly connected component of an LTS restricted to τ transitions is referred to as a *τ-component*. The above construction takes the original LTS and replaces each MSCC C_i with a single representative state s'_i. The incoming and outgoing transitions of a particular representative state are then the union of the incoming and outgoing transitions of the states forming the corresponding MSCC. Let $repr()$ be a function from S_i to S'_i such that

$repr(s_j) = s'_k$ iff $s_j \in C_k$. It holds that if the original system reaches a global state $\langle s_1, \ldots, s_n \rangle$ with some execution, then the modified system contains an execution with the same visible actions that reaches the global state where each of the local states is replaced by its τ-component, i.e., the global state $\langle repr(s_1), \ldots, repr(s_n) \rangle$. In addition, as is shown in Section 4.6, even with the preprocessing above, it is still possible to verify reachability properties of the *original* system.

As in the case of the non-determinized versions, the formulas are given fully for the interleaving model and for the step and process execution models, the presentation is limited to the required changes to the interleaving model. For simplicity, it is assumed that when referring to an LTS with notation like $\mathcal{L}_i$, the LTS in question does not contain non-trivial τ loops. If an LTS under discussion is not preprocessed, then this is explicitly mentioned.

4.5.1 Determinized Interleaving Executions

The differences between the determinized synchronized product and the standard synchronized product are reflected already in the constraint encoding the initial states. Whereas in the execution models not applying on-the-fly determinization, each component is required to start from one initial state, in the determinized version each component starts from the set of states formed by the τ-closure of every initial state. The set can be easily computed statically and encoded in the following formula:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in \tau(I_i)} in(s_j, 1) \Big). \tag{4.15}$$

where

- $\tau(I_i)$ denotes the τ-closure of the set I_i.

Secondly, no other states may be reached in execution state 1:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i \setminus \tau(I_i)} \neg in(s_j, 1) \Big). \tag{4.16}$$

In the determinized version, the executed action determines the executed transitions uniquely. To achieve this, the implication in constraint (4.3) is changed into an equivalence as follows:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{a_j \in \Sigma_i \setminus \{\tau\}} \Big(\bigwedge_{t_k \in \Delta_i^{a_j}} (ex(t_k, \mathfrak{t}) \leftrightarrow (in(sr(t_k), \mathfrak{t}) \wedge ex(a_j, \mathfrak{t})))\Big)\Big) \tag{4.17}$$

where

- $sr(t_k)$ is the source state of t_j and
- $\Delta_i^{a_j} = \{(s, a_j, s') \in \Delta_i\}$, i.e., the set of transitions in component i labeled with a_j.

In the non-determinized version, the encoding proceeds by giving a constraint limiting the executed transitions in each component to at most one. In the determinized version, this constraint is not sound. The next constraint is the same as (4.5). The intuition is that some action a is executed in component $\mathcal{L}_j$ iff at least one transition from that component labeled a is executed:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \Big(\bigwedge_{j \in C_{a_i}} (ex(a_i, j, \mathfrak{t}) \leftrightarrow \bigvee_{t_k \in \Delta_j^{a_i}} ex(t_k, \mathfrak{t})) \Big). \tag{4.18}$$

- $C_{a_i} = \{1 \leq j \leq n \mid a_i \in \Sigma_j\}$, i.e., it is the set of component indexes j such that the alphabet of component $\mathcal{L}_j$ contains a_i.

Also in the determinized model, the synchronization between components needs to be respected. Synchronization is implemented as follows:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \Big(\bigwedge_{j \in C_{a_i}} (ex(a_i, \mathfrak{t}) \leftrightarrow ex(a_i, j, \mathfrak{t})) \Big). \tag{4.19}$$

At most one action can be executed in each step. The cardinality constraint to implement this, and also disable idling, is as follows:

$$\mathrm{card}_1^1 \{ex(a_i, \mathfrak{t}) \mid a_i \in \Sigma \setminus \{\tau\}\}. \tag{4.20}$$

What remains is the definition of the progress of control flow. Compared to the non-determinized model, the definition has to the contain the possibility of reaching a state due to it being in the τ-closure of some state reached by executing its incoming transition. To simplify the presentation, the following notational convention is applied:

Definition 34 *Let $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$ be an LTS and $s_j \in S_i$. Then, the set $ps_\tau(s_j) = \{s_k \in S_i \setminus \{s_j\} \mid (s_k, \tau, s_j) \in \Delta_i\}$, i.e., it is the set of states from which s_j is reachable by executing a single τ transition that is not a self-loop.*

The formula is then as follows:

$$\begin{aligned} \bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i} (in(s_j, \mathfrak{t}+1) \;\leftrightarrow\; \Big(& \bigvee_{t_k \in pr(s_j) \setminus \Delta_i^\tau} ex(t_k, \mathfrak{t}) \vee \\ & (sc(i, \mathfrak{t}) \wedge \bigvee_{s_l \in ps_\tau(s_j)} in(s_l, \mathfrak{t}+1)) \vee \\ & (in(s_j, \mathfrak{t}) \wedge \neg sc(i, \mathfrak{t}))))\Big). \end{aligned} \tag{4.21}$$

The definition above applies the $sc(i, \mathfrak{t})$ literal whose definition is similar to that of the non-determinized model, namely:

$$\bigwedge_{1 \leq i \leq n} (sc(i, \mathfrak{t}) \leftrightarrow \bigvee_{a_j \in \Sigma_i \setminus \{\tau\}} ex(a_j, i, \mathfrak{t})). \tag{4.22}$$

The determinized interleaving executions are encoded by instantiating the constraints above. The complete formula, denoted $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is of the following form:

$$IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS_d(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$

where

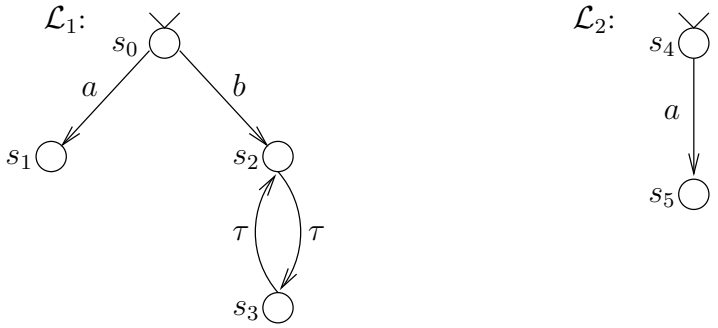

Figure 4.1: Purpose of Preprocessing

- $IS_d(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ encodes the initial states, i.e., it is a conjunction of constraints (4.15) and (4.16) and
- $TR^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ encodes the transition relation from step i to $i+1$ and is a conjunction of constraints (4.17) to (4.22) where $\mathfrak{t}$ is instantiated to i.

Purpose of Preprocessing

Constraint (4.21) encoding the progress of control flow appears in the same form also in the determinized step and process models. That constraint is the reason why the preprocessing procedure presented in Section 4.5 is applied. Namely, if the preprocessing procedure were not applied, the resulting formula would allow unintended models. These models are such that whenever component $\mathcal{L}_i$ is scheduled in step $\mathfrak{t}$, the $in(s_j, \mathfrak{t}+1)$ literals are true for all the states s_j in some non-trivial τ-component even though its states do *not* belong to the τ-closure of a state whose incoming transition is executed.

This fact is due to the second disjunct in constraint (4.21). Assuming that component $\mathcal{L}_i$ contains a non-trivial τ-component and that $\mathcal{L}_i$ is scheduled but no transition ending in a state of the τ-component is executed, constraint (4.21) reduces to $in(s_j, \mathfrak{t}) \leftrightarrow \bigvee_{s_l \in ps_\tau(s_j)} in(s_l, \mathfrak{t})$ for every state s_j in the τ-component. Then, the truth value of the literal $in(s_j, \mathfrak{t})$ for the states s_j in the τ-component can be either true or false as long as it is the same for all the states.

The situation is illustrated in Figure 4.1 that presents a system consisting of two components. The component $\mathcal{L}_1$ on the left has a non-trivial τ-component that consists of the states s_2 and s_3. Consider the determinized interleaving execution $\langle\{s_0\}, \{s_4\}\rangle, \xrightarrow{a} \langle\{s_1\}, \{s_5\}\rangle$ and a model for the formula $IL_d(\mathcal{L}_1, \mathcal{L}_2, 1)$ obtained using the mapping $\mathcal{V}$ given in the proof of Theorem 12. Since the action a is executed, then component $\mathcal{L}_1$ is scheduled. In addition, its transition labeled b is not executed. With this information, constraint (4.21) for states s_2 and s_3 in the global state 2 can be simplified to:

$$in(s_2, 2) \leftrightarrow in(s_3, 2) \text{ and}$$
$$in(s_3, 2) \leftrightarrow in(s_2, 2).$$

This however, allows besides the intended model where both the literals $in(s_2, 2)$ and $in(s_3, 2)$ are false a model where both of them are true.

By now, it has been established that if non-trivial τ-components are not preprocessed, then the BMC formula can have too many models. It should also be shown that the preprocessing algorithm removes the problem. This is shown in Theorem 11 proving the soundness of the encoding. The intuitive explanation is that the preprocessing removes the possibility of cyclic dependencies in the instances of constraint (4.21).

Since also in the determinized model of interleaving executions only one action is executed in a time step, the soundness of the encoding is established using the concept of a $\mathcal{V}$-sequence defined in Definition 25. That definition is repeated here for completeness:

Definition 35 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and an integer k, let $\mathcal{V}$ be a valuation of any formula containing the following atomic propositions:*

- *for all $s_j \in S_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k+1$, the proposition $in(s_j, \mathfrak{t})$ and*
- *for all actions $a \in \Sigma_1 \cup \cdots \cup \Sigma_n$ and all $1 \leq \mathfrak{t} \leq k$, the proposition $ex(a, \mathfrak{t})$.*

Then, the $\mathcal{V}$-sequence *corresponding to $\mathcal{V}$ is the sequence:*

$$\langle T_1^1, \ldots, T_n^1 \rangle \xrightarrow{L_1} \cdots \xrightarrow{L_k} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

such that each $T_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k+1$ is a set of states and a component state $s_l \in T_i^{\mathfrak{t}}$ iff $s_l \in S_i$ and $\mathcal{V}(in(s_l, \mathfrak{t})) = true$. Each $L_{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k$ is a set of actions and an action a_i is in $L_{\mathfrak{t}}$ iff $\mathcal{V}(ex(a_i, \mathfrak{t})) = true$.

Definition 36 *Given a $\mathcal{V}$-sequence corresponding to a satisfying valuation $\mathcal{V}$ of the formula $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a valid* determinized interleaving sequence *iff the following conditions hold:*

1. *for all $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k+1$, $|T_i^{\mathfrak{t}}| \neq \emptyset$ and*
2. *for all $1 \leq \mathfrak{t} \leq k$, $|L_{\mathfrak{t}}| = 1$.*

Lemma 6 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}$ of the formula $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}$-sequence is a determinized interleaving sequence.*

Proof. By definition of a determinized interleaving sequence (Definition 36), it has to be shown that in each step of the $\mathcal{V}$-sequence, the $in(s, \mathfrak{t})$ literal is true for at least one state from each component. Secondly, the $ex(a, \mathfrak{t})$ literal is true for precisely one action a.

The former condition can be established using induction over the states of the $\mathcal{V}$-sequence.

1. **Base Case.** It is true in state $\langle T_1^1, \ldots, T_n^1 \rangle$ since constraint (4.15) requires the $in(s_j, 1)$ to be true for those states s_j in each component that form the τ-closure of the component's initial states. The set of initial states is by definition non-empty and thus also its τ-closure.

2. **Induction Hypothesis.** Assume that the number of true $in(s_j, \mathfrak{t})$ literals is non-zero for every component up to some state l.

3. **Induction Step.** Consider the state $\langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$. The desired condition follows from analysis of constraint (4.21). Firstly, if the number of true $ex(t_i, l)$ literals is zero for every transition of a component $\mathcal{L}_i$, then constraint (4.21) reduces to:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow in(s_j, l)).$$

Thus, the $in(s_j, l+1)$ literals have the same truth value for all component states in both global states l and $l+1$. By induction hypothesis, the number of true literals in state l is non-zero.

Secondly, if the literal $ex(t_k, l)$ is true for some transitions t_k from a component, constraint (4.21) becomes:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow (\bigvee_{t_k \in pr(s_i) \setminus \Delta_i^\tau} ex(t_k, l) \vee \bigvee_{s_o \in ps_\tau(s_j)} in(s_o, l+1))).$$

Since some $ex(t_k, l)$ literal is true, the $in(s_j, l+1)$ literal is true for at least the target state of t_k.

The latter constraint (that for all $1 \leq i \leq k$, $|L_i| = 1$) is guaranteed by constraint (4.20). □

Given a determinized interleaving sequence, let for all $1 \leq i \leq k$, l_i be the single element of L_i.

Definition 37 *A determinized interleaving sequence of $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the determinized synchronized product $\mathcal{L}_{il}^d$ of $\mathcal{L}_1, \ldots, \mathcal{L}_n$ iff the following conditions hold:*

1. *for all components $1 \leq i \leq n$, $T_i^1 = \tau(I_i)$ and*

2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle T_1^{\mathfrak{t}}, \ldots, T_n^{\mathfrak{t}} \rangle \xrightarrow{l_{\mathfrak{t}}} \langle T_1^{\mathfrak{t}+1}, \ldots, T_n^{\mathfrak{t}+1} \rangle \in \Delta_{il}^d$.*

In order to demonstrate that the encoding is sound for LTSs that are preprocessed with the algorithm given in Section 4.5, the concept of a topological order is needed. It is defined as follows:

Definition 38 *Let $G = (V, E)$ be a directed acyclic graph (DAG). A* topological order *of G is an order $s_{i_1} < \cdots < s_{i_n}$ of its vertices $V = \{s_1, \ldots, s_n\}$ such that for each edge $(s_j, s_k) \in E$, $s_j < s_k$.*

The soundness of the encoding is then established as follows:

Theorem 11 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}$ of the formula $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}$-sequence is an execution of the determinized synchronized product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$.*

Proof. In the proof of Lemma 6 it is shown that any $\mathcal{V}$-sequence obtained from a valuation $\mathcal{V}$ of formula $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is a determinized interleaving sequence. Now it needs to be established that every step in this sequence is a transition of the determinized synchronized product. For this, information of the executed transitions in components is needed. This is encoded using the $ex(t_j, \mathfrak{t})$ literals ($\mathcal{V}(ex(t_j, \mathfrak{t})) =$ true iff transition t_j is executed in step $\mathfrak{t}$). The proof is by induction over the states of the $\mathcal{V}$-sequence.

1. **Base Case.** Due to constraints (4.15) and (4.16), in every valuation $\mathcal{V}$ of $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ the state $\langle T_1^1, \ldots, T_n^1 \rangle$ is the initial state of the determinized synchronized product.

2. **Induction Hypothesis.** Assume that the $\mathcal{V}$-sequence is valid determinized interleaving execution up to some step l.

3. **Induction Step.** Consider the step $\langle T_1^l, \ldots, T_n^l \rangle \xrightarrow{l_l} \langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$. The proof of the induction step is a modified version of the proof for the non-determinized version, namely:

   (a) it is shown for the executed action l_l that every component $\mathcal{L}_i$ such that $l_l \in \Sigma_i$ executes a transition labeled l_l,

   (b) all the executed transitions are labeled l_l,

   (c) it is shown that the executed transitions start from reached component states, and that every such transition labeled l_l is executed, and

   (d) it is shown that the state $\langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$ is the state of the determinized synchronized product that is reached from the state $\langle T_1^l, \ldots, T_n^l \rangle$ by executing the action l_l.

   The first condition that every component with transitions labeled l_l executes one is guaranteed by (4.18).

   The second condition that no transitions with another label are executed is established as follows. If $ex(t_j, l)$ is true for some transition not labeled with l_l, the literal $ex(a, i, l)$ has to be true as well for the component $\mathcal{L}_i$ and the action a that transition t_j is labeled with. Then however, constraint (4.20) is violated.

   The third condition is guaranteed by constraint (4.17). The equivalence forces the $ex(t_j, l)$ literals to true for every reached transition labeled l_l.

   The final condition that the reached state is the correct one is a similar case analysis as in the case of the non-determinized versions. However, in the presented model the case for scheduled components is different. For idle ones, the literal $sc(i, l)$ defined in constraint (4.22) is false and constraint (4.21) becomes:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow in(s_j, l)).$$

Thus, such components remain in the reached state. For scheduled ones, noticing that for them constraint (4.22) forces the $sc(i,l)$ literal to be true, the constraint is:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow \bigvee_{t_k \in pr(s_j) \setminus \Delta_i^\tau} ex(t_k, l) \vee \bigvee_{s_o \in ps_\tau(s_j)} in(s_o, l+1)).$$

Since no LTS contains non-trivial τ-loops (due to the preprocessing algorithm), it holds that when such an LTS is restricted to τ transitions, the remaining structure is a directed acyclic graph. This implies that this structure has a topological order.

Using this topological order, it is possible to show that for all the local states s_i, the value of $in(s_i, l+1)$ is determined. In addition, $in(s_i, l+1)$ is true iff s_i belongs to the τ-closure of a state whose incoming transition is executed in step l.

The first state s_i in the topological order is such that it has no incoming τ transitions. Thus, its truth value is defined based on the truth values of the $ex(t_k, l)$ literals for its incoming transitions. For such states, $in(s_i, l+1)$ is true iff some incoming transition of s_i is executed in step l.

For all the remaining states s_j, the truth value of the literal $in(s_j, l+1)$ is evaluated based on the execution of their incoming transitions and the truth value of the literals $in(s_i, l+1)$ for states s_i *earlier* in the topological order than s_j. For these states s_i, the value of $in(s_i, l+1)$ has been correctly determined.

Therefore, for the following states s_j, $in(s_j, l+1)$ is true iff some incoming transition of s_j with a visible action is executed in step l or s_j belongs to the τ-closure of such a state. □

Theorem 12 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the determinized synchronized product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is an $\mathcal{V}$-execution of some satisfying valuation $\mathcal{V}$ of $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. The idea is to define the literals of $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ based on the execution and then check that every constraint of the formula is satisfied. Any determinized interleaving execution of length k can be presented as follows:

$$\langle T_1^1, \ldots, T_n^1 \rangle \xrightarrow{l_1} \cdots \xrightarrow{l_k} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

The mapping $\mathcal{V}$ is as follows:

- $\mathcal{V}(in(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $s_j \in T_i^{\mathfrak{t}}$,
- $\mathcal{V}(ex(t_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $(s_i^{\mathfrak{t}}, l_{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) = t_j$ such that $s_i^{\mathfrak{t}} \in T_i^{\mathfrak{t}}$ and $s_i^{\mathfrak{t}+1} \in T_i^{\mathfrak{t}+1}$
- $\mathcal{V}(ex(a_i, \mathfrak{t})) =$ true iff $l_{\mathfrak{t}} = a_i$,
- $\mathcal{V}(ex(a_i, j, \mathfrak{t})) =$ true iff $l_{\mathfrak{t}} = a_i$, and

- $\mathcal{V}(sc(i, \mathfrak{t})) = \text{true}$ iff $l_{\mathfrak{t}} \in \Sigma_i$.

The proof is by induction over the conjuncts corresponding to different states of the execution (different values for $\mathfrak{t}$).

1. **Base Case.** Every determinized interleaving execution starts from the τ-closure of the initial states. Thus, constraints (4.15) and (4.16) are satisfied.

2. **Induction Hypothesis.** Assume no conjunct is violated in formula $IL_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ up to the state l.

3. **Induction Step.** Consider the step $\langle T_1^l, \ldots, T_n^l \rangle \xrightarrow{l_l} \langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$. The goal is to show that mapping $\mathcal{V}$ satisfied all the conjuncts of the formula $TR^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, l)$. By definition of a determinized interleaving execution, every transition labeled l_l and having its source state in the reached set of states is executed. Therefore constraint (4.17) is satisfied.

   Component $\mathcal{L}_i$ executes an action l_l iff it executes a transition labeled l_l. Constraint (4.18) is thus satisfied.

   Any determinized interleaving execution respects the synchronization requirement encoded in constraint (4.19).

   By definition of a determinized interleaving execution, l_l is unique. This satisfies constraint (4.20).

   The control flow remains in the same state for idle components and moves to the τ-closure of the target states of executed transitions. This satisfies constraint (4.21).

   The definition of the $sc(i, \mathfrak{t})$ literal in constraint (4.22) satisfied. □

4.5.2 Determinized Step Executions

Similarly as in the case of the composition operators not applying on-the-fly determinization, when step and process models are presented, the presentation is limited to studying the differences to the interleaving model.

In the determinized interleaving model, the cardinality constraint given in (4.20) limits the number executed actions. The upper limit is lifted by replacing the cardinality constraint by a disjunction as follows:

$$\bigvee_{a_i \in \Sigma \setminus \{\tau\}} ex(a_i, \mathfrak{t}) \tag{4.23}$$

However, only this modification has in the determinized model, where the number of executed transitions in a single component is not limited, the undesired effect that the resulting formula has models where several actions can be executed from a particular component. This can be ruled out with the following constraint:

$$\bigwedge_{1 \leq i \leq n} \text{card}_0^1\{ex(a_j, i, \mathfrak{t}) \mid a_j \in \Sigma_i \setminus \{\tau\}\} \tag{4.24}$$

Again, the executed actions are tuples, thus the concept of $\mathcal{V}_s$-sequence, defined in Definition 28 and repeated here, is applied in the proofs.

Definition 39 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and an integer k, let $\mathcal{V}_s$ be a valuation of any formula containing the following atomic propositions:*

- *for all $s_j \in S_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k+1$, the proposition $in(s_j, \mathfrak{t})$ and*
- *for all action–component pairs $(a, i) \in \bigcup_{1 \leq i \leq n} (\Sigma_i \times \{i\})$ and all $1 \leq \mathfrak{t} \leq k$, the proposition $ex(a, i, \mathfrak{t})$.*

*Then, the $\mathcal{V}_s$-*sequence *corresponding to $\mathcal{V}_s$ is the sequence:*

$$\langle T_1^1, \ldots, T_n^1 \rangle \overset{\langle L_1^1, \ldots, L_n^1 \rangle}{\rightarrow} \cdots \overset{\langle L_1^k, \ldots, L_n^k \rangle}{\rightarrow} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

such that each $T_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k+1$, is a set of states and a component state $s_l \in T_i^{\mathfrak{t}}$ iff $s_l \in S_i$ and $\mathcal{V}(in(s_l, \mathfrak{t})) =$ true. Each $L_i^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k$, is a set of actions and an action a_j is in $L_i^{\mathfrak{t}}$ iff $\mathcal{V}_s(ex(a_j, i, \mathfrak{t})) =$ true.

Definition 40 *Given a $\mathcal{V}_s$-sequence corresponding to a valuation $\mathcal{V}_s$ of the formula $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a valid* determinized step-sequence *iff the following conditions hold:*

1. *for all $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k+1$, $|T_i^{\mathfrak{t}}| \neq \emptyset$ and*
2. *for all $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k$, $|L_i^{\mathfrak{t}}| \leq 1$.*

Lemma 7 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}_s$ of the formula $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}_s$-sequence is a determinized step-sequence.*

Proof. It has to be proven that in all the global states of the $\mathcal{V}_s$-sequence the number of local states is non-zero. Secondly, in each step, the $ex(a, i, \mathfrak{t})$ literal is true for at most one action a from the alphabet $\Sigma_i \setminus \{\tau\}$.

The former condition, on the number of reached states is proven in precisely the same way as for the determinized interleaving model in the proof of Theorem 6. The latter condition is guaranteed by constraint (4.24). □

Given a determinized step-sequence, for all $1 \leq i \leq n$ and $1 \leq \mathfrak{t} \leq k$, if $L_i^{\mathfrak{t}} \neq \emptyset$, let the single element of $L_i^{\mathfrak{t}}$ be denoted $l_i^{\mathfrak{t}}$. Otherwise, let $l_i^{\mathfrak{t}}$ denote the empty action ϵ.

Definition 41 *A determinized step-sequence of $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the determinized step product $\mathcal{L}_{st}^d$ of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, iff the following conditions hold:*

1. *for all components $1 \leq i \leq n$, $T_i^1 = \tau(I_i)$ and*
2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle T_1^{\mathfrak{t}}, \ldots, T_n^{\mathfrak{t}} \rangle \overset{\langle l_1^{\mathfrak{t}}, \ldots, l_n^{\mathfrak{t}} \rangle}{\rightarrow} \langle T_1^{\mathfrak{t}+1}, \ldots, T_n^{\mathfrak{t}+1} \rangle \in \Delta_{st}^d$.*

Theorem 13 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}_s$ of the formula $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, then the corresponding $\mathcal{V}_s$-sequence is an execution of the determinized step product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$.*

Proof. In Lemma 7 it is shown that a $\mathcal{V}_s$-sequence corresponding to a valuation $\mathcal{V}_s$ of $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is a determinized step-sequence. Now it has to be established that every one of its steps is a transition of the determinized step product of components $\mathcal{L}_1, \ldots, \mathcal{L}_n$. For that knowledge about the executed transition, encoded using the $ex(t_j, \mathfrak{t})$ literals, is needed.

The proof is by induction over the states in the $\mathcal{V}_s$-sequence. The base case and the induction hypothesis are similar than in the case of the determinized interleaving model, namely:

1. **Base Case.** Due to constraints (4.15) and (4.16), in every valuation $\mathcal{V}_s$ of $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ the state $\langle T_1^1, \ldots, T_n^1 \rangle$ is the initial state of the determinized step product.

2. **Induction Hypothesis.** Assume that the $\mathcal{V}_s$-sequence is valid determinized step execution up to some state l.

3. **Induction Step.** Consider then the execution step $\langle T_1^l, \ldots, T_n^l \rangle \overset{\langle l_1^l, \ldots, l_n^l \rangle}{\rightarrow} \langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$. The proof of the induction step is a modified version of the proof for the non-determinized version. It is shown that:

   (a) the tuple $\langle l_1^l, \ldots, l_n^l \rangle$ can not be $\langle \epsilon, \ \ldots, \epsilon \rangle$.

   (b) if l_i^l is visible, then for every component j such that $l_j^l \in \Sigma_j, l_j^l = l_i^l$,

   (c) if $l_i^l \neq \epsilon$, then component $\mathcal{L}_i$ executes a transition labeled l_i^l,

   (d) if $l_i^l = \epsilon$, then component $\mathcal{L}_i$ executes no transition,

   (e) the executed transitions start from reached component states and every such transition labeled l_l is executed, and

   (f) $\langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle$ is the state of the determinized step product that is reached from the state $\langle T_1^l, \ldots, T_n^l \rangle$ by executing the action tuple $\langle l_1^l, \ldots, l_n^l \rangle$.

   The first condition that components may not idle is guaranteed by constraint (4.23).

   The second condition, the synchronization requirement, is guaranteed by constraint (4.19).

   The third condition requires that a transition labeled with l_i^l is executed. This is guaranteed by constraint (4.18).

   The fourth condition follows from the fact that if $l_i^l = \epsilon$, then for all $a \in \Sigma_i$, the literal $ex(a, i, l)$ is false. Then by constraint (4.18), no transition is executed.

   The two last items concerning the executed transitions and the progress of the control flow are established in precisely the same way as in the case of determinized interleaving executions. □

Theorem 14 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the determinized step product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is a $\mathcal{V}_s$-execution of some satisfying valuation $\mathcal{V}_s$ of $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. The proof presents a mapping that gives all the atomic propositions of $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ a truth value based on the elements of the given executions. Any determinized step execution of length k can be presented in the following form:

$$\langle T_1^1, \ldots, T_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\rightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\rightarrow} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

The mapping $\mathcal{V}_s$ is as follows:

- $\mathcal{V}_s(in(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $s_j \in T_i^{\mathfrak{t}}$,
- $\mathcal{V}_s(ex(t_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $(s_i^{\mathfrak{t}}, l_i^{\mathfrak{t}}, s_i^{\mathfrak{t}+1}) = t_j$ such that $s_i^{\mathfrak{t}} \in T_i^{\mathfrak{t}}$ and $s_i^{\mathfrak{t}+1} \in T_i^{\mathfrak{t}+1}$,
- $\mathcal{V}_s(ex(a_i, j, \mathfrak{t})) =$ true iff $l_j^{\mathfrak{t}} = a_i$,
- $\mathcal{V}_s(ex(a_i, \mathfrak{t})) =$ true iff for some $1 \leq j \leq n$, $l_j^{\mathfrak{t}} = a_i$, and
- $\mathcal{V}_s(sc(i, \mathfrak{t})) =$ true iff $l_i^{\mathfrak{t}} \neq \epsilon$.

The proof is by induction over conjuncts in $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ corresponding to different states of the execution (different values for $\mathfrak{t}$).

1. **Base Case.** Every determinized step execution starts from the unique initial state. This is precisely the state in which constraints (4.15) and (4.16) are satisfied.
2. **Induction Hypothesis.** Assume that all the conjuncts of the formula $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ are satisfied up to time step l.
3. **Induction Step.** Consider the step

   $$\langle T_1^l, \ldots, T_n^l \rangle \overset{\langle l_1^l, \ldots, l_n^l \rangle}{\rightarrow} \langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle.$$

   The goal is to show that mapping $\mathcal{V}_s$ satisfies all the conjuncts in the formula $TR_s^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, l)$. If action a is executed in component $\mathcal{L}_i$, then it holds that every reached transition labeled a is executed and constraint (4.17) is satisfied. Indeed, at least one such transition has to be executed and constraint (4.18) is satisfied.

   The synchronization requirement, i.e. that an action can be executed iff every component having that action in its alphabet participates, guarantees that constraint (4.19) is satisfied.

   Since the executions in the determinized step product are such that it is not possible to idle, constraint (4.23) is satisfied. Secondly, a single component can execute only a single action and (4.24) is satisfied.

   Finally, in determinized step executions, idle components do not execute any transitions and remain in the same control state, in accordance with the constraints (4.21) and (4.22).

The constraints are also satisfied for scheduled components where the reached state set is the τ-closure of the target states of executed transitions. The first disjunct in constraint (4.21) guarantees reachability for immediate target states and the second for their τ-closure. □

4.5.3 Determinized Process Executions

The needed modification to move from the determinized step executions to determinized process executions is similar than in the case of the non-determinized version. In fact, it is the same except for the fact that on-the-fly determinization handles τ transitions differently. The single additional constraint is as follows:

$$\bigwedge_{a \in \Sigma \setminus \{\tau\}} ex(a, \mathfrak{t}+1) \rightarrow \bigvee_{j \in C_a} sc(j, \mathfrak{t}) \tag{4.25}$$

Due to on-the-fly determinization, no constraint corresponding to (4.14) is needed. Given n components and a bound k, the formula to encode determinized process executions of length k, denoted $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, is of the following form:

$$PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS_d(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR_s^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i) \wedge \bigwedge_{1 \leq i < k} PRC_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$

where

- $IS_d(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is the constraint on the initial state,
- $TR_s^d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ encodes the transition relation from state i to $i+1$, and
- $PRC_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ is the constraint to encode the process condition, i.e., constraint (4.25) with the variable $\mathfrak{t}$ instantiated to i.

Definition 42 *Let $\mathcal{L}_{st}^d$ be the determinized step product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$ A $\mathcal{V}_s$-sequence of $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the determinized process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, iff the following conditions hold:*

1. *for all components $1 \leq i \leq n$, $T_i^1 = \tau(I_i)$,*
2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle T_1^{\mathfrak{t}}, \ldots, T_n^{\mathfrak{t}} \rangle \overset{\langle l_1^{\mathfrak{t}}, \ldots, l_n^{\mathfrak{t}} \rangle}{\rightarrow} \langle T_1^{\mathfrak{t}+1}, \ldots, T_n^{\mathfrak{t}+1} \rangle \in \Delta_{st}^d$, and*
3. *this step execution fulfills the process condition.*

The theorems of soundness and completeness are as follows:

Theorem 15 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any satisfying valuation $\mathcal{V}_s$ of the formula $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ then the corresponding $\mathcal{V}_s$-execution is an execution of the determinized process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$.*

Proof. Firstly, any $\mathcal{V}_s$-execution constructed from a model of the formula $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is a determinized step execution. Then, the proof is the same as that of Theorem 9. □

Theorem 16 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of length k of the determinized process product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, the corresponding determinized step execution is a $\mathcal{V}_s$-execution of some satisfying valuation $\mathcal{V}_s$ of $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. Firstly, any step execution obtained from an execution of the determinized process product satisfies all the constraints in $ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. These constraints are also present in $PR_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.

Then, the proof is the same as that of Theorem 10. □

4.6 REACHABILITY PROPERTIES

This dissertation focuses on bounded verification of reachability properties of concurrent systems where the components are given as LTSs. Bounded verification of reachability properties of such systems equals answering the question of whether in the product of the components there is an execution of some bounded length to a state where the property holds. The product, which is a single LTS, can be different depending on the execution model. This section demonstrates how the Boolean representations of reachability properties are combined with the BMC formulas presented in the previous sections. This is done in a different manner depending on whether on-the-fly determinization is applied or not.

In this work, it is assumed that a reachability property to be verified is given as a propositional formula over the local states of the components forming the system. Assume that the system consists of n components, each component $\mathcal{L}_i$ contains l_i local states and the local states of component $\mathcal{L}_i$ are represented in the reachability property with variables $s_{i1}, \ldots, s_{il_i}$, exactly one of which is true at a time. Then a reachability property is the formula:

$$\phi(s_{11}, \ldots, s_{1l_1}, \ldots, s_{n1}, \ldots, s_{nl_n}).$$

The formula evaluates to true iff the true variables s_{ij} are such that they form a global state where the property holds. The verification of a reachability property is performed as follows. All of the presented translation constraints contain a literal $in(s, \mathrm{t})$ for every local state of every component. The literal is true in a model of the formula iff the local state s is reached in state t. The question one is interested in answering is whether there is an execution whose last global state satisfies the reachability property. The models of the BMC formulas are limited to such executions by forming a conjunction where the first conjunct is the BMC formula (the symbolic unrolling of the transition relation k steps) and the second conjunct is the reachability predicate. However, the state variables s_{ij} in the reachability property have to replaced with the literal $in(s_{ij}, k+1)$. For instance, assuming that a step product is formed from the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$, then the complete

formula is:

$$ST(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) \wedge$$
$$\phi(in(s_{11}, k+1), \ldots, in(s_{1l_1}, k+1), \ldots, in(s_{n1}, k+1), \ldots, in(s_{nl_n}, k+1))$$

However, the presentation above can be applied only if in the construction of the product on-the-fly determinization is *not* applied. If that is not the case, the situation is harder. For this, there are two reasons: (i) When on-the-fly determinization is applied, the global states of the product are such that each component is in a set of local states. Thus, a global state in the product represents a set of global states of the system under study. (ii) On-the-fly determinization requires a preprocessing step that eliminates non-trivial τ-loops from the components. Therefore, a local state of the preprocessed system can represent several local states of the original system.

If on-the-fly determinization is applied, the $in(s, k+1)$ literals in the models of the corresponding BMC formula can be true for several local states from each component of the preprocessed system. However, the BMC formulas can be augmented with additional constraints. This can be done in such a way that the resulting formula has a model iff a particular global state of the *original* system is reachable with an execution of the desired execution semantics (interleaving / step / process) of length k.

The additional constraints to the formula contain a new literal $fs(s_{ij}, k+1)$. This literal is used to encode the fact that the original (not preprocessed) component $\mathcal{L}_i$ is in the (original) state s_{ij}. The literal is constrained in two ways. Firstly, when s_{ij} is the reached state (of the original system), then in the model of the BMC formula the state $repr(s_{ij})$ representing its τ-component has to be reached. Secondly, precisely one state can be reached from each component. Formally:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq l_i} fs(s_{ij}, k+1) \rightarrow in(repr(s_{ij}), k+1) \tag{4.26}$$

$$\bigwedge_{1 \leq i \leq n} \text{card}_1^1\{fs(s_{ij}, k+1) \mid 1 \leq j \leq l_i\} \tag{4.27}$$

where

- $repr(s_{ij})$ is the state representing the τ-component corresponding to state s_{ij} of the original system.

Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and a bound k, let the conjunction of the constraints (4.26) and (4.27) be denoted by $\psi(\mathcal{L}_1, \ldots, \mathcal{L}_n, k+1)$. The models of, for instance, the determinized step product are then modified to those reaching a global state where the reachability predicate holds as follows:

$$ST_d(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) \wedge$$
$$\psi(\mathcal{L}_1, \ldots, \mathcal{L}_n, k+1) \wedge$$
$$\phi(fs(s_{11}, k+1), \ldots, fs(s_{1l_1}, k+1), \ldots, fs(s_{n1}, k+1), \ldots, fs(s_{nl_n}, k+1))$$

As can be seen, the variables s_{ij} of the original reachability predicate are now replaced with the $fs(s_{ij}, k+1)$ literals encoding the reached global state.

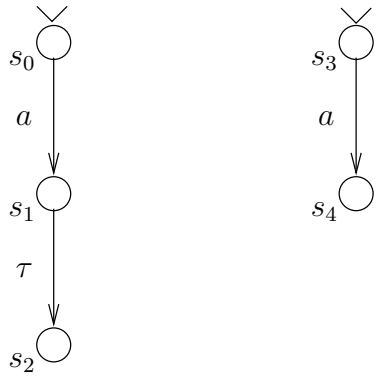

Figure 4.2: The Literal $fs(s_i, \mathfrak{t})$

Indeed, constraints (4.26) and (4.27) are needed if on-the-fly determinization is applied and the reachability predicate is global. This is the case even if the original components do not contain non-trivial τ-loops. If one is interested in local reachability, constraints (4.26) and (4.27) are not needed.

The situation is illustrated in Figure 4.2. Consider a reachability predicate that holds only in the state $\langle s_2, s_4 \rangle$ of the product of these components. With any of the presented execution models that apply on-the-fly determinization, the two LTSs reach this state after executing a. The component on the left-hand side is thereafter in the state set $\{s_1, s_2\}$ and the component on the right-hand side in the state set $\{s_4\}$. Thus, if the $in(s_i, \mathfrak{t})$ literal is used in a reachability predicate, this predicate evaluates to false, since the literal is true for both the state s_1 and s_2. The purpose of the $fs(s_i, \mathfrak{t})$ literal is to allow the model where $fs(s_2, \mathfrak{t})$ is true and $fs(s_1, \mathfrak{t})$ is false. This allows the correct detection of the witness to the property.

A deadlock state predicate which holds for exactly those global states from which no execution can be extended by any transition is a particularly interesting reachability property. Indeed, the state $\langle s_2, s_4 \rangle$ in for instance the synchronized product of the components in Figure 4.2 is a deadlock. It is possible to encode a deadlock by analyzing the system statically and listing all the global states from which no transition is possible. Then, the reachability property would be a disjunction where each disjunct is a conjunction of the form $s_{1i_1} \wedge \cdots \wedge s_{ni_n}$, exactly one state from each component. However, the number of such states can be as large as the state space of the product and most of these states are probably not reachable.

The same states can be encoded in a more compact manner by introducing two additional literals that are evaluated based on the s_{ij} literals encoding the local states. The literal $en(a, i, k+1)$ evaluates to true iff the reached local state in component $\mathcal{L}_i$ in the global state $k+1$ has an outgoing visible transition labeled a. The other literal, $en(a, k+1)$, is true iff all the components having a in their alphabet are in local states having an outgoing visible transition labeled a. The deadlock state predicate is then completed by requiring that the deadlock predicate holds in a global state s iff no action is enabled in s. However, the above treatment does not take into account states having outgoing τ transitions. No such local state can be an element of a global deadlocking state (because a τ transition is possible).

Formally, the encoding of the deadlock state predicate is the conjunction of four items. Firstly, a constraint ruling out local states with outgoing τ transitions. Secondly, the definitions of the two new literals given above. Finally, a constraint ruling out state vectors where some visible action is enabled. In presenting the constraint, the following definition is applied:

Definition 43 *Let $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$ be an LTS and $s \in S_i$. Then the set $ac(s) = \{a \in \Sigma_i \mid (s, a, s') \in \Delta_i\}$, i.e, the set of transition labels of the outgoing transitions of the state s.*

The formula to encode the deadlock state predicate is then as follows:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_{ij} \in S_i, \tau \in ac(s_{ij})} \neg s_{ij} \Big) \wedge$$
$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{a \in \Sigma \setminus \{\tau\}} (en(a, i, k+1) \leftrightarrow \bigvee_{s_{ij} \in S_i, a \in ac(s_{ij})} s_{ij}) \Big) \wedge$$
$$\bigwedge_{a \in \Sigma \setminus \{\tau\}} (en(a, k+1) \leftrightarrow \bigwedge_{i \in C_a} en(a, i, k+1)) \wedge$$
$$\bigwedge_{a \in \Sigma \setminus \{\tau\}} \neg en(a, k+1).$$

Depending on whether deadlock checking is performed using execution models applying on-the-fly determinization or not, the s_{ij} literals in the predicate above are replaced with the $fs(s_{ij}, k+1)$ or $in(s_{ij}, k+1)$ literals, respectively.

4.7 COMPLETENESS

Section 3.1 presents briefly a technique to guarantee completeness by computing a number called the *recurrence diameter*, the longest loop-free path between two states, using propositional satisfiability. The idea in the encoding is to create a formula encoding all the executions of some length k together with the constraint that all the states are different. Then, if this formula is unsatisfiable, no such execution exists and the bound k suffices for a complete verification procedure.

The verifier can then apply the idea above by starting from a small bound and if the resulting formula is satisfiable (there is an execution where all the states are different), increase the bound step by step until either the formula turns out to be unsatisfiable or the computational resources are exhausted.

In general, the reached states in the presented encodings is encoded using the literal $in(s, \mathfrak{t})$. The literal is true iff the local state s is reached in state $\mathfrak{t}$. The encoding of the condition that all the state have to be different is then as follows. Firstly, let $sa(\mathfrak{t}_1, \mathfrak{t}_2)$ be a new literal that is true iff the global states $\mathfrak{t}_1$ and $\mathfrak{t}_2$ are the same. Its definition is then as follows:

$$sa(\mathfrak{t}_1, \mathfrak{t}_2) \leftrightarrow \Big(\bigwedge_{1 \leq i \leq n} \bigwedge_{s_j \in S_i} in(s_j, \mathfrak{t}_1) \leftrightarrow in(s_j, \mathfrak{t}_2) \Big). \tag{4.28}$$

Then, the complete constraint to implement the requirement that all the states in the execution of length k are different is:

$$\bigwedge_{1\leq \mathtt{t}_1\leq k} \Big(\bigwedge_{\mathtt{t}_1<\mathtt{t}_2\leq k+1} \neg sa(\mathtt{t}_1,\mathtt{t}_2)\Big). \tag{4.29}$$

It should be noted that since constraint (4.29) has to be instantiated for all the state *pairs* from 1 to $k+1$, the resulting formula is quadratic in the bound k.

However, the situation is more complex, when the process or determinized process products are studied. With them, namely, the state vector consists of component states and truth values carrying information of the executed actions in the previous step. Thus, implementing the check that all states are different using the constraint (4.28) (omitting the truth values from the state vector) above might result in too small a bound. It turns out, though, that this is not the case. This is shown by demonstrating that if there is a process execution σ_{P} having two states that have exactly the same component states (but possibly different Boolean values in the state vectors), then σ_{P} can always be converted to a shorter execution σ'_{P} whose final global state contains exactly the same component states as the final state of σ_{P}.

Lemma 8 *Given the components $\mathcal{L}_1,\ldots,\mathcal{L}_n$ and their process product $\mathcal{L}_{pr}$, if*

$$\langle s_1^1, b_1^1, \ldots, s_n^1, b_n^1\rangle \stackrel{\langle l_1^1,\ldots,l_n^1\rangle}{\longrightarrow} \cdots \stackrel{\langle l_1^k,\ldots,l_n^k\rangle}{\longrightarrow} \langle s_1^{k+1}, b_1^{k+1} \ldots, s_n^{k+1}, b_n^{k+1}\rangle$$

is an execution σ_{P} of $\mathcal{L}_{pr}$ such that there is are integers $1\leq i\leq k$, $i<j\leq k+1$ where for all $1\leq l\leq n$, $s_l^i=s_l^j$, then there is an execution σ'_{P} of $\mathcal{L}_{pr}$ for which it holds that:

- *σ'_{P} reaches the state $\langle s_1^{k+1}, b_1, \ldots, s_n^{k+1}, b_n\rangle$ for some Boolean values b_1, $\ldots$, b_n and*
- *$|\sigma_{\mathrm{P}}|<k$.*

Proof. The proof is as follows. Firstly, omitting the truth values from the state vectors of σ_{P} results in an executions of $\mathcal{L}_{st}$ where, by the assumption in the theorem, the states i and j are identical. Then, however, the execution

$$\langle s_1^1,\ldots,s_n^1\rangle \stackrel{\langle l_1^1,\ldots,l_n^1\rangle}{\longrightarrow} \cdots \stackrel{\langle l_1^{i-1},\ldots,l_n^{i-1}\rangle}{\longrightarrow} \langle s_1^j,\ldots,s_n^j\rangle \stackrel{\langle l_1^j,\ldots,l_n^j\rangle}{\longrightarrow} \cdots \stackrel{\langle l_1^k,\ldots,l_n^k\rangle}{\longrightarrow} \langle s_1^{k+1},\ldots,s_n^{k+1}\rangle$$

is an execution of $\mathcal{L}_{st}$. Now, for this execution σ_{S}, it holds that its length is $|\sigma_{\mathrm{S}}|<k$ and it reaches the state $k+1$ of σ_{P} when the truth values are omitted from the state vector. However, it can be the case that σ_{S} does not fulfill the process condition. If that is the case, by repeated application of Lemma 2 in Chapter 2, it can be converted to an execution of $\mathcal{L}_{st}$ that fulfills the process condition and reaches the same state. Secondly, by Lemma 1 in Chapter 2, for any such execution, there is an execution σ'_{P} of $\mathcal{L}_{pr}$. □

Theorem 17 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and their process product $\mathcal{L}_{pr}$, any execution σ_{P} of $\mathcal{L}_{pr}$ can be converted to another execution σ'_{P} of $\mathcal{L}_{pr}$ such that:*

- *σ'_{P} reaches the same state than σ_{P},*
- *$|\sigma'_{\mathrm{P}}| \leq |\sigma_{\mathrm{P}}|$, and*
- *σ'_{P} contains no two states i and j such that for all $1 \leq l \leq n$, $s_l^i = s_l^j$.*

Proof. The proof is by repeated application of the construct given in the proof of Lemma 8. Since σ_{P} is of bounded length, and any application of the construct renders it shorter, eventually it can not contain states where only the truth values of the state vector are different. □

The theorem above states that any process execution that has two global states having the same component states but differing in the values b_i in the state vectors, can be converted to a process execution reaching the same state and in which all the state vectors differ also in the component states. However, from this it follows that the definition of the literal $sa(\mathrm{t}_1, \mathrm{t}_2)$ given in constraint (4.28) can be soundly used also with the process product.

5 ENCODING ALGORITHM WITH LOCAL TRANSITION MERGING

This chapter analyzes the problems of encoding the executions of the path product, i.e., the execution model where local transition merging is used. The path product has a transition between two states iff the step product of the same components has one or more executions between these states. The label on the transition denotes the regular language that is obtained by considering all the executions of the step product between the transition's source and target states. If such executions of arbitrary length are allowed (as in the most general model), the violation of any reachability property can be demonstrated with a counterexample of length one. It is assumed that a BMC encoding of this most general model can not be polynomial. Namely, if that was the case, then a **PSPACE**-complete problem, reachability of a state in the synchronized product of LTSs [58], could be solved in non-deterministic polynomial time (**NP**). This is generally considered unlikely (see [73] for further discussion on these complexity classes).

This chapter also presents a composition operator that yields an LTS following the idea of local transition merging but with additional constraints. The encoding of the executions of this product is possible but the resulting formula is cubic in the size of the alphabet. However, analyzed test cases support the claim that this encoding can in some cases be effective, especially if performed iteratively. Finally, the chapter is concluded by analyzing the problems encountered when trying to lift the presented limitations to the path product.

The following definition characterizes the additional constraints applied on local transition merging:

Definition 44 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $\mathcal{L}_{st}$ their step product. An execution*

$$\langle s_1^1, \ldots, s_n^1 \rangle \overset{\langle l_1^1, \ldots, l_n^1 \rangle}{\rightarrow} \cdots \overset{\langle l_1^k, \ldots, l_n^k \rangle}{\rightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

of $\mathcal{L}_{st}$ is simple *iff*

1. *for all $1 \leq i \leq n$, $1 \leq \mathfrak{t} \leq k$, if $l_i^{\mathfrak{t}} = a$ and $a \in \Sigma_i \setminus \{\tau\}$, then for all $\mathfrak{t}' \neq \mathfrak{t}$, $l_i^{\mathfrak{t}'} \neq a$ and*
2. *for all $1 \leq i \leq n$, $1 \leq \mathfrak{t} < k$, if $l_i^{\mathfrak{t}} \neq \epsilon$, then for all $\mathfrak{t}' \leq \mathfrak{t}$, $s_i^{\mathfrak{t}'} \neq s_i^{\mathfrak{t}+1}$.*

Intuitively, simple executions are step executions such that each component executes a path of transitions where each visible action from the component's alphabet appears at most once. Furthermore, if along the path some local state is visited twice, then the component can not execute further transitions.

Definition 45 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$, $1 \leq i \leq n$, be LTSs. Let $\mathcal{L}_{st} = \langle S_{st}, I_{st}, \Sigma_{st}, \Delta_{st} \rangle$ be the step product $\mathcal{L}_1 \parallel_{st} \cdots \parallel_{st} \mathcal{L}_n$. The* limited path product *of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, denoted $\mathcal{L}_1 \parallel_{pt}^{l} \mathcal{L}_2 \parallel_{pt}^{l} \cdots \parallel_{pt}^{l} \mathcal{L}_n$, is the LTS $\mathcal{L}_{pt}^{l} = \langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = S_1 \times S_2 \times \cdots \times S_n$,

- $I = I_1 \times I_2 \times \cdots \times I_n$,
- $\Sigma = \{r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle} | \ \langle s_1,\ldots,s_n\rangle \in S, \langle s'_1,\ldots,s'_n\rangle \in S\}$, *and*
- $\Delta = \{\langle s_1,\ldots,s_n\rangle, r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle}, \langle s'_1,\ldots,s'_n\rangle |$ *there is a simple execution from* $\langle s_1,\ldots,s_n\rangle$ *to* $\langle s'_1,\ldots,s'_n\rangle$ *in* $\mathcal{L}_{st}$.$\}$

In each step at least one component has to be scheduled and the maximum length for a simple execution is therefore $\sum_{1\leq i\leq n} |S_i|$. Secondly, the number of outgoing transitions (outdegree) from each state is finite. Thus, the number of simple executions between any two states in $\mathcal{L}_{st}$ is necessarily finite and therefore the language describing these executions regular. That is why the labels of the transitions are of the form $r_{\langle s_1,\ldots,s_n\rangle,\langle s'_1,\ldots,s'_n\rangle}$.

Similarly as in the case of the execution models applying on-the-fly determinization, in order to the encoding presented in Section 5.1 to work, the component LTSs have to be preprocessed. However, the reason for the preprocessing and the preprocessing algorithm are different. The algorithm applies the following concept.

Definition 46 *Let $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i\rangle$ be an LTS. A state $s \in S_i$ is a* divergence *state iff it holds that $s \xrightarrow{\tau^\omega}$. The set of all divergence states of LTS $\mathcal{L}_i$ is denoted $div(\mathcal{L}_i)$.*

The preprocessing algorithm is as follows: Let $\mathcal{L}_1,\ldots,\mathcal{L}_n$ be LTSs. For all $1 \leq i \leq n$:

> Replace $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i\rangle$ with the LTS $\mathcal{L}'_i = \langle S'_i, I'_i, \Sigma'_i, \Delta'_i\rangle$ such that:
>
> - $S'_i = S_i$,
> - $I'_i = I_i$,
> - $\Sigma'_i = \Sigma_i$, and
> - $\Delta'_i = \{(s,\tau,s) \mid s \text{ is a divergence state in } \mathcal{L}_i\} \cup \{(s,a,s') \in S_i \times \Sigma_i \setminus \{\tau\} \times S_i \mid \text{there is a path } s \xrightarrow{\tau^* a \tau^*} s' \text{ in } \mathcal{L}_i.\}$

The intuitive idea of the algorithm above is to eliminate from all the LTSs all the τ transitions except for self-loops. This is needed in order to be able to encode compactly the constraint to be presented in Section 5.1.1. The preprocessing algorithm is illustrated in Figure 5.1 that gives an LTS (left hand side) and the LTS obtained after preprocessing it (right hand side).

In the present work, the BMC encodings are used to detect deadlocks. In the following, it is established that preprocessing the components with the algorithm above does not introduce any new nor remove any deadlocks from the components' limited path product.

Lemma 9 *Let $\mathcal{L}_i$ be an LTS and $\mathcal{L}'_i$ its preprocessed version. A $\mathcal{L}_i$ contains the execution $s_1 \xrightarrow{\ell(\sigma)} s_2$ iff $\mathcal{L}'_i$ contains the execution $s_1 \xrightarrow{\ell(\sigma')} s_2$ such that $vis(\sigma) = vis(\sigma')$.*

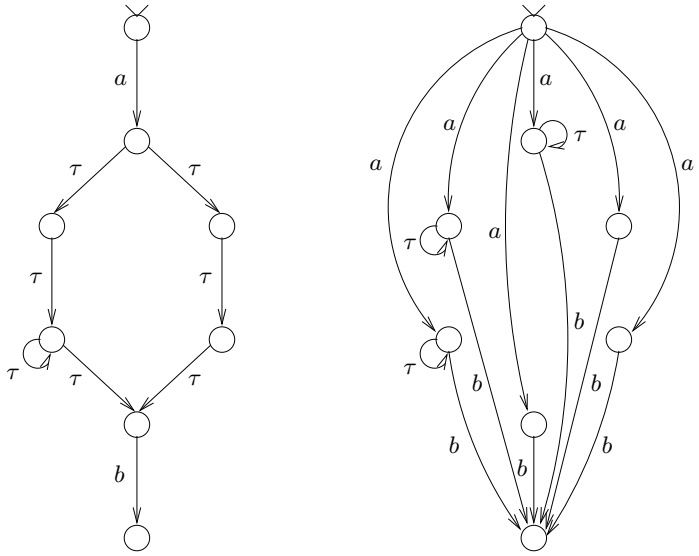

Figure 5.1: Preprocessing Algorithm

Proof. The labels of any execution σ of $\mathcal{L}_i$ can be presented in the form $\tau^* a_1 \tau^* \dots \tau^* a_k \tau^*$ (where for all $1 \leq j \leq k$, $a_j \neq \tau$). Let σ' be the execution of $\mathcal{L}'_i$ to be constructed. By definition of the transition relation in the preprocessing algorithm, if σ contains a segment $s_i \overset{\tau^* a_i \tau^*}{\rightarrow} s_{i+j}$, then $\mathcal{L}'_i$ contains a transition $s_i \overset{a_i}{\rightarrow} s_{i+j}$. Thus, σ' can be constructed from s_1 along these transitions up to the state s_2.

The labels of any execution σ' of $\mathcal{L}'_i$ can also be presented in the form $\tau^* a_1 \tau^* \dots \tau^* a_k \tau^*$ (where for all $1 \leq j \leq k$, $a_j \neq \tau$). However, since the τ^* sequences are executions of self-loops, then there is an execution with the same initial and final states with labels $a_1 \dots a_k$. Let σ be the execution of $\mathcal{L}_i$ to be constructed. If σ' contains a transition $s_i \overset{a_i}{\rightarrow} s_{i+1}$, then σ contains a segment $s_i \overset{\tau^* a_i \tau^*}{\rightarrow} s_{i+1}$. Thus, σ can be constructed up to state s_2. □

Lemma 10 *Let $\mathcal{L}_i$ be an LTS and $\mathcal{L}'_i$ its preprocessed version. It holds that $div(\mathcal{L}_i) = div(\mathcal{L}'_i)$.*

Proof. Let $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$ and $\mathcal{L}'_i = \langle S'_i, I'_i, \Sigma'_i, \Delta'_i \rangle$. The transition relation Δ'_i contains a transition (s, τ, s) iff $s \in S_i$ is a divergence state in $\mathcal{L}_i$. Because of the transition (s, τ, s), s is a divergence also in $\mathcal{L}'_i$.

Because these transitions are the only τ transitions in Δ'_i, no other states in $\mathcal{L}'_i$ are divergences and the result follows. □

The following theorem establishes that if the components are preprocessed before the composition yielding their limited path product, deadlock executions are preserved. However, it can be the case that in the preprocessed version a deadlock is detected a few steps earlier than in the original. The precise relationship is as follows:

Theorem 18 *Let $\mathcal{L}_1, \dots, \mathcal{L}_n$ be LTSs and for all $1 \leq i \leq n$, let $\mathcal{L}'_i$ be the preprocessed version of $\mathcal{L}_i$. The LTS $\mathcal{L}_{pt} = \mathcal{L}_1 \parallel^l_{pt} \cdots \parallel^l_{pt} \mathcal{L}_n$ has a deadlock*

execution reaching state $\langle s_1, \ldots, s_n \rangle$ *iff* $\mathcal{L}'_{pt} = \mathcal{L}'_1 \parallel^l_{pt} \cdots \parallel^l_{pt} \mathcal{L}'_n$ *has a deadlock execution reaching* $\langle s'_1, \ldots, s'_n \rangle$ *and it holds that for all* $1 \leq i \leq n$, $\mathcal{L}_i$ *has a path* $s'_i \xrightarrow{\tau^*} s_i$.

Proof. Firstly, if any state $\langle s_1, \ldots, s_n \rangle$ in a limited path product of n components does not have outgoing transitions, then no state s_i, $1 \leq i \leq n$ can have outgoing τ transitions. For every visible action $a \in \Sigma \setminus \{\tau\}$, there has to be some component $\mathcal{L}_i$ such that $a \in \Sigma_i$ but there is no outgoing transition from s_i labeled a.

If $\mathcal{L}_{pt}$ has a deadlock execution reaching state $s = \langle s_1, \ldots, s_n \rangle$, then $\mathcal{L}'_{pt}$ has an execution reaching $s' = \langle s_1, \ldots, s_n \rangle$. This follows from Lemma 9. It is shown that s' in $\mathcal{L}'_{pt}$ is a deadlock state. For all $1 \leq i \leq n$, if $s_i \in s$ does not have outgoing τ transitions, then $s_i \in s'$ can not have a self-loop labeled τ. Secondly, if $s_i \in s$ does not have an outgoing visible transition labeled a nor outgoing τ transitions, then $s_i \in s'$ can not have an outgoing transition labeled a. Thus s' in $\mathcal{L}'_{pt}$ does not have outgoing transitions.

If $\mathcal{L}'_{pt}$ has a deadlock execution reaching state $s' = \langle s'_1, \ldots, s'_n \rangle$, then $\mathcal{L}_{pt}$ has an execution reaching $s = \langle s'_1, \ldots, s'_n \rangle$. Since no $s_i \in s'$ has outgoing τ transitions, by Lemma 10, no state $s'_i \in s$ can be a divergence. Therefore $\mathcal{L}_i$ has a (possibly empty) path $s'_i \xrightarrow{\tau^*} s_i$ where all the outgoing transitions of s_i are labeled with visible actions. Let then A_i be the set of labels of the outgoing transitions $t_j \in \Delta_i$ from state s_i in $\mathcal{L}_i$ and A'_i the set of labels of the outgoing transitions $t'_j \in \Delta'_i$ from s'_i in $\mathcal{L}'_i$. It holds that $A_i \subseteq A'_i$. Thus $\langle s_1, \ldots, s_n \rangle$ in $\mathcal{L}_{pt}$ is a deadlock state. □

If the LTSs $\mathcal{L}_1, \ldots, \mathcal{L}_n$ are preprocessed using the algorithm above, the local transition sequences of the simple executions of their step product are such that if some component executes a τ transition, that transition has to be the last one on the path. This follows directly from the fact that the τ transitions are limited to self-loops and thus, if such a transition is executed, a local state is visited twice and the local path has to terminate.

5.1 ENCODING ALGORITHM FOR LIMITED PATH PRODUCT

The encoding of the bounded executions of the limited path product follows the ideas and uses the literals used in the encoding of the execution models already presented. The encoding starts with constraints on the initial states. These are precisely the same as in the non-determinized models, i.e., the following constraints have to be added:

$$\bigwedge_{1 \leq i \leq n} \text{card}^1_1\{in(s_j, 1) \mid s_j \in I_i\} \text{ and} \tag{5.1}$$

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i \setminus I_i} \neg in(s_j, 1) \Big). \tag{5.2}$$

The following constraints are needed to limit the possible choices of executed transitions to paths. Firstly, if a transition is executed, then either its source state has to be reached (it is the first transition on the path) or some

incoming transition to its source state is executed. The incoming transition can not be a self-loop. The constraint applies the following concept.

Definition 47 *Let $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$. Then for all $s_j \in S_i$, $prn(s_j) = \{(s_k, a, s_j) \in \Delta_i \mid s_k \neq s_j\}$, i.e., the set of incoming transitions of state s_j that are not self-loops.*

The constraint is then as follows:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{t_j \in \Delta_i} (ex(t_j, \mathfrak{t}) \rightarrow in(sr(t_j), \mathfrak{t}) \vee \bigvee_{t_k \in prn(sr(t_j))} ex(t_k, \mathfrak{t})) \Big) \tag{5.3}$$

where

- $sr(t_j)$ is the source state of transition t_j and
- $prn(sr(t_j))$ is the set of incoming transitions of the source state of t_j that are not self-loops.

The constraint above is still not sufficient to limit transitions to paths. Namely, several outgoing transitions from a particular state can be executed. This is ruled out as follows:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{s_j \in S_i} \text{card}_0^1\{pt(s_j)\} \tag{5.4}$$

where

- $pt(s_j)$ is the set of outgoing transitions of state s_j.

Obviously, the path executions have to implement the synchronization requirement central to the definition of the semantics of synchronizing LTSs:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \Big(\bigwedge_{j \in C_{a_i}} (ex(a_i, j, \mathfrak{t}) \leftrightarrow \bigvee_{t_k \in \Delta_j^{a_i}} ex(t_k, \mathfrak{t})) \Big). \tag{5.5}$$

The $ex(a_i, j, \mathfrak{t})$ literals are required to have the same truth value for a particular action a_i:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \Big(\bigwedge_{j \in C_{a_i}} ex(a_i, \mathfrak{t}) \leftrightarrow ex(a_i, j, \mathfrak{t}) \Big). \tag{5.6}$$

Finally, as in the other execution models, the control flow has to be encoded. However, for that purpose, the last state on the path has be inferred based on the used literals. This is a bit cumbersome, therefore an additional literal $lst(s_j, \mathfrak{t})$ is introduced. In the final formula, the literal evaluates to true iff s_j is the last state on the path. Thereafter, the formula for control flow is:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_j \in S_i} (in(s_j, \mathfrak{t}+1) \leftrightarrow (lst(s_j, \mathfrak{t}) \vee (in(s_j, \mathfrak{t}) \wedge \neg\, sc(i, \mathfrak{t}))))\Big). \tag{5.7}$$

If an execution is simple, then along all the local paths of the components, if some local state is encountered twice, the latter instance has to be the last state on the path. From this fact it follows that the local paths are of the following form:

- straight segments (all local states different),
- loops to the initial state, or
- loops ending to some intermediate state (lasso-shaped).

The definition of the literal $lst(s_j, \mathfrak{t})$ is a disjunction of these three cases. Firstly, a state terminates a straight segment iff one of its non-looping incoming transitions and no outgoing transitions are executed. Secondly, a state ends a loop to the initial state iff that state is reached in the previous global state and some incoming transition to it is executed. Finally, a state terminates a lasso-shaped path iff more than one of its incoming transitions are executed. Formally:

$$\begin{aligned}\bigwedge_{1\leq i\leq n}\,(\bigwedge_{s_j\in S_i}(lst(s_j,\mathfrak{t}) \;\leftrightarrow\; ((&\bigvee_{t_k\in prn(s_j)} ex(t_k,\mathfrak{t})\wedge\bigwedge_{t_l\in pt(s_j)}\neg ex(t_l,\mathfrak{t}))\vee\\ &(\bigvee_{t_m\in pr(s_j)}(ex(t_m,\mathfrak{t})\wedge in(s_j,\mathfrak{t}-1)))\vee\\ &\neg\mathrm{card}_0^1\{pr(s_j)\}))).\end{aligned}\tag{5.8}$$

Once again, the definition of the control flow in constraint (5.7) makes use of the $sc(i, \mathfrak{t})$ literal that is defined in the same way as previously:

$$\bigwedge_{1\leq i\leq n}(sc(i,\mathfrak{t})\leftrightarrow\bigvee_{a_j\in\Sigma_i} ex(a_j,i,\mathfrak{t})).\tag{5.9}$$

Figure 5.2 gives an example LTS. In the following, some examples of the instantiation of constraint (5.8) are given. State s_1 is the initial state of the system. In subsequent steps it is the last state of the executed path of transitions iff the path is a straight segment or a loop to the initial state. This is the case since it only has one incoming transition. Consider, for instance, executions of length two. Constraint (5.8) for state s_1 and step 2 becomes:

$$lst(s_1,2)\leftrightarrow(ex(l_4,2)\wedge\ \neg ex(l_1,2))\vee(ex(l_4,2)\wedge in(s_1,1))$$

State s_2, on the other hand, can close a straight segment, a loop to the initial state, or a lasso-shaped path. Notice that if transition l_2 is executed, then the executed path is either a loop or lasso-shaped. The encoding for step 2 is as follows:

$$\begin{aligned}lst(s_2,2) \;\leftrightarrow\; &(ex(l_1,2)\wedge\ (\neg ex(l_2,2)\wedge\neg ex(l_3,2)))\vee\\ &((ex(l_1,2)\vee ex(l_2,2))\wedge in(s_2,1))\vee\\ &\neg\mathrm{card}_0^1\{ex(l_1,2),ex(l_2,2)\}\end{aligned}$$

In many cases, constraint (5.8) can be simplified. For instance, if a state does not have any outgoing transitions, it can not close a loop. In addition, it is always the case that none of its (non-existent) outgoing transitions are executed. Consider the state s_4 in Figure 5.2. It is the last state on the executed path of transitions iff transition l_5 is executed. Formally for step 2:

$$lst(s_4,2)\leftrightarrow ex(l_5,2)$$

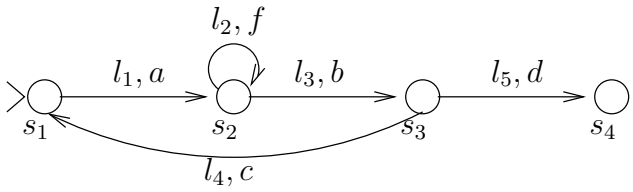

Figure 5.2: Running Example

It may seem that constraint (5.8) is not sound. After all, what prevents for instance the executed path of transitions from being l_1, l_2 and l_3. Then the *lst*-predicate would be true for both the state s_2 and s_3, which is incorrect. However, constraint (5.4) rules out that path. In fact, as elaborated below, constraint (5.4) plays a crucial role in proving the soundness of constraint (5.8).

5.1.1 Consistency Check

If the constraints (5.1) to (5.9) above are instantiated for some LTSs and time steps 1 to k, some formula f is obtained. However, that formula has too many models, i.e., the encoding is still not sound. Extra models arise from three sources.

Firstly, no attention is paid to the order of the executed actions. The situation is illustrated on the left hand side of Figure 5.3. If only the schemes above were taken as a basis for the encoding, the three components would have an execution of length one where every transition is executed. However, their step product (and thus their limited path product) consists of a single state from which no transition is possible. This is due to the fact that every visible action is present in two components but in conflicting orders.

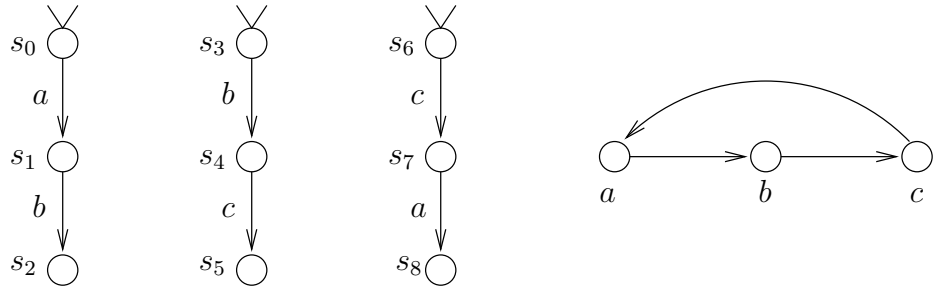

Figure 5.3: Inconsistent Executions and the Ordering Graph

Secondly, by constraint (5.3) if a transition is executed, then either its source state is reached or some incoming transition to its source state is executed. This opens up for the possibility that every transition "justifies" its execution by the fact that some incoming transition to its source state is executed. Then, provided that the synchronization requirements are respected, f has a model corresponding to an execution where unreached transition cycles are executed. For instance, if the constraints (5.1) to (5.9) are instantiated for the LTS on the left hand side of Figure 5.4 for bound 1, the resulting

formula has a model corresponding to an "execution", where the transitions drawn with a thicker line are executed. Obviously, the step product of the single LTS (which is the original LTS) has no such execution.

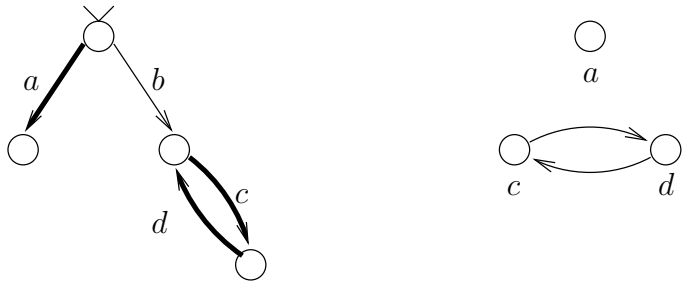

Figure 5.4: Unreachable Transition Cycles and the Ordering Graph

Finally, so far there has been no limitation on the number of executed transitions with the same label from a single component. However, the definition of a simple execution (Definition 44) limits this number to at most one.

A sound encoding of the limited path product needs an additional scheme that considers the order of the executed transitions. For each component, the executed path of transitions induces a sequence of transition labels. These sequences have to be *consistent*, i.e., it must be possible to choose a global ordering of all the actions appearing in the sequences so that all the local orderings are respected.

This consistency is modeled by constructing a global *ordering graph* based on the local label sequences. Its vertices are the actions appearing in the local sequences and there is an edge between two vertices a_i and a_j iff a_i occurs *immediately* before a_j in some local sequence. Then, the local label sequences are consistent iff the the ordering graph is acyclic. The formal definitions of the ordering graph and consistency are as follows:

Definition 48 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$ and let $\sigma_1, \ldots, \sigma_n$ be label sequences from the alphabets $\Sigma_1, \ldots, \Sigma_n$, respectively. The* ordering graph *$G = (V, E)$ of $\sigma_1, \ldots, \sigma_n$ is such that:*

1. *$V = (\Sigma_1 \cup \cdots \cup \Sigma_n) \setminus \{\tau\}$ and*
2. *$E = \{(a_i, a_j) \in V \times V \mid a_i$ occurs immediately before a_j in some σ_k, where $1 \leq k \leq n.\}$*

The definition of consistency in the context of the limited path product is then as follows:

Definition 49 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$ and let $\sigma_1, \ldots, \sigma_n$ be label sequences from the alphabets $\Sigma_1, \ldots, \Sigma_n$, respectively. The sequences $\sigma_1, \ldots, \sigma_n$ are* consistent *iff their ordering graph is acyclic.*

It should be easy to see that all the three sources of extra models are such that they induce cyclic ordering graphs. Therefore, the remaining element

of constructing a sound encoding for the limited path product is to create schemes for first modeling the ordering graphs for each step of the execution and then for ruling out cyclic graphs. The possible cycle in the ordering graph is detected by first computing the individual edges. These are then used to compute a reachability predicate. Finally, if the graph has a cycle, then every node in that cycle is reachable from itself. The edge computation has to take into account the possibility of a loop.

The idea is to define for each state-action pair the literals $inc(s_i, a_j, \mathfrak{t})$ and $og(s_i, a_j, \mathfrak{t})$ whose semantics are that an incoming (outgoing) transition to (from) state s_i labeled a_j is executed, respectively. However, for reasons elaborated below, for the target state s of the last transition on the path, the literal $inc(s, a_j, \mathfrak{t})$ should not evaluate to true for any action a_j. The definitions of the literals $inc(s_i, a_j, \mathfrak{t})$ and $og(s_i, a_j, \mathfrak{t})$ are then as follows:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_i \in S_i} \Big(\bigwedge_{a_j \in \Sigma_i} \big(inc(s_i, a_j, \mathfrak{t}) \leftrightarrow \bigvee_{t_k \in prn(s_i) \cap \Delta_i^{a_j}} (ex(t_k, \mathfrak{t}) \wedge \neg ltr(t_k, \mathfrak{t})) \big) \big) \big) \tag{5.10}$$

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{s_i \in S_i} \Big(\bigwedge_{a_j \in \Sigma_i} \big(og(s_i, a_j, \mathfrak{t}) \leftrightarrow \bigvee_{t_k \in pt(s_i) \cap \Delta_i^{a_j}} ex(t_k, \mathfrak{t}) \big) \big) \big) \tag{5.11}$$

The edges of the graph are then encoded using the literal $ed(a_i, a_j, \mathfrak{t})$ whose definition is as follows:

$$\bigwedge_{a_i, a_j \in \Sigma \setminus \{\tau\}} \big(ed(a_i, a_j, \mathfrak{t}) \leftrightarrow \bigvee_{1 \leq k \leq n} \Big(\bigvee_{s_l \in S_k} \big(inc(s_l, a_i, \mathfrak{t}) \wedge og(s_l, a_j, \mathfrak{t}) \big) \Big) \big) \tag{5.12}$$

The intuition behind the constraint above is that the graph contains an edge from a_i to a_j iff there is a local state s_l to which an incoming transition labeled a_i and an outgoing transition labeled a_j is executed.

Constraint (5.10) above contains an atomic proposition $ltr(t_k, \mathfrak{t})$ that has not been used so far. The intuitive idea of this proposition is that $ltr(t_k, \mathfrak{t})$ evaluates to true in the satisfying valuations of the final BMC formula iff transition t_k is the last executed transition on a local path. If the executed transition sequences were limited to straight segments, this literal could be left out from the conjunction $ex(t_k, \mathfrak{t}) \wedge \neg ltr(t_k, \mathfrak{t})$. However, with looping paths this would result in that an extra edge would be inferred. The situation is illustrated in Figure 5.5. Consider the case where the local path of executed transitions is such that the transitions labeled a, b, and c are executed in that order. If constraint (5.12) were implemented without the $ltr(t_k, \mathfrak{t})$ literal, the $ed(a_i, a_j, \mathfrak{t})$ literal would in this case be true for the action pairs (a, b), (b, c), and (c, b) of which the last one is incorrect.

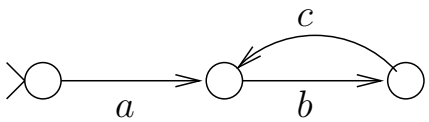

Figure 5.5: Literal $ltr(t_k, \mathfrak{t})$.

The literal $ltr(t_k, \mathfrak{t})$ is constrained based on the reasoning presented below. Firstly, if the executed transition is a self-loop, then it is the last transition on the path. Secondly, if a transition is the last transition on a path, then it has to be executed and its target state has to be the last state on the path. Finally, if a component is scheduled, then exactly one transition is the last transition on the path. Let the set of transitions that are self-loops be defined as follows:

Definition 50 *Let $\mathcal{L} = \langle S, I, \Sigma, \Delta \rangle$ be an LTS. Then the set $sl(\mathcal{L}) = \{t_i \in \Delta \mid t_i = (s, a, s)\}$, i.e., the set of transitions from $\mathcal{L}$ that are self-loops.*

The atomic proposition $ltr(t_k, \mathfrak{t})$ literal is then constrained as follows:

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{t_j \in sl(\mathcal{L}_i)} (ex(t_j, \mathfrak{t}) \rightarrow ltr(t_j, \mathfrak{t})) \Big), \tag{5.13}$$

$$\bigwedge_{1 \leq i \leq n} \Big(\bigwedge_{t_j \in \Delta_i} (ltr(t_j, \mathfrak{t}) \rightarrow ex(t_j, \mathfrak{t}) \wedge lst(tar(t_j), \mathfrak{t})) \Big), \text{ and} \tag{5.14}$$

$$\bigwedge_{1 \leq i \leq n} (sc(i, \mathfrak{t}) \rightarrow \text{card}_1^1\{ltr(t_j, \mathfrak{t}) \mid t_j \in \Delta_i\}), \tag{5.15}$$

where

- $tar(t_j)$ is the target state of transition t_j.

Graphs with cycles are then ruled out by modeling reachability from a single action to another by transitive edges of the graph and by constraining the models in such a way that no action is reached from itself. An action is reachable from another iff there is a direct edge or some transitive edge:

$$\bigwedge_{a_i, a_j \in \Sigma \setminus \{\tau\}} (rch(a_i, a_j, \mathfrak{t}) \leftrightarrow (ed(a_i, a_j, \mathfrak{t}) \vee \bigvee_{a_k \in \Sigma \setminus \{\tau\}} (rch(a_i, a_k, \mathfrak{t}) \wedge ed(a_k, a_j, \mathfrak{t})))). \tag{5.16}$$

Finally, to disallow cyclic graphs, no action should be reachable from itself. Formally:

$$\bigwedge_{a_i \in \Sigma \setminus \{\tau\}} \neg rch(a_i, a_i, \mathfrak{t}). \tag{5.17}$$

In the following, the constraints used to encode the consistency check (constraints (5.16) and (5.17)) are proved sound, i.e., it is shown that if the $ed(a_1, a_2, \mathfrak{t})$ literal is used to encode the edges of a graph, constraints (5.16) and (5.17) rule out graphs that are cyclic. The idea is based on the following proposition.

Proposition 1 *A graph is* cyclic *iff the irreflexive transitive closure of its edge relation has an element (v_i, v_j) such that $v_i = v_j$.*

In the following, Γ denotes a set of actions and f a formula containing the literal $ed(a_i, a_j, \mathfrak{t})$ for every $(a_i, a_j) \in \Gamma \times \Gamma$ and every $1 \leq \mathfrak{t} \leq k$. Furthermore, f' denotes the formula where constraint (5.16) is instantiated for every action pair $(a_i, a_j) \in \Gamma \times \Gamma$ and every $1 \leq \mathfrak{t} \leq k$.

Lemma 11 *Let $\mathcal{V}$ be a satisfying valuation of f and $G^{\mathfrak{t}} = (V^{\mathfrak{t}}, E^{\mathfrak{t}})$ the graph such that $V^{\mathfrak{t}} = \Gamma$ and $(a_i, a_j) \in E^{\mathfrak{t}}$ iff $\mathcal{V}(ed(a_i, a_j, \mathfrak{t})) = true$. Then for every satisfying valuation $\mathcal{V}'$ of the formula $f \wedge f'$ such that $\mathcal{V} \subset \mathcal{V}'$ it holds that if the irreflexive transitive closure of $G^{\mathfrak{t}}$ has the edge (a_i, a_j), then $\mathcal{V}'(rch(a_i, a_j, \mathfrak{t})) = true$.*

Proof. The goal is to show that if a formula $f \wedge f'$ where f models the graph $G^{\mathfrak{t}}$ and f' contains the instantiations of constraint (5.16) has a satisfying valuation, then the literal $rch(a_i, a_j, \mathfrak{t})$ models an overapproximation of the irreflexive transitive closure of $G^{\mathfrak{t}}$.

Firstly, it holds that if $\mathcal{V}'(ed(a_i, a_j, \mathfrak{t})) = \text{true}$, then $\mathcal{V}'(rch(a_i, a_j, \mathfrak{t})) = \text{true}$ by the first argument of the disjunction on the right hand side of constraint (5.16). Since this holds, then, again by the second disjunct of (5.16), $\mathcal{V}'(rch(a_i, a_k, \mathfrak{t})) = \text{true}$ for action pairs connected with paths consisting of two transitions in $G^{\mathfrak{t}}$. However, then $\mathcal{V}_t(rch(a_i, a_l, \mathfrak{t})) = \text{true}$ also for action pairs corresponding to paths consisting of three transitions and so on.

Constraint (5.16) does not *precisely* model the irreflexive transitive closure of graphs $G^{\mathfrak{t}}$, $1 \leq \mathfrak{t} \leq k$, though. Let a satisfying valuation $\mathcal{V}$ of f model the graph presented in Figure 5.6. Then, $f \wedge f'$ has a satisfying val-

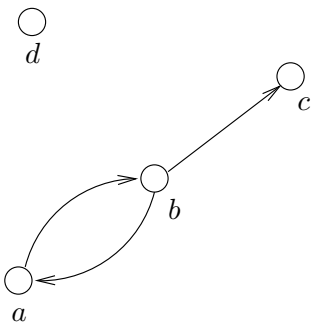

Figure 5.6: Cyclic Ordering Graph

uation where the $rch(a_1, a_2, \mathfrak{t})$ is true for the action pairs (d, a), (d, b) and (d, c), i.e., from a vertice d not connected to the rest of the graph to its every vertice.

This problem arises from the fact that f' contains, for instance, the following elements:

$$\begin{aligned} rch(d, a, \mathfrak{t}) &\leftrightarrow \ldots (rch(d, b, \mathfrak{t}) \wedge ed(b, a, \mathfrak{t})) \\ rch(d, b, \mathfrak{t}) &\leftrightarrow \ldots (rch(d, a, \mathfrak{t}) \wedge ed(a, b, \mathfrak{t})) \end{aligned}$$

Due to the fact that the graph contains the edges (a, b) and (b, a), the definitions of the reachability predicates above can be simplified to $rch(d, a, \mathfrak{t}) \leftrightarrow rch(d, b, \mathfrak{t})$ and $rch(d, b, \mathfrak{t}) \leftrightarrow rch(d, a, \mathfrak{t})$. Thus, both can either be false (the correct irreflexive transitive closure) or true. In the general case, this problem extends to every vertice outside a cycle in the graph and unfortunately, for larger graphs $f \wedge f'$ can have many models that are overapproximations of the irreflexive transitive closure of $G^{\mathfrak{t}}$.

However, if the graph is acyclic, constraint (5.16) models precisely its irreflexive transitive closure. Let f model an acyclic graph $G^{\mathfrak{t}} = (V^{\mathfrak{t}}, E^{\mathfrak{t}})$, f' defined as previously, and $\mathcal{V}'$ be a satisfying valuation of $f \wedge f'$.

Since $G^{\mathfrak{t}}$ is acyclic, then it has one or more vertices with no incoming edges. By constraint (5.16), for all vertices a_i with no incoming edges, it holds that $\mathcal{V}'(rch(a_j, a_i, \mathfrak{t})) = \text{false}$ for every vertice a_j in $G^{\mathfrak{t}}$. Thus, in the models of f', the rch-literal is correctly evaluated for transitive edges whose target state is a_i.

Consider then the graph $G^{\mathfrak{t}'} = (V^{\mathfrak{t}'}, E^{\mathfrak{t}'})$ obtained from $G^{\mathfrak{t}}$ by removing all the vertices with no incoming edges. Since $G^{\mathfrak{t}}$ is acyclic, then also $G^{\mathfrak{t}'}$ has vertices with no incoming edges. For such vertices, their only incoming edges are from vertices $a_i \in V^{\mathfrak{t}} \setminus V^{\mathfrak{t}'}$. Thus, for the vertices in $G^{\mathfrak{t}'}$ with no incoming edges a_k, constraint (5.16) reduces to:

$$\bigvee_{a_j \in \Sigma \setminus \{\tau\}} rch(a_j, a_k, \mathfrak{t}) \leftrightarrow ed(a_j, a_k, \mathfrak{t}) \vee \bigvee_{a_i \in V^{\mathfrak{t}} \setminus V^{\mathfrak{t}'}} (rch(a_j, a_i, \mathfrak{t}) \wedge ed(a_i, a_k, \mathfrak{t})).$$

The crucial point in the formula above is that the literals $rch(a_j, a_i, \mathfrak{t})$ are evaluated correctly (in this case to false for every a_j). From this it follows that the irreflexive transitive closure modeled by the rch-literal can not contain false edges whose target state is a_k.

Due to acyclicity, removing the edges a_k from $G^{\mathfrak{t}'}$ introduces again vertices a_l with no incoming edges (or alternatively, whose only incoming edges are from already removed vertices). For these nodes it can again be shown that the literal $rch(a_j, a_l, \mathfrak{t})$ is correctly evaluated since an instance can be true iff some other instance of the rch-literal is true and secondly it holds that that particular instance is correctly evaluated. This argument can be repeated until all the nodes of $G^{\mathfrak{t}}$ have been analyzed. □

To show the correctness of the formulas used to encode the consistency check, let Γ denote a set of actions and f a formula containing the literal $ed(a_i, a_j, \mathfrak{t})$ for every $(a_i, a_j) \in \Gamma \times \Gamma$ and every $1 \leq \mathfrak{t} \leq k$. Furthermore, let f' denote the formula where constraint (5.16) is instantiated for every action pair $(a_i, a_j) \in \Gamma \times \Gamma$ and every $1 \leq \mathfrak{t} \leq k$ and finally f'' the formula containing instantiations of constraint (5.17) for all actions $a_i \in \Gamma$ and all $1 \leq \mathfrak{t} \leq k$.

Lemma 12 *Let $\mathcal{V}$ be a satisfying valuation of the formula $f \wedge f' \wedge f''$ and for all $1 \leq \mathfrak{t} \leq k$, $G^{\mathfrak{t}} = (V^{\mathfrak{t}}, E^{\mathfrak{t}})$ be the graph such that $V^{\mathfrak{t}} = \Gamma$ and $(a_i, a_j) \in E^{\mathfrak{t}}$ iff $\mathcal{V}(ed(a_i, a_j, \mathfrak{t})) = \text{true}$. $\mathcal{V}$ is a* model *of $f \wedge f' \wedge f''$ iff every $G^{\mathfrak{t}}$ is acyclic.*

Proof. Follows directly from Proposition 1 and Lemma 11, and the fact that the rch-literal models the exact irreflexive transitive closure for acyclic graphs. □

Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and a bound k, let the formula encoding the executions of length k of the limited path product of the components be denoted $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. It is of the following form:

$$PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k) = IS(\mathcal{L}_1, \ldots, \mathcal{L}_n) \wedge \bigwedge_{1 \leq i \leq k} TR_{pt}(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$$

where

- $IS(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is the constraint on initial state, i.e., a conjunction of the constraints (5.1) and (5.2) and
- $TR_{pt}(\mathcal{L}_1, \ldots, \mathcal{L}_n, i)$ encodes the transition relation from state i to $i+1$ and is a conjunction of the constraints from (5.3) to (5.17) where the variable $\mathfrak{t}$ is instantiated to i.

Compared to the execution models applying partial order semantics or on-the-fly determinization, the size of the encoding for the limited path product is no more linear. The translation up to the consistency check (constraints (5.1) to (5.9)) are linear but for constraints (5.10), (5.11), (5.12), and (5.16) used to encode the consistency check this is not the case.

It may seem that constraints (5.10) and (5.11) defining the $inc(s_i, a_j, \mathfrak{t})$ and $og(s_i, a_j, \mathfrak{t})$ literals have to be instantiated for every state - action pair. However, this can be optimized by considering only such state - action pairs where actual incoming (outgoing) transitions with a particular label exist. If this is done, then a tighter bound is obtained, namely, the size of constraint (5.10) is $\mathcal{O}(|S| \cdot n_i)$ where $|S|$ is the number of local state and n_i the maximum indegree, i.e., the maximum number of incoming transitions of a local state of the system. Similarly, the size of constraint (5.11) is $\mathcal{O}(|S| \cdot n_j)$ where n_j is the maximum outdegree of a local state of the system.

Constraint (5.12) defines the literal $ed(a_i, a_j, \mathfrak{t})$ encoding the edges in the ordering graphs. Thus, it has to be instantiated for every action pair for which there exists a local state with an incoming transition labeled a_i and an outgoing transition labeled a_j. The size of the entire constraint is therefore $\mathcal{O}(|S| \cdot n_i \cdot n_j)$ in the worst case. Finally, constraint (5.16) encoding the $rch(a_i, a_j, \mathfrak{t})$ literal is also not linear. This constraint needs to be instantiated for every action pair Furthermore, on the right-hand side, a third action is needed. Thus, the entire constraint is cubic in the size of the alphabet, i.e., $\mathcal{O}(|\Sigma|^3)$ in the worst case.

Purpose of Preprocessing

The reason for applying the preprocessing algorithm given in Section 5 is that otherwise the encoding of the consistency check given in Section 5.1.1 does not work properly. Namely, if an LTS can contain τ transitions that are not self-loops, it may, for instance, execute a transition sequence with following labels: $a\tau\tau b$. The internal τ transitions do not have any synchronization requirements and should therefore not contribute to the ordering graph of the sequence but the graph should contain an edge between the vertices labeled a and b. However, the literal $ed(a, b, \mathfrak{t})$ in constraint (5.12) evaluates to true iff a transition labeled a is executed immediately before a transition labeled b. Since the τ transitions are limited to self-loops, the local state is not changed when such a transition is executed and thus constraint (5.12) is sound.

Obviously, constraint (5.12) could be implemented differently by considering paths of the form $a\tau^*b$ (where τ^* denotes zero or more τ labels). One idea to implement such a scheme is to propagate information about the last executed visible transition over the following τ transitions and then infer a

new edge to the graph when a visible action is again encountered. However, in that case constraint (5.12), even though still polynomial, would become more complex.

Another way to avoid the preprocessing algorithm is to consider every internal transition as actually begin labeled with a unique visible label. However, the adverse effect of this approach is that the alphabet of the systems grows with the number of internal transitions. This, on the other hand affects adversely the size of constraint (5.16) that is cubic in the alphabet size.

Soundness and Completeness

In the following, the soundness and completeness of the encoding of the executions of the limited path product is established. The presentation follows the same ideas as in Chapter 4 for the execution models applying partial order semantics and on-the-fly determinization.

The soundness of the encoding is established using two auxiliary lemmata. Firstly, it is shown that if the reached states and executed transitions for each global state and execution step, respectively, are read from a satisfying valuation of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, certain kind of sequences are obtained (Lemma 14).

These sequences (defined in Definition 52) are such that in each global state, each component is in exactly one local state. In each step, the executed local transitions from each component form a path from the source state of the first transition to the target state of the last transition.

Then, in Lemma 15, it is shown that these local transition segments can be combined to a step execution segment. Finally, in Theorem 19, it is shown that the first execution segment starts from a initial state of the limited path product and that all the step execution segments are simple. Thus, from all satisfying valuations of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, an execution of the limited path product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$ is obtained.

The proof of completeness is similar, i.e., uses a mapping from an execution to the literals of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ and shows that with this mapping, all the constraints in $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ are satisfied.

Definition 51 *Given the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and an integer k, let $\mathcal{V}_t$ be a valuation of any formula containing the following atomic propositions:*

- *for all $s_j \in S_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k+1$, the proposition $in(s_j, \mathfrak{t})$ and*
- *for all $t_j \in \Delta_i, 1 \leq i \leq n$ and all $1 \leq \mathfrak{t} \leq k$, the proposition $ex(t_j, \mathfrak{t})$.*

*Then, the $\mathcal{V}_t$-*sequence *corresponding to $\mathcal{V}_t$ is the sequence:*

$$\langle T_1^1, \ldots, T_n^1 \rangle \overset{\langle L_1^1, \ldots, L_n^1 \rangle}{\rightarrow} \cdots \overset{\langle L_1^k, \ldots, L_n^k \rangle}{\rightarrow} \langle T_1^{k+1}, \ldots, T_n^{k+1} \rangle$$

such that each $T_i^{\mathfrak{t}}$ is a set of states and a component state $s_k \in T_i^{\mathfrak{t}}$ iff $s_k \in S_i$ and $\mathcal{V}_t(in(s_k, \mathfrak{t})) = true$. Each $L_i^{\mathfrak{t}}$ is a set of local transitions and transition t_l is in $L_i^{\mathfrak{t}}$ iff $t_l \in \Delta_i$ and $\mathcal{V}_t(ex(t_l, \mathfrak{t})) = true$.

Definition 52 *Given the $\mathcal{V}_t$-sequence corresponding to a satisfying valuation $\mathcal{V}_t$ of the formula $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, it is a* path sequence *iff the following conditions hold:*

1. *for all* $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k+1$, $|T_i^{\mathfrak{t}}| = 1$ *and*

2. *for all* $1 \leq i \leq n, 1 \leq \mathfrak{t} \leq k$, *the transitions in* $L_i^{\mathfrak{t}}$ *form a connected sequence from the unique state forming* $T_i^{\mathfrak{t}}$ *to the unique state forming* $T_i^{\mathfrak{t}+1}$.

Lemma 13 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs and* σ_{M} *a* $\mathcal{V}_t$*-sequence of length* k *corresponding to a satisfying valuation* $\mathcal{V}_t$ *of* $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. *If* $|T_i^{\mathfrak{t}}| = 1$ *for some* $1 \leq i \leq n$ *and* $1 \leq \mathfrak{t} \leq k$ *in* σ_{M}, *then the local transitions in* $L_i^{\mathfrak{t}}$ *form a single path that is either (i) a straight segment, (ii) a loop to the initial state, or (iii) a lasso-shaped path.*

Proof. By constraint (5.4), at most one outgoing transition of a state can be executed. Thus, the executed transitions divide to connected sequences that do not have common states.

By constraint (5.3), if a transition is executed in step $\mathfrak{t}$, then either its source state is reached in the previous global state or some incoming transition to its source state is executed in the same step. If for component $\mathcal{L}_i$, $|T_i^{\mathfrak{t}}| = 1$, it follows that the connected sequences of executed transitions are such that there is precisely one sequence for which it holds that the source state of the first transition is reached in the previous global state. By constraint (5.3), all the remaining sequences have to form cycles (referred to as *unreachable* cycles).

It remains to show that if $\mathcal{V}_t$ is a satisfying valuation of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, no unreachable cycles are possible. Firstly, if for all the transitions t_j in such a cycle in time step $\mathfrak{t}$ it holds that $\mathcal{V}_t(ltr(t_j, \mathfrak{t})) = \text{false}$, then by constraint (5.12), a cyclic ordering graph is induced. Then, by Lemma 12, $\mathcal{V}_t$ can not be a satisfying valuation of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.

Therefore, it has to be the case that for all unreachable cycles, there has to be at least one transition t_j in the cycle for which $\mathcal{V}_t(ltr(t_j, \mathfrak{t})) = \text{true}$. Then however, by constraint (5.14), $\mathcal{V}_t(lst(tar(t_j), \mathfrak{t}))$ has to be true. This, however, can not be the case by constraint (5.8). This is shown by a case analysis of the three cases of constraint (5.8). For $tar(t_j)$ it holds that (i) both an incoming and an outgoing transition to it is executed, (ii) by assumption of the cycle being unreachable, $\mathcal{V}_t(in(tar(t_j), \mathfrak{t}-1))$ must be false, and (iii) exactly one of its incoming transitions are executed. Therefore, $\mathcal{V}_t(lst(tar(t_j), \mathfrak{t}))$ has to be false, a contradiction. From this it follows that no unreachable cycles can exist in the satisfying valuations of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. □

Lemma 14 *Let* $\mathcal{L}_1, \ldots, \mathcal{L}_n$ *be LTSs. Given any satisfying valuation* $\mathcal{V}_t$ *of the formula* $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, *then the corresponding* $\mathcal{V}_t$*-sequence is a path sequence.*

Proof. The following proof establishes by induction that in each component $1 \leq i \leq n$ and each global state $1 \leq \mathfrak{t} \leq k+1$, $|T_i^{\mathfrak{t}}| = 1$ and for each $1 \leq \mathfrak{t}' \leq k$, if some local transitions are executed from $\mathcal{L}_i$ the single state forming $T_i^{\mathfrak{t}'+1}$ is the target state of the last transition in the connected sequence of transitions in the set $L_i^{\mathfrak{t}'}$. Otherwise $T_i^{\mathfrak{t}'+1} = T_i^{\mathfrak{t}'}$.

1. **Base Case.** Constraints (5.1) and (5.2) guarantee that in the first global state, the local state from each component is unique.

2. **Induction Hypothesis.** Assume that up to some global state l all the local states are unique.

3. **Induction Step.** Consider the step

$$\langle T_1^l, \ldots, T_n^l \rangle \overset{\langle L_1^l, \ldots, L_n^l \rangle}{\rightarrow} \langle T_1^{l+1}, \ldots, T_n^{l+1} \rangle.$$

By induction hypothesis, it holds that for all $1 \leq i \leq n$, $|T_i^l| = 1$. By Lemma 13, it follows that transitions in step l form a single path that is either (i) a straight segment, (ii) a loop to the initial state, or (iii) a lasso-shaped path. It is shown that for all $1 \leq i \leq n$, $|T_i^{l+1}| = 1$ and that if L_i^l is non-empty (component $\mathcal{L}_i$ is scheduled), then the single local state forming it is the target state of the last state on the local path of executed transitions in step l. Otherwise, $T_1^{l+1} = T_1^l$.

The set T_i^{l+1} is defined from the literal $in(s_j, l+1)$ that is defined in constraint (5.7). The treatment is divided to idle and scheduled components (for which the $sc(i,j)$ literal is false and true, respectively). For idle components, constraint (5.7) reduces to:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow in(s_j, l)).$$

Thus, idle components remain in the same state that is unique by induction hypothesis. For scheduled components, the same constraint reduces to:

$$\bigwedge_{s_j \in S_i} (in(s_j, l+1) \leftrightarrow lst(s_j, l)).$$

It remains to show that, given the form of the path of executed transitions, the $lst(s_j, l+1)$ literal (defined in constraint (5.8)) evaluates to true for exactly one local state from each component. Firstly, it holds that for all the local states not along the single path, no incoming nor any outgoing transitions are executed. Thus, for any such state s_j, the literal $lst(s_j, l+1)$ is false.

If the executed path is (i) a straight segment, it holds that no incoming transition to the local state reached in the global state l is executed (otherwise the path would be a loop). Furthermore, there is no local state such that more than one of its incoming transitions are executed (since then the executed path would be a lasso). However, there is exactly one state such that one incoming and no outgoing transitions are executed, the last state on the path. That is also the only state for which $lst(s_j, l)$ is true.

If the executed path is (ii) a loop to the initial state, then there is no local state along the path such that no outgoing transitions of that state are executed. Neither is there a local state along the path such that more than one incoming transitions to that state are executed. However, an incoming transition to the state reached in step l is executed, the last state on the path. That is also the only state for which $lst(s_j, l)$ is true.

If the executed path is (iii) lasso shaped, then there is no local state along the path such that no outgoing transitions of that state is executed. Neither is an incoming transition to the local state reached in step l executed. However, there is a state such that two of its incoming transitions are executed, the last state on the path. That is also the only state for which $lst(s_j, l)$ is true.

Thus, in every case, there is exactly one local state from each component for which $lst(s_j, l)$ and thus $in(s_j, l+1)$ is true. That state is the target state of the last executed transition on the local path.

□

For every $1 \leq i \leq n$, $1 \leq \mathfrak{t} \leq k$, let the single element of $T_i^{\mathfrak{t}}$ be denoted $s_i^{\mathfrak{t}}$. The path sequence is still rather far away from an execution of the limited path product. In order to prove the soundness of the encoding, it has to be shown that the local connected sequences of transitions form a simple step execution. This is shown in two phases. Firstly, in Lemma 15, it is shown that the local connected sequences form a step execution and finally, in Theorem 19 it is shown that this execution is simple.

Lemma 15 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $\mathcal{V}_t$ a satisfying valuation of the formula $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ and*

$$\langle s_1^1, \ldots, s_n^1 \rangle \overset{\langle L_1^1, \ldots, L_n^1 \rangle}{\rightarrow} \cdots \overset{\langle L_1^k, \ldots, L_n^k \rangle}{\rightarrow} \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$$

the path sequence corresponding to $\mathcal{V}_t$. Let $\mathcal{L}_{st} = \mathcal{L}_1 \parallel_{st} \ldots \parallel_{st} \mathcal{L}_n$. It holds that for all $1 \leq \mathfrak{t} \leq k$, $\mathcal{L}_{st}$ contains a path from $\langle s_1^{\mathfrak{t}}, \ldots, s_n^{\mathfrak{t}} \rangle$ to $\langle s_1^{\mathfrak{t}+1}, \ldots, s_n^{\mathfrak{t}+1} \rangle$ such that for all $1 \leq i \leq n$, the executed local transitions from component $\mathcal{L}_i$ are $L_i^{\mathfrak{t}}$.

Proof. By Lemma 14, the transitions in $L_1^{\mathfrak{t}}, \ldots, L_n^{\mathfrak{t}}$ form connected sequences in the respective components. Furthermore, for every component $1 \leq i \leq n$, the source state of the first transition in the sequence $L_i^{\mathfrak{t}}$ is $s_i^{\mathfrak{t}}$ and the target state of the last transition is $s_i^{\mathfrak{t}+1}$. The claim of Lemma 15 follows from the fact that transition sequences $L_1^{\mathfrak{t}}, \ldots, L_n^{\mathfrak{t}}$ (i) respect the synchronization requirement in the definition of the step product and (ii) are consistent.

The first condition follows from the fact that $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ contains constraint (5.6) that guarantees that if action a is executed, then every component executes a transition labeled a. The second condition (on the consistency of the execution) is guaranteed by the following two facts.

Firstly, if the ordering graph of the executed local sequences in step $\mathfrak{t}$ contains an edge (a_i, a_j), then $\mathcal{V}_t(ed(a_i, a_j, \mathfrak{t})) = \text{true}$. Secondly, the ordering graph that the local sequences form is acyclic.

The first condition is shown as follows. If the ordering graph contains the edge (a_i, a_j), then there is a local state s for which an incoming transition labeled a_i and an outgoing transition labeled a_j is executed. By constraint (5.11) $\mathcal{V}_t(og(s, a_j, \mathfrak{t})) = \text{true}$. By constraint (5.10), $\mathcal{V}_t(inc(s, a_i, \mathfrak{t})) = \text{true}$ iff for the incoming transition t labeled a_i, it holds that $\mathcal{V}_t(ltr(t, \mathfrak{t})) = \text{false}$.

Thus, to show that $\mathcal{V}_t(inc(s, a_i, \mathfrak{t}))$ is indeed true, it has to be shown that $\mathcal{V}_t(ltr(t, \mathfrak{t})) = \text{true}$ iff t is the last executed transition on the path. In Lemma 14, it is shown that executed transitions are either straight segments, loops to the initial state, or lasso-shaped. By constraint (5.14), if $\mathcal{V}_t(ltr(t, \mathfrak{t})) = \text{true}$, then $\mathcal{V}_t(lst(tar(t), \mathfrak{t})) = \text{true}$. It has already been shown in the proof of Lemma 14 that if this it the case, then $tar(t)$ has to be the target state of the last executed transition.

If the executed path is a straight segment or a loop to the initial state, there is only one transition, for which the target state is the last state on the path and therefore only one possible transition t for which $\mathcal{V}_t(ltr(t, \mathfrak{t})) = \text{true}$. This is the last transition on the path. If the executed path is lasso-shaped, there are *two* transitions whose target state is the last state on the path. It has to be shown that in satisfying valuations of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, the literal $ltr(t, \mathfrak{t})$ can only be true for the correct one. Firstly, by constraint (5.15), $ltr(t, \mathfrak{t})$ can not be true for both of them. Secondly, if this literal was true for the first transition on the path entering the last state of the path, then the remaining segment would form a cycle. In this cycle, for all the transitions t_k, $\mathcal{V}_t(ltr(t_k, \mathfrak{t})) = \text{false}$. However, then $\mathcal{V}_t$ can not be a satisfying valuation of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ since constraint (5.17) is violated. Therefore, $\mathcal{V}_t(ltr(t, \mathfrak{t})) = \text{true}$ for the last transition also in this case.

Thus, the literal $ed(a_i, a_j, \mathfrak{t})$ models the edges of the ordering graph. That the graph is acyclic is guaranteed by Lemma 12. □

Given a satisfying valuation $\mathcal{V}_t$ of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$, let the execution segment of $\mathcal{L}_{st}$ for $1 \leq \mathfrak{t} \leq k$ be $\sigma^{\mathfrak{t}}$.

Definition 53 *The $\mathcal{V}_t$-sequence corresponding to a satisfying valuation $\mathcal{V}_t$ of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is an execution of the limited path product $\mathcal{L}_{pt} = \langle S_{pt}, I_{pt}, \Sigma_{pt}, \Delta_{pt} \rangle$ of the components $\mathcal{L}_1, \ldots, \mathcal{L}_n$ iff*

1. *for all components $1 \leq i \leq n$, $s_i^1 \in I_i$ and*

2. *for all $1 \leq \mathfrak{t} \leq k$, $\langle s_1^{\mathfrak{t}}, \ldots, s_n^{\mathfrak{t}} \rangle \xrightarrow{\sigma^{\mathfrak{t}}} \langle s_1^{\mathfrak{t}+1}, \ldots, s_n^{\mathfrak{t}+1} \rangle \in \Delta_{pt}$.*

Theorem 19 *Let $\mathcal{V}_t$ be a satisfying valuation of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$. Then the corresponding $\mathcal{V}_t$-sequence is an execution of the limited path product of $\mathcal{L}_1, \ldots, \mathcal{L}_k$.*

Proof. In Lemma 15 it is shown that for all $1 \leq \mathfrak{t} \leq k$, $\sigma^{\mathfrak{t}}$ is an execution segment in the step product $\mathcal{L}_{st}$ of $\mathcal{L}_1, \ldots, \mathcal{L}_n$. Theorem 19 is established by showing that (i) σ^1 starts from an initial state of $\mathcal{L}_{st}$ and that (ii) all the execution segments $\sigma^{\mathfrak{t}}$ are simple. The first condition follows from the fact that $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ contains the constraints (5.1) and (5.2). Furthermore, for all $1 \leq j \leq n$ it holds that the $in(s_k, 1)$ literal has to evaluate to true for the source state s_k of the first transition in the connected sequence of transitions in L_j^1.

The second condition is established by showing that the executions $\sigma^{\mathfrak{t}}$ fulfill the requirements in Definition 44. Firstly, for all $\sigma^{\mathfrak{t}}$ it holds that if some local state s_k is traversed twice, s_k has to be the last state on the local path. This follows from constraint (5.8). Furthermore, it holds for all the visible

actions a of $\Sigma_1 \cup \cdots \cup \Sigma_n$ that $\sigma^{\mathfrak{t}}$ can contain at most one action tuple where the local action a is present. This follows from constraints (5.12), (5.16), and (5.17) modeling the ordering graph and (an overapproximation of) its irreflexive transitive closure, and ruling out cyclic graphs. If some $\sigma^{\mathfrak{t}}$ contained two action tuples containing action a, then some component $\mathcal{L}_j$ would execute a transition labeled a twice. However, then the ordering graph $G^{\mathfrak{t}}$ is necessarily cyclic and constraint (5.17) violated. Thus, for all $1 \leq \mathfrak{t} \leq k$, $\sigma^{\mathfrak{t}}$ is simple. □

The completeness of the encoding is shown in the following. The proof uses the following concept.

Definition 54 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and $\mathcal{L}_{st}$ their step product. Furthermore, let σ be a finite execution of $\mathcal{L}_{st}$. Then for all $1 \leq i \leq n$, let $pa_i(\sigma) = t_1, \ldots, t_{j_i}$ be the sequence of local transitions executed from $\mathcal{L}_i$ in σ.*

Theorem 20 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs. Given any execution of the limited path product product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, it is a $\mathcal{V}_t$-sequence of some satisfying valuation $\mathcal{V}_t$ of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.*

Proof. The proof presents a mapping $\mathcal{V}_t$ from the execution to the atomic propositions of $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ and shows that if the truth values of the atomic propositions are defined based on $\mathcal{V}_t$, every constraint of the formula $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is satisfied. Any limited path execution of length k can be presented as follows:

$$\langle s_1^1, \ldots, s_n^1 \rangle \xrightarrow{\sigma^1} \cdots \xrightarrow{\sigma^k} \langle s_1^{k+1}, \ldots, s_1^{k+1} \rangle$$

where for all $1 \leq \mathfrak{t} \leq k$, $\sigma^{\mathfrak{t}}$ is a an execution of the step product of the component $\mathcal{L}_1, \ldots, \mathcal{L}_n$.

Let $\langle l_1^1, \ldots, l_n^1 \rangle, \ldots, \langle l_1^{p_{\mathfrak{t}}}, \ldots, l_n^{p_{\mathfrak{t}}} \rangle$ be the sequence of actions executed in $\sigma^{\mathfrak{t}}$. The notational convention of writing $t_i \in pa_j(\sigma^{\mathfrak{t}})$ for every transition t_i appearing in the sequence $pa_j(\sigma^{\mathfrak{t}})$ is applied. Similarly, for any label $\langle l_1^{\mathfrak{t}'}, \ldots, l_n^{\mathfrak{t}'} \rangle$ appearing in $\sigma^{\mathfrak{t}}$, $\langle l_1^{\mathfrak{t}'}, \ldots, l_n^{\mathfrak{t}'} \rangle \in \sigma^{\mathfrak{t}}$.

The mapping is as follows:

- $\mathcal{V}_t(in(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $s_i^{\mathfrak{t}} = s_j$,
- $\mathcal{V}_t(ex(t_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $t_j \in pa_i(\sigma^{\mathfrak{t}})$,
- $\mathcal{V}_t(ex(a, i, \mathfrak{t})) =$ true iff for some label $\langle l_1^j, \ldots, l_n^j \rangle \in \sigma^{\mathfrak{t}}$, $l_i^j = a$,
- $\mathcal{V}_t(ex(a, \mathfrak{t})) =$ true iff for some label $\langle l_1^i, \ldots, l_n^i \rangle \in \sigma^{\mathfrak{t}}$, for a $1 \leq j \leq n$ $l_j^i = a$,
- $\mathcal{V}_t(sc(i, \mathfrak{t})) =$ true iff there is a label $\langle l_1^j, \ldots, l_n^j \rangle \in \sigma^{\mathfrak{t}}$, such that $l_i^j \neq \epsilon$,
- $\mathcal{V}_t(lst(s_j, \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, $\mathcal{V}_t(sc(i, \mathfrak{t})) =$ true and $s_i^{\mathfrak{t}+1} = s_j$,
- $\mathcal{V}_t(ltr(t_i, \mathfrak{t})) =$ true iff for some $1 \leq j \leq n$, $t_i \in \Delta_j$ and $pa_j(\sigma^{\mathfrak{t}}) = t_1, \ldots, t_i$,

- $\mathcal{V}_t(inc(s_j, a, \mathfrak{t})) =$ true iff there is a transition $t_l = (s_m, a, s_j)$ such that $\mathcal{V}_t(ex(t_l, \mathfrak{t})) =$ true and it holds that $\mathcal{V}_t(ltr(t_l, \mathfrak{t})) =$ false,

- $\mathcal{V}_t(og(s_j, a, \mathfrak{t})) =$ true iff there is a transition $t_l = (s_j, a, s_m)$ such that $\mathcal{V}_t(ex(t_l, \mathfrak{t})) =$ true,

- $\mathcal{V}_t(ed(a, a', \mathfrak{t})) =$ true iff for some $1 \leq i \leq n$, there is a local state s_j such that it holds that s_j has an incoming transition t_k labeled a, $\mathcal{V}_t(ex(t_k, \mathfrak{t})) =$ true, and $\mathcal{V}_t(ltr(t_k, \mathfrak{t})) =$ false. Secondly, s_j has an outgoing transition t_l labeled a' and $\mathcal{V}_t(ex(t_l, \mathfrak{t})) =$ true, and

- $\mathcal{V}_t(rch(a, a', \mathfrak{t})) =$ true iff $(a, a') \in S^{\mathfrak{t}}$ where $S^{\mathfrak{t}} \subseteq \Sigma \times \Sigma$ is the smallest set such that:

  1. if $\mathcal{V}_t(ed(a, a', \mathfrak{t})) =$ true, then $(a, a') \in S^{\mathfrak{t}}$ and
  2. if $(a, a') \in S^{\mathfrak{t}}$ and $(a', a'') \in S^{\mathfrak{t}}$, then $(a, a'') \in S^{\mathfrak{t}}$.

The proof is by induction over the step $\mathfrak{t}$ in $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$.

1. **Base Case.** Any limited path execution starts from an initial state. Thus, the constraints (5.1) and (5.2) are satisfied.

2. **Induction Hypothesis.** Assume that every conjunct in the formula $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ is satisfied up to some global state l.

3. **Induction Step.** Consider the step

   $$\langle s_1^l, \ldots, s_n^l \rangle \xrightarrow{\sigma^l} \langle s_1^{l+1}, \ldots, s_n^{l+1} \rangle.$$

   The goal is to show that the mapping $\mathcal{V}_t$ is consistent with the formula $TR_{pt}(\mathcal{L}_1, \ldots, \mathcal{L}_n, l)$. The executed transitions in each step in each component form paths. For the first transition it is the case that its source state is reached and for the following transitions it holds that some incoming transition to their source state is executed. Thus, constraint (5.3) is satisfied. In addition, by the fact that the step executions in step l is simple, the executed path can either be a straight segment, loop to the initial state, or lasso-shaped. Thus, at most one outgoing transition from each state is executed and constraint (5.4) satisfied.

   An action is executed in a limited path execution iff some transition labeled with the action is executed. This adheres to constraint (5.5). In addition, the synchronization requirement is respected also in the limited path product and constraint (5.6) is satisfied.

   In any limited path execution after a particular step, the control flow reaches local states that are:

   - the same state as in the previous global state for idle components or
   - the target state of the last executed transition on the executed path.

Therefore, constraint (5.7) is satisfied by the presented mapping. An analysis of constraint (5.8) follows. Firstly, is a component $\mathcal{L}_i$ is not scheduled, then constraint (5.8) is satisfied by setting the $lst(s_j, l)$-literal to false for all the local states of $\mathcal{L}_i$, as done in the presented mapping.

If a component is scheduled, then if the path of executed transitions in some component forms a straight segment, then the constraint can be satisfied for all the local states of that component with the mapping above, i.e., setting the literal true for the target state of the last executed transition (one of its incoming and none of its outgoing transitions is executed) and false for the rest of the states.

Secondly, if the path of executed transitions forms a loop to the initial state, $lst(s_j, l)$ can be set to true for the last state of the path since its source state is reached in the global state l and one of its incoming transitions is executed. For all the remaining local states of that component, the literal can be set to false.

Finally, if the path of executed transitions forms a lasso, then the literal $lst(s_j, l)$ can be set to true for the last state of the path since two of its incoming transitions are executed. Again, for all the remaining local states of that component, the literal can be set to false.

Constraint (5.9) is satisfied. Constraints (5.10) and (5.11) are satisfied, since the literal $inc(s_i, a, l)$ (resp. $og(s_i, a, l)$) is mapped to true iff some incoming (outgoing) transition of s_i labeled a is executed. Constraint (5.13) is satisfied, since a transition is that is a self-loop can only be the last executed transition on the path. Constraint (5.14) is also satisfied, since if $t_i \in pa_j(\sigma^l)$, then $\mathcal{V}_t(ex(t_i, l)) = \text{true}$. In each scheduled component, only one transition is such that $\mathcal{V}_t(ltr(t_i, l)) = \text{true}$ and constraint (5.15) is satisfied.

Constraint (5.12) is satisfied and indeed, $\mathcal{V}_t(ed(a_i, a_j, l)) = \text{true}$ iff the ordering graph for time step l has an edge from a_i to a_j. All the executions of the limited path product are consistent and thus no vertice of the ordering graph is reachable from itself. Therefore, constraints (5.16) and (5.17) are satisfied.

□

So far, it has been established that $PA(\mathcal{L}_1, \ldots, \mathcal{L}_n, k)$ correctly encodes the executions of length k of the limited path product of $\mathcal{L}_1, \ldots, \mathcal{L}_n$. However, the main drawback of the consistency check is that it is cubic in the size of the alphabet probably rendering the complete encoding too complex compared to the interleaving, step and process models.

For these reasons, it is sensible to seek alternative approaches for encoding the consistency check (or the more general problem of finding a cycle in a graph using propositional satisfiability). The following section presents one such approach.

5.1.2 An Alternative Approach for Consistency Check

An alternative, possibly simpler way (proposed by K. Heljanko and independently by V. Khomenko [79]) of ruling out inconsistent executions is as follows. Given the executed transitions, introduce for every executed action an integer and require that the actions occurring earlier in a particular component have smaller numbers than those occurring later. If the execution is inconsistent, then conflicting requirements arise and the model candidate is ruled out. More formally, the following kind of scheme is required.

$$\bigwedge_{a_i, a_j \in \Sigma \setminus \{\tau\}} (ed(a_i, a_j, \mathfrak{t}) \rightarrow (f(a_i, \mathfrak{t}) < f(a_j, \mathfrak{t})))$$

With the encoding presented above, $\log_2(|\Sigma|)$ additional bits are needed for each visible action of the alphabet. Furthermore, a Boolean implementation of the $<$ operation on those bits encoding an integer value needs to be implemented. However, that operation can be implemented linearly in the size (number of bits) of the arguments.

Consider the execution in Figure 5.3. If every transition is executed, then the scheme above requires the following inequalities to hold: $f(a,1) < f(b,1), f(b,1) < f(c,1)$ and $f(c,1) < f(a,1)$. That is not possible for any valuation and the false execution is ruled out.

5.1.3 Iterative Approach

Encoding the consistency check completely as presented above can be seen as modeling all the possible ordering graphs for all time steps independent of the actually executed transitions. The analysis of test cases (presented in detail Chapter 6) warrants the following facts. With *these* cases

- the size of the resulting formula is in some cases dominated by the literals of the consistency check and
- the are examples where the spurious counterexamples that the check prevents do not materialize.

Thus, with the analyzed examples, some of the deadlocks are detected faster if the consistency check is completely dropped. In addition, the points above also suggest a procedure that analyzes the counterexamples and then if the counterexample is incorrect, adds additional clauses to the formula such that this spurious counterexample is not among the models.

The complete procedure is easy. The counterexample is analyzed by creating the ordering graph and checking its cyclicity. Secondly, a spurious counterexample is ruled out by instantiating the same schemes as presented above but only based on the executed transitions in the spurious counterexample. However, the complete story is not that simple, since it is theoretically easy to conceive an example that has a real deadlock but also many inconsistent deadlocking executions. This is done in Figure 5.7. The figure presents a parameterized example where the parameter n is the number of branches in the component on the left hand side. Branch i is labeled with the actions a_i, b_i. Each branch has a corresponding component LTS where the actions

are in reverse order. The example has one real deadlock (depth 1) and n spurious ones.

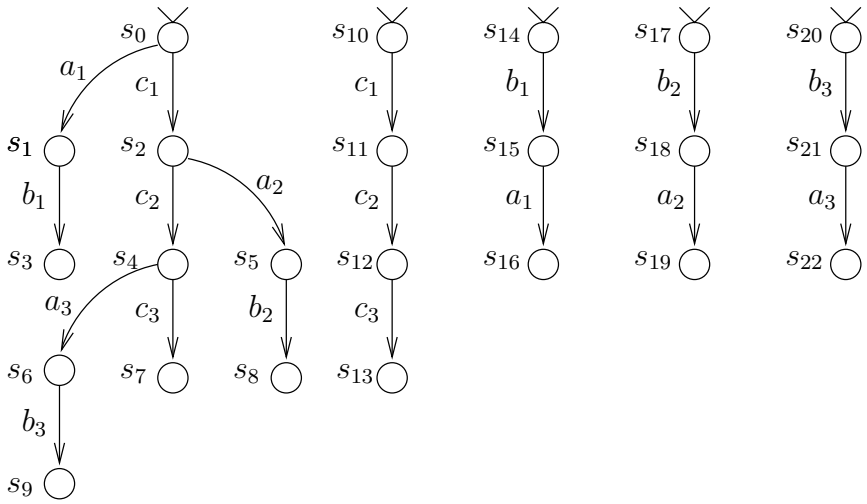

Figure 5.7: A Hard Example for Iterative Strengthening

The presented approach can be seen as a special case of counterexample guided abstraction refinement [23]. The abstraction is based on relaxing a particular requirement of the used non-standard execution model, namely the consistency check. Thus the initial formula may have models that do not correspond to the real executions of the system (spurious counterexamples). The initial formula is then automatically iteratively strengthened (the model is refined) with additional clauses until either a correct counterexample is found or the formula is proved unsatisfiable. The approach will in the worst case converge to the full consistency check formula in a polynomial number of iterations. See [50] for more details.

5.2 LIFTING CONSTRAINTS FROM LIMITED PATH PRODUCT

The limited path product has a transition between two states iff the step product of the same components has a *simple* execution between the two states. The following subsections analyze the problems encountered when a more general model is attempted. It is noticed that in many cases lifting the presented limitations renders the interpretation of the model harder, i.e., a model of the formula could correspond to several different executions. When this problem is addressed, it is noticed that the additional complexity outweighs the potential gains brought about by the possibly reduced bound.

5.2.1 Limitations on the Structure of the Path

The simple executions used in the definition of the transition relation of the limited path product are required to visit a particular component state at most once. This section examines why this limitation is used.

Figure 5.8 presents the two basic cases ruled out by the limitations. On the left hand side the idea is to examine what problems would be caused if an

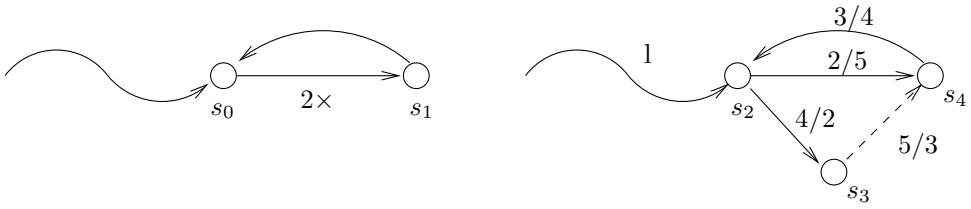

Figure 5.8: Paths Not Allowed

encoding is desired where it is possible to execute a path entering the local state s_0 from the left, proceeding along the transitions from s_0 to s_1, then back to s_0 and then along the transition from s_0 to s_1 a second time (the transition is marked with $2\times$).

The central problem arises when trying to maintain the uniqueness of the model and an easy mapping between the executed transitions and the true literals of the model of the resulting formula. If it is possible to execute a transition a second time there should be a literal true iff that is the case. Thus, an additional literal is needed for all the transitions. Indeed, if repetition of n times is desired, n literals are needed.

Thus, to be able to execute a loop twice to in the best case halve the needed bound you need to approximately double the number of literals. Any savings in the bound are therefore lost in the additional complexity of the encoding of the consistency check. Finally, to generally determine a suitable repetition count based on the structure of the system seems to be non-trivial.

On the right hand side of Figure 5.8 the idea is to illustrate the idea that a state (in this case s_2) is exited twice, but to *different* states. Consider first the case where the control flow proceeds from the transition labeled 1 along the numbers on the left hand side of the slash sign (/) and finally reaches the state s_3. Thus 4 transitions are executed and the visited states are s_2, s_4, s_2 and s_3. If such paths are allowed the simple cardinality constraint in scheme (5.4) would have to be replaced. It seems hard to conceive a simple scheme allowing this type of paths and still maintaining the soundness of the encoding, i.e. being able to rule out all the models that are not executions of the system.

The situation is even more complicated if the path above is extended adding the transition from s_3 to s_4 (the dashed line). Assuming the verifier is presented with a formula stating that these transitions are executed, how should it be interpreted ? There are two possibilities, following the numbering on the left and the right side of the slash sign, respectively. Thus, only having literals of the form $ex(t_1, \mathfrak{t})$ is not sufficient, but the order has to be taken into account. (This is relevant when synchronizing with the other components of the system.) It seems that a literal encoding the fact that transition t_1 is executed as the nth transition on the path is needed. This renders the encoding so complex (a new literal for every transition in every possible position on the path) that the possible gains in the bound are again outweighed.

5.2.2 Limitations on the Repetition of Same Action

The simple executions can not contain repeated actions. Thus, for instance the execution of all the transitions from the component in Figure 5.9 is not

possible.

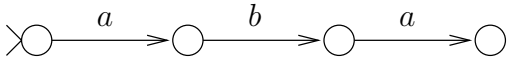

Figure 5.9: Repetition of Actions

The argument to rule out these kind of executions is similar to that of ruling out executions where a particular transition is executed several times. Firstly, it should be noted that if executions repeating visible actions were allowed the consistency check could not be implemented in the same way as presented in Section 5.1.1. This due to the fact that the executed transitions from a single component would induce a circular ordering graph and thus such valuation is not a model of the formula.

Therefore for instance, the second time an action appears has to be differentiated from the first time. But then, two literals are needed for any action. Again, if an action appears n times, n literals are needed. Thus, the formula resulting from the presented encoding is probably still easier to solve even though a few additional time steps might be needed.

5.3 DETERMINIZED LIMITED PATH PRODUCT

This section considers the problems encountered when trying to reduce the bounds needed for a counterexample by applying on-the-fly determinization to the domain of the limited path product. The definition of a determinized limited path product is easy and is as follows:

Definition 55 *Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ where each $\mathcal{L}_i = \langle S_i, I_i, \Sigma_i, \Delta_i \rangle$, $1 \leq i \leq n$ be LTSs. Let $\mathcal{L}_{st}^d = (S_{st}, I_{st}, \Sigma_i, \Delta_{st})$ be the determinized step product $\mathcal{L}_1 \|_{st}^d \cdots \|_{st}^d \mathcal{L}_n$. The* determinized limited path product *of $\mathcal{L}_1, \ldots, \mathcal{L}_n$, denoted $\mathcal{L}_1 \|_{pt}^{ld} \mathcal{L}_2 \|_{pt}^{ld} \cdots \|_{pt}^{ld} \mathcal{L}_n$, is the LTS $\langle S, I, \Sigma, \Delta \rangle$ such that:*

- $S = 2^{S_1} \times 2^{S_2} \times \cdots \times 2^{S_n}$,
- $I = \tau(I_1) \times \tau(I_2) \times \cdots \times \tau(I_n)$,
- $\Sigma = \{r_{\langle T_1, \ldots, T_n \rangle, \langle T_1', \ldots, T_n' \rangle} | \ \langle T_1, \ldots, T_n \rangle \in S, \langle T_1', \ldots, T_n' \rangle \in S\}$,
- $\Delta = \{\langle T_1, \ldots, T_n \rangle, r_{\langle T_1, \ldots, T_n \rangle, \langle T_1', \ldots, T_n' \rangle}, \langle T_1', \ldots, T_n' \rangle |$ *there is a simple execution from* $\langle T_1, \ldots, T_n \rangle$ *to* $\langle T_1', \ldots, T_n' \rangle$ *in* $\mathcal{L}_{st}^d\}$.

The problem in devising an encoding to the product above lies in the fact that from a reached set of state, the same actions in the same order should be executed.

Enforcing this seems to have the following difficulties. To maintain the correct order to rule out inconsistency in the component level requires a local version of the consistency check. This introduces additional complexity, but is probably possible. However, the consistency check can not be the same as presented above to detect global inconsistency. The problem is illustrated in Figure 5.10. Namely, if the singleton set $\{s_0\}$ is reached and the sequence

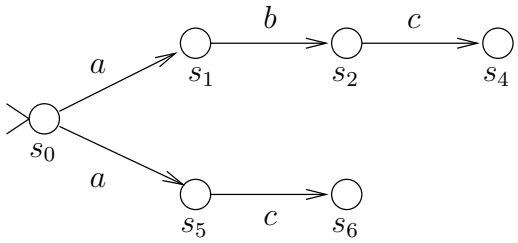

Figure 5.10: On-the-fly Determinization

abc is executed, the reached set should be $\{s_4\}$. However, both versions of the consistency checks presented above in Sections 5.1.1 and 5.1.2 would allow the reached set to be $\{s_4, s_6\}$. This due to the fact that the sequence *ac* is not inconsistent with *abc*, it merely omits an action. What is lacking is a local synchronization requirement that enforces the local ordering graph to be a chain. However, a harder problem is to enforce that every path from the reached set of states having the same visible labels in the same order are executed.

The limited path product model seems to provide a reasonable compromise in that in some test cases it is possible to reduce the needed bound significantly. Still, a reasonable encoding is possible and in some test cases, especially if performed iteratively, that encoding compares favorably to the encodings of other execution models.

6 IMPLEMENTATION

The ideas presented have been implemented to a toolset that reads system models specified as labeled transition systems and produces a Boolean formula in CNF form whose models correspond to the executions of the chosen execution model (interleaving, step, process, and limited path executions).

6.1 OPTIMIZATIONS BASED ON STATIC ANALYSIS

The presented execution models allow certain optimizations based on the structure of the system. The idea is to compute for each state and transition a number that tells the earliest possible moment ℓ that a particular state can be reached or a particular transition executed, respectively. If this is done in a sound manner, nothing is lost if the literals $in(s_j, \mathfrak{t})$ and $ex(t_k, \mathfrak{t})$ are replaced with $\bot$ when the value of $\mathfrak{t}$ is less than the computed value ℓ for the state or transition. The purpose of this is to simplify the resulting BMC formula.

For the interleaving, step, and process execution models without on-the-fly determinization, the optimization is based on first computing locally the smallest distance (the number of transitions) from an initial state to every other state. The earliest execution moment for a local transition, if the component is considered in isolation, is then the value obtained for its source state.

Then, based on the structure of all the components to be composed, it is checked whether some transitions can still be postponed. The reasoning in this latter step is that if for action a, the earliest moment it can be executed is ℓ in some component $\mathcal{L}_i$, then other components can not execute it earlier either. Even if they could execute a in isolation in, say $\ell - 2$, they can not proceed due to having to synchronize with $\mathcal{L}_i$. Therefore, the earliest execution moments in components other than $\mathcal{L}_i$ are updated to ℓ. Thereafter, the process is iterated by computing the local values again based on the component structure and the obtained new synchronization information. The process is repeated until no value is changed.

Let $\mathcal{L}_1, \ldots, \mathcal{L}_n$ be LTSs and the earliest execution moment for every local state s_i and transition t_j be denoted $er(s_i)$ and $er(t_j)$, respectively. Assume further that the BMC process is initiated with bound k. The complete algorithm that computes the earliest possible execution moment for every local state and transition is as follows:

1. For all $\mathcal{L}_i$, $1 \leq i \leq n$, let for all $s_j \in S_i$, n_{s_j} be the length (the number of transitions) of the shortest path to s_j from an initial state. If $n_{s_j} < k$, then $er(s_j) = n_{s_j} + 1$, otherwise $er(s_j) = k + 1$. For every local transition $t_l \in \Delta_i$, let $er(t_l) = er(pr(t_l))$.

2. For all visible actions a and all components $\mathcal{L}_i$ containing transitions labeled a, let $T \subseteq \Delta_i$ be the set of transition labeled a from $\mathcal{L}_i$. Let $n_{a,\mathcal{L}_i}$ be the smallest value $er(t_j)$ such that $t_j \in T$. Let then n_a be the maximum of the values $n_{a,\mathcal{L}_i}$ and L the set of components with

$n_{a,\mathcal{L}_i} = n_a$. If $t_l = (s, a, s')$, $t_l \in \Delta_m$, $\mathcal{L}_m \notin L$ and $er(t_l) < n_a$, then $er(t_l) = n_a$.

3. If no value is changed in step 2, terminate.

4. For all $\mathcal{L}_i$, $1 \leq i \leq n$:

   (a) for all local states $s_j \in S_i \setminus I_i$, let n be the lowest value $er(t_l)$ of the incoming transitions t_l of s_j. If $n+1 > er(s_j)$, then $er(s_j) = n+1$. However, if $n > k$, then $er(s_j) = k+1$ and

   (b) for all local transitions $t_j \in \Delta_i$, if $er(t_j) < er(pr(t_j))$, then $er(t_j) = er(pr(t_j))$. If $er(t_j)$ is changed for some local transition repeat step 4.

5. Goto step 2.

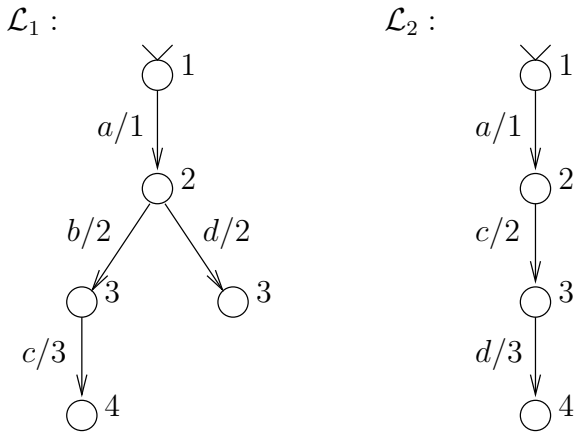

Figure 6.1: Optimization Algorithm (Beginning)

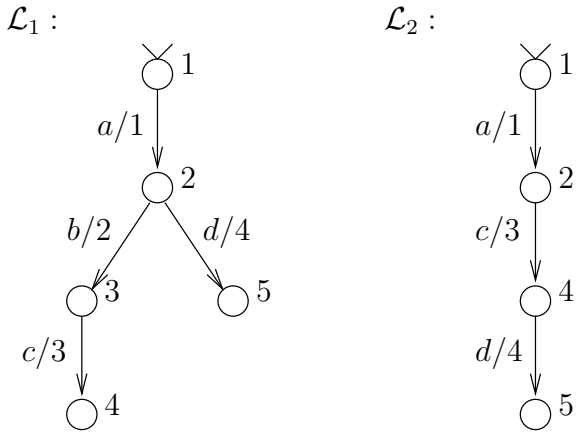

Figure 6.2: Optimization Algorithm (End)

An example is given in Figures 6.1 and 6.2 where the transitions are labeled for instance $c/3$ denoting the fact that the action of the transition is c

and that the algorithm computed that the earliest moment it can be executed is the third step. Similarly next to the local states, there is a number telling the earliest global state where that local state can be reached.

These numbers are obtained by first computing the minimum distances for each local state from an initial state. For each transition, the number is that of its source state. These numbers are given in Figure 6.1. In the second phase, it is noticed that the action c can occur only in the third step in component $\mathcal{L}_1$. Therefore the field $c/2$ in the component $\mathcal{L}_2$ is updated to $c/3$. However, this implies, due to the local structure of $\mathcal{L}_2$, that action d in component $\mathcal{L}_2$ can only take place in the fourth step. This information can then be propagated back to $\mathcal{L}_1$ after which the process is complete. The final result is shown in Figure 6.2.

It should be noted that limiting the values of the local states and transitions to the given bound is not done only due to computational efficiency. Namely, if this is not done, the algorithm may in some cases continue to increment the earliest values ad infinitum. The situation is illustrated in Figure 6.3. Since in the example system the transitions labeled a and b occur in conflicting orders, their synchronized, step, and process products deadlock in the initial state. The preprocessing algorithm computes first for the local transitions in component $\mathcal{L}_1$ the values $a/1$ and $b/2$ and for component $\mathcal{L}_2$ the values $b/1$ and $a/2$, respectively. Based on synchronization requirements, the earliest moment for execution becomes 2 for all the local transitions. Then, based on the local structure of $\mathcal{L}_1$, the values of its local transitions become $a/2$ and $b/3$, respectively. For $\mathcal{L}_2$, the values become $b/2$ and $a/3$ and the algorithm never stops. The limitation of the maximum value attached to a state or transition to the used bound is thus necessary.

With that limitation, the optimization algorithm is bound to terminate. This can be seen from the fact that it updates $\sum_{1 \leq i \leq n}(|S_i| + |\Delta_i|)$ counters whose value is a natural number that can only grow during the execution of the algorithm. Furthermore, the algorithm maintains the invariant that if the value $er(s_i)$ or $er(t_j)$ is k for some state s_i or transition t_j, state s_i can not be reached and transition t_j can not be executed in less than k steps, respectively. This can be seen by analyzing the steps of the algorithm. Firstly, at the outset the invariant holds. Secondly, every step in the algorithm maintains it. Finally, due to the polynomial (in the size of the system) number of updated counters, the running time of the algorithm is polynomial.

The optimization algorithm above works correctly for the standard interleaving, step, and process models. If on-the-fly determinization is used, the algorithm has to be modified slightly. The modification is such that τ transitions do not contribute when computing the values attached to states / transitions. This means that in step 1, when the initial values for states are computed based on the length of the shortest path to the state, if a τ transition is encountered, that transition is not added to the length of the path. Secondly, in phase 4a, when computing $er(s_i)$ of a local state s_i based on its incoming transitions t_j, the value $er(t_j)$ is incremented only if t_j is labeled with a visible action.

For the limited path execution model, the algorithm above does not provide correct results. However, a similar earliest value can be computed for every local state and transition. The reasoning behind the optimization al-

Figure 6.3: Example System

gorithm is based on the fact that simple executions used in the definition of the limited path product (Definition 45) do not allow the repetition of visible actions. Thus, if all the paths from the initial states to a particular local state contain a lot of repeated actions, then it takes several steps to reach that state. For instance, consider an LTS with the alphabet $\{a, b, c, d\}$. The execution of the path $s_0 \stackrel{aaabb}{\rightarrow} s_1$ takes 4 steps (the execution steps being a, a, ab and b) whereas the path $s_0 \stackrel{abcd}{\rightarrow} s_2$ can be executed in a single step. If these are the only paths to the state s_1 and s_2, respectively, the state s_1 can be reached only using an execution of length 4 whereas the state s_2 can be reached with an execution containing one step. In general, the computed number for a particular state is obtained from the path from an initial state that can be executed in the fewest number of steps.

However, in this case, the earliest step that a *transition* can be executed is not simply the number obtained for its source state. Namely, it depends on the label of the transition. Consider again an LTS with the alphabet $\{a, b, c, d\}$ and a path $s_0 \stackrel{aabb}{\rightarrow} s_1$ and assume that s_0 is its initial state and that the presented path is the only one to state s_1. It takes three steps to execute the path (the steps being a, ab, and b). Consider then the possible transitions leaving state s_1. Every transition whose label is not b can be combined to the last step b of the given path $aabb$. Thus, such transitions can be executed in the third step. For transitions labeled b, though, this is not the case since then the last step becomes bb, which is not allowed. Therefore, for any outgoing transitions from s_1 labeled b, their earliest execution step is four.

The local algorithm is again supplemented with a global analysis based on the synchronization requirements that may further postpone the execution of some actions. This phase is similar as in the algorithm for the interleaving, step, and process models. The complete algorithm is then again an iteration of a local and a global part until no value is changed.

The algorithm uses the following notation. Similarly as in the algorithm for interleaving, step, and process models, $er(t_i)$ and $er(s_j)$ denote the earliest possible execution moment for transition t_i and state s_j, respectively. In addition, it is assumed that the given bound is k. In this case a new counter, $er(s_j, a_l)$, is introduced. It denotes the earliest moment that a transition labeled a_l leaving state s_j can be executed. The algorithm is as follows:

1. For all $\mathcal{L}_i$, $1 \leq i \leq n$, let for all $s_j \in S_i$ and $a_l \in \Sigma_i$, n_{s_j,a_l} be the minimum number of transitions labeled a_l with which s_j can be

reached from some initial state. If $n_{s_j,a_l} < k$, then $er(s_j, a_l) = n_{s_j,a_l} + 1$, otherwise $er(s_j, a_l) = k + 1$.

2. For all $\mathcal{L}_i$, $1 \leq i \leq n$, let for all $s_j \in S_i$, n_{s_j} be the maximum of $er(s_j, a_l)$ for all $a_l \in \Sigma_i$. For all $a_m \in \Sigma_i$, if $er(s_j, a_m) < n_{s_j}$, then $er(s_j, a_m) = n_{s_j} - 1$. For all $t_n \in \Delta_i$, let $er(t_n) = er(pr(t_n), a)$ where a is the label of t_n.

3. For all visible actions a and all components $\mathcal{L}_i$ containing transitions labeled a, let $T \subseteq \Delta_i$ be the the set of transition labeled a from $\mathcal{L}_i$. Let $n_{a,\mathcal{L}_i}$ be the smallest value $er(t_j)$ such that $t_j \in T$. Let then n_a be the maximum of the values $n_{a,\mathcal{L}_i}$ and L the set of components with $n_{a,\mathcal{L}_i} = n_a$. If $t_l = (s, a, s')$, $t_l \in \Delta_m$, $\mathcal{L}_m \notin L$ and $er(t_l) < n_a$, then $er(t_l) = n_a$.

4. If no value changed in step 3, then let for all $\mathcal{L}_i$, $1 \leq i \leq n$ and all $s_j \in S_i$, $er(s_j)$ be the minimum of all $er(s_j, a_l)$, $a_l \in \Sigma_i$. Terminate.

5. For all $\mathcal{L}_i$, $1 \leq i \leq n$:

   (a) For all local states $s_j \in S_i \setminus I_i$ and all visible actions $a_l \in \Sigma_i$. Compute for s_j and a_l the value n_{a_l,t_m} over all the transitions $t_m = (s, a, s_j) \in pr(s_j)$ obtained as follows: If t_m is labeled with a_l, then $n_{a_l,t_m} = er(s, a_l) + 1$, otherwise $n_{a_l,t_m} = er(s, a_l)$. Let n_{a_l,s_j} be the minimum of these values. If $er(s_j, a_l) < n_{a_l,s_j}$, then if $n_{a_l,s_j} < k + 1$, $er(s_j, a_l) = n_{a_l,s_j}$, otherwise $er(s_j, a_l) = k + 1$.

   (b) For all transitions $t_j = (s, a, s') \in \Delta_i$. If $er(t_j) < er(s, a)$, then $er(t_j) = er(s, a)$.

   (c) If the value for some transition changed in step 5b, repeat step 5.

6. Goto step 2.

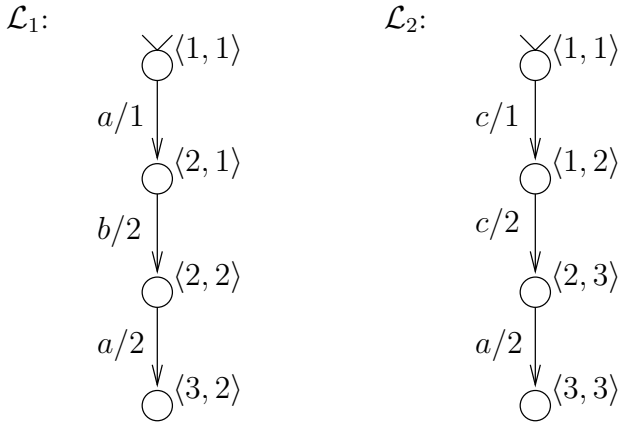

Figure 6.4: Optimization Algorithm after Step 2, Limited Path Model

An example is given in Figures 6.4 and 6.5. The values $er(s_j, a)$ are in each state given as a tuple of integers following the alphabetical order of actions. LTS $\mathcal{L}_1$ has the alphabet $\{a, b\}$ and $\mathcal{L}_2$ the alphabet $\{a, c\}$. Figure 6.4

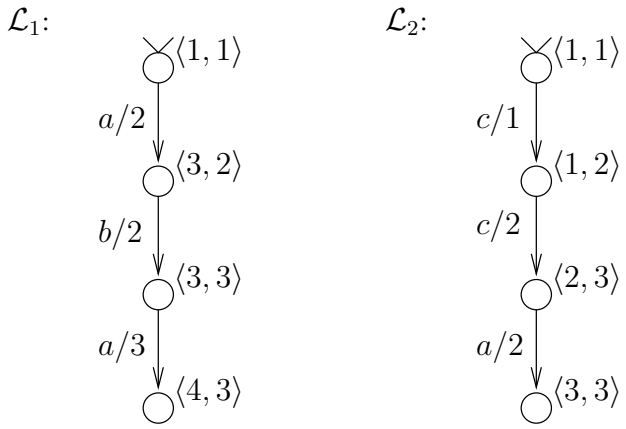

Figure 6.5: Completed Optimization Algorithm, Limited Path Model

shows the intermediate result after step 2. Then it is noticed that a can be postponed in $\mathcal{L}_1$ due to the structure of $\mathcal{L}_2$ to time step 2. Since it has to occur before b and the second a due to the structure of $\mathcal{L}_1$ those values are adjusted accordingly. The final result is shown in Figure 6.5.

The optimization algorithm for the limited path product model is also bound to terminate. This time it updates $\sum_{1\leq i\leq n}(|S_i|\cdot|\Sigma_i|+|\Delta_i|)$ counters whose value is a natural number that can only grow during the execution of the algorithm. However, the value of each counter is bound by the number $k+1$ where k is the given bound. Similarly as in the case for the algorithm for interleaving, step, and process models it can be shown that the algorithm maintains the invariant that the $er()$-values are conservative approximations of the earliest moment a state can be reached or a transition executed. Finally, the number of updated counters is polynomial and the running time of the algorithm thus polynomial in the size of the system.

6.2 TRANSLATION TO A BOOLEAN FORMULA

A toolset has been implemented that, given a set of components to be composed, the desired execution model, and the bound k on the length of the executions, yields a Boolean formula in DIMACS form. The models of this formula are the executions of length k of that product of the components that corresponds to the given execution model. Besides the algorithm based on static analysis, the translation is optimized in the following way. Firstly, the resulting formula is represented as a Boolean circuit (elaborated below). Secondly, the circuit is reduced by simplification rules, substructure sharing and cone of influence reduction [48]. The reduced circuit is transformed to CNF using a linear size translation introducing a new atom for each gate in the circuit. The circuit reduction and mapping are done using a tool called `BCzChaff` [46] which is integrated with the `zChaff` SAT solver [70] so that `zChaff` can be run directly on the generated CNF formula.

The Boolean circuit representation of a BMC formula is as follows: for each atom and connective in the formula, a gate is introduced in the circuit. This is easy as the tool used supports directly the extended set of Boolean

functions used in the translation, including the cardinality constraints. The left hand side of Figure 6.6 presents, for instance, a possible instantiation of constraint (4.4) from the interleaving execution model. To obtain a sound encoding, this gate has to be constrained to the value of true.

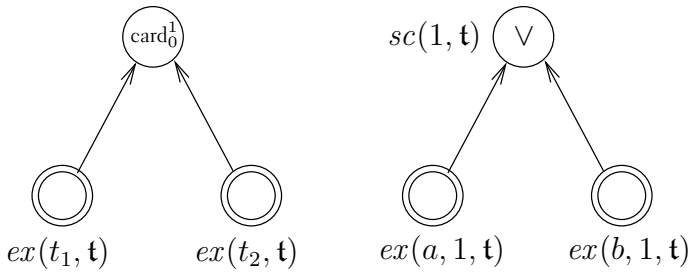

Figure 6.6: Example Circuits

However, the representation is optimized when translating constraints that are equivalences, like for instance constraint (4.11). For such constraints, no gate for the atom on the left nor for the equivalence connective are introduced. Rather, the atom on the left is identified with the gate for the main connective on the right hand side of the equivalence. For example, the right hand side of Figure 6.6 presents the subcircuit for the literal $sc(1, \mathfrak{t})$ (instance of constraint (4.11)) defined as being equivalent to the disjunction of the literals $ex(a, 1, \mathfrak{t})$ and $ex(b, 1, \mathfrak{t})$.

The complete process of translating an LTS system to a Boolean circuit encoding the presented execution models results in a circuit whose schematic diagram can be seen in Figure 6.7. It presents the transitions relation unrolled three steps. The circuit is a faithful representation of the encoding as a propositional formula because the satisfying truth valuations of the BMC formula and the satisfying valuations of the circuit coincide. A satisfying valuation of the circuit is a truth value assignment for the input gates (gates with no incoming edges) of the circuit such that the resulting value of each constrained gate matches it's specified value (true).

The circuit in Figure 6.7 corresponds to the execution models not applying on-the-fly determinization. This can be seen from the fact that the input gates of the circuits are labeled $ex(t_i, \mathfrak{t})$ for the time steps $\{0, 1, 2\}$. If a circuit is created for the interleaving, step, or process models applying on-the-fly determinization, the input gates are the $ex(a, \mathfrak{t})$ literals where a varies over the visible actions of the system.

In order to limit the execution to those violating a reachability property, additional gates encoding the property are added after the last step in the circuit.

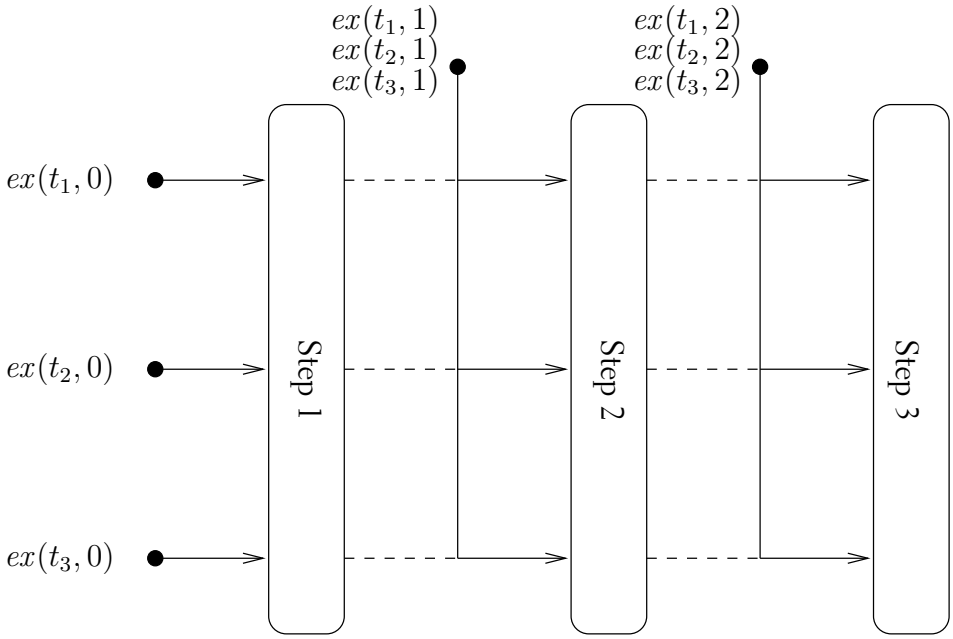

Figure 6.7: Schematic Diagram

7 TEST CASES

This dissertation presents different techniques to create composition operators for labeled transition systems with the goal of creating efficient BMC algorithms. This chapter presents some experimental results. Firstly, the different composition operators are compared to each other and, secondly, to existing state-of-the-art model checking tools.

Whereas direct comparison of the different presented BMC methodologies is feasible, the task of comparing to other tools is not trivial, since there are several sources of bias. Firstly, these tools may have been written to different types of systems and their input syntax vary. Secondly, to verify that the models for the tools indeed represent the system to be verified in the same way (for instance, contain the same number of states) is hard. In [26] Corbett performs comparative study of different deadlock detection techniques. He compares the following systems that have different approaches to the state explosion problem.

- SPIN, an explicit state model checker applying partial order reduction [42],
- SMV, a symbolic model checker [63], and
- INCA, a tool based on inequality necessary conditions [27].

The choice of tools is justified by their different way of alleviating the state-explosion problem. In the paper [26], the input files to the SPIN, SMV, and INCA tools are created from models where the systems are described as synchronizing labeled transitions systems (in the paper called finite state automata). Corbett analyzes the difficulties caused by the different input syntaxes and tries to ensure that given a system description as LTSs, the resulting SPIN, SMV, and INCA descriptions model the system in the same way.

Since the system models are available in different input languages, some of the presented tests are used here to compare the established tools against the step, process, and transition merging execution models. Since some of the examples are scalable, they have been scaled up to take into account the added computer power since the paper's publication. Here, the comparison is limited to the state-of-the-art model checking tools SPIN, version 4.2.4 [42] and NuSMV, version 2.2.3 [19]. [1]

The presented examples are as follows (entry of the type **Name(n)** indicates scalability, where **n** is the parameter):

Dac(n): A program modeling a divide and conquer computation by forking up to n solver tasks that proceed in parallel. When $n = 200$, this system contains ca. $1.2 \cdot 10^{52}$ reachable states.

Dartes: The communication skeleton of a fairly complex Ada program with 32 tasks. This system contains ca. 10^{13} reachable states.

DP(n): The dining philosopher's problem. When $n = 12$, this system contains 531440 reachable states.

[1] the latest versions available in March 2005

Elev(n): A model of a controller for a building with n elevators. When $n = 4$, this system contains 47436 reachable states.

Hartstone(n): The communication skeleton of an Ada program in which one task starts and then stops n worker tasks. This system contains $n + 2$ reachable states.

Key(n): The communication skeleton of an Ada program that manages the keyboard / screen interaction in a window manager. When $n = 4$, this system contains 44819 reachable states. When $n = 5$, the system contains 398760 reachable states.

Mmgt(n): The communication skeleton of an Ada program implementing a memory management scheme with n users. When $n = 4$, this system contains 66308 reachable states.

Q(n): The Ada skeleton of an RPC client / server-based user interface. When $n = 1$, this system contains 123596 reachable states.

Sentest(n): The communication skeleton of an Ada program that starts n tasks to test sensors. When $n = 400$, this system contains 1962 reachable states.

Speed: The communication skeleton of an Ada program to regulate the speed of a car. This system contains 8689 reachable states.

In order to test the special characteristics of the execution model applying transition merging, the above set is extended with the following case:

Tree(n): The example presented in Section 5.1.3. It is trivial for the step and process models but potentially hard for the transition merging model due to the fact that it contains a lot of inconsistent executions. This system contains $n + 1$ reachable states.

It is known that all these examples have a deadlock. Furthermore, for all examples the bound with which the deadlock is reached is known from earlier work with these models for all the presented composition operators. It is an unfortunate fact that most of the examples have a small reachable state space for modern computers. However, the author is not aware of any other deadlock checking benchmark collection for LTSs.

When reporting results, the following conventions are applied. The running time for a particular instance is obtained from `/usr/bin/time` and it is the sum of the user time and the system time. In all the test runs, a time limit of one hour is used. If the running time of a particular instance exceeds this limit, the result is reported in the form $> 1h$. All the tests are run using an AMD Athlon machine with a 1400 MHz CPU with one gigabyte of memory running the Debian GNU/Linux operating system.

The first results are presented in Table 7.1. The table compares the bound and the running time of interleaving and parallel order (step and process) semantics. In this case, no on-the-fly determinization nor any static analysis is applied. The columns in the table are as follows:

- Problem, the name of the test case,

Table 7.1: Test Results of Interleaving, Step and Process Models

Problem	Il. k	Il. s	St. k	St. t	Pr. k	Pr. s
Dac(200)	3	**0.51**	3	0.98	3	0.87
Dartes	32	**2.7**	32	3.4	32	4.0
Dp(12)	12	$> 1h$	1	**0.02**	1	**0.02**
Elev(4)	17	268	10	4.0	10	**3.1**
Hart.(100)	201	97.7	201	**81.2**	201	147
Key(4)	50	962	37	**9.9**	37	264
Key(5)	52	1610	38	**6.2**	38	348
Mmgt(4)	12	6.4	8	**0.43**	8	0.70
Q(1)	21	2.1	9	**0.36**	9	0.42
Sentest(400)	413	337	408	**199**	408	471
Speed(1)	7	0.03	4	0.03	4	**0.01**
Tree(100)	100	**18.5**	100	19.8	100	23.2

- Il. k, the bound using interleaving executions,
- Il. s, the running time in seconds using interleaving executions,
- St. k, the bound using step executions,
- St. s, the running time in seconds using step executions,
- Pr. k, the bound using process executions, and
- Pr. s, the running time in seconds using process executions.

The reported time is the time it takes to solve the *single* SAT formula corresponding to the reported bound using the solver `BCzChaff` that uses `zChaff` version 2004.11.15 as a back end. The reported bound is the smallest number of steps required to reach a deadlock. For all examples, the best running time is highlighted.

From Table 7.1 it can be seen that in many cases, the bound needed to detect a deadlock is smaller using a composition operator applying partial order semantics than using interleaving semantics. In one example, **Dp(12)**, the bound reduces from 12 to one when interleaving semantics is replaced by step semantics. In this example, also the running time reduces from over one hour to a fraction of a second. It seems to be the case that in many examples the reduced bound of partial order semantics yields a shorter running time of the solver. This analysis can also be supported mathematically by computing the arithmetic average a of the ratio (running time step semantics / running time interleaving semantics) for the used examples. Then, the number $1 - a$ is used as an indication of the average speedup of using step semantics. The average speedup in the running time using step semantics compared to interleaving semantics is 32% (excluding **Dp(12)** since it did not terminate within an hour using interleaving semantics). However, an assessment of the superiority of step semantics is not conclusive for at least two reasons. Firstly,

the set of test cases is limited and secondly, using another SAT solver, the running times could be different.

When step and process semantics are compared, it can firstly be (trivially) seen that the bound needed to detect a deadlock is the same. A second observation is a bit surprising. In most cases, the running time with step semantics is smaller than with process semantics. In average, the speedup of using step semantics instead of processes is 13 %. Compared to the formula modeling step semantics, the formula modeling process semantics contains one additional constraint. This constraint reduces the number of models but does not seem to provide a performance gain, at least not with `BCzChaff`. This fact is interesting since it is in direct conflict with the conclusions of [39]. It would be interesting to know whether there is something in the test setup used in [39] that renders process semantics more competitive than step semantics.

Table 7.2 presents the same examples with the same execution models, however, this time applying the static analysis algorithm presented in Section 6.1. Even though not comprehensively reported, the size of the Boolean circuit is typically significantly reduced when this algorithm is applied. For instance, for the example **Dartes**, the size of the circuit for interleaving semantics reduces from roughly 6.7 megabytes to 4.0 megabytes when static analysis is applied. This seems in most cases to be reflected also in the running times. Indeed, comparing the circuits encoding step semantics from Table 7.2 to those from Table 7.1, in average the running times of the former are 28% shorter. Also with static analysis (within Table 7.2), step semantics seems to outperform interleaving and process semantics.

Table 7.2: Test Results of Interleaving, Step and Process Models (Static Analysis)

Problem	Il. k	Il. s	St. k	St. t	Pr. k	Pr. s
Dac(200)	3	0.24	3	**0.17**	3	0.29
Dartes	32	2.9	32	**1.8**	32	2.2
Dp(12)	12	$> 1h$	1	0.05	1	**0.01**
Elev(4)	17	119	10	1.9	10	**1.7**
Hart.(100)	201	88.2	201	**75**	201	131
Key(4)	50	2151	37	**3.1**	37	26.3
Key(5)	52	1168	38	**5.6**	38	431
Mmgt(4)	12	4.6	8	**0.42**	8	1.4
Q(1)	21	1.5	9	0.22	9	**0.17**
Sentest(400)	413	**197**	408	324	408	341
Speed(1)	7	0.03	4	**0.00**	4	0.03
Tree(100)	100	**8.6**	100	12.6	100	13.5

Table 7.3 presents the same examples, this time with interleaving, step and process models applying on-the-fly determinization. Static analysis is also applied. Compared to tables 7.1 and 7.2, the needed bounds in 7.3 to detect a deadlock are never greater, in many examples they are lower. This

follows from the fact that the internal transitions of the components do not contribute to the length of the execution. For interleaving semantics, the difference is in general larger than for the execution models applying partial order semantics. For instance, for the example **Elev(4)**, the bound reduces from 17 to 10. In this case, the running time of BCzChaff is also significantly smaller. In Table 7.3, step and process semantics give comparable results, the average speedup of step semantics (excluding the examples **Dp(12)** and **Speed(1)** with trivial running times) is only 8%.

When step semantics with on-the-fly determinization (from Table 7.3) is compared to step semantics with static analysis (from Table 7.2), the running times using the former are 169% worse in average. An important contributing factor to this is the very good running times of the examples **Key(4)** and **Key(5)** in Table 7.2. If these two examples are excluded from the comparison, the average speedup of on-the-fly determinization using step semantics is 20%.

Table 7.3: Test Results of Interleaving, Step and Process Models (Static Analysis and On-the-fly Determinization)

Problem	Il. k	Il. s	St. k	St. t	Pr. k	Pr. s
Dac(200)	2	0.14	2	0.1	2	**0.07**
Dartes	31	2.9	31	**1.3**	31	2.9
Dp(12)	12	$> 1h$	1	**0.00**	1	**0.00**
Elev(4)	10	2.6	9	1.6	9	**1.2**
Hart.(100)	200	84.3	200	**74.4**	200	79
Key(4)	47	98.4	37	51.7	37	**36.3**
Key(5)	49	450	38	**21.0**	38	36
Mmgt(4)	8	1.7	8	**0.58**	8	0.60
Q(1)	19	0.88	9	0.18	9	**0.16**
Sentest(400)	412	786	408	**56.4**	408	236
Speed(1)	7	0.04	4	0.01	4	**0.00**
Tree(100)	100	**7.4**	100	10.5	100	14.7

One source of bias to the presented experimental data is the fact that so far, only a single solver, BCzChaff, is applied. Therefore, some of the harder examples for BCzChaff were rerun using another solver, siege_v4 [80]. The examples are **Dp(12)**, **Elev(4)**, **Key(4)**, and **Key(5)**. The reason for choosing these examples is that at least with some execution semantics, they have running times exceeding 100 seconds. For **Dp(12)**, this is the case only for interleaving semantics. However, since that running time exceeds one hour, comparison to siege_v4 is interesting. Secondly, the chosen examples contain relatively complex component LTSs. Simple structure is the reason why the examples **Sentest(400)**, **Hartstone(100)** and **Tree(100)** are left out, even though their running time in some cases exceeds 100 seconds.

The results are presented in Tables 7.4, 7.5, and 7.6. The columns of the tables are as follows:

- Il.(B), the running time in seconds of BCzChaff with interleaving se-

mantics,

- Il.(s), the running time in seconds of siege_v4 with interleaving semantics,
- St.(B), the running time in seconds of BCzChaff with step semantics,
- St.(s), the running time in seconds of siege_v4 with step semantics,
- Pr.(B), the running time in seconds of BCzChaff with process semantics, and
- Pr.(s), the running time in seconds of siege_v4 with process semantics.

The presentation follows the conventions applied when presenting Tables 7.1, 7.2 and 7.3. In Table 7.4 the running times are presented when no static analysis nor on-the-fly determinization is applied. In Table 7.5 static analysis is added and finally in Table 7.6 both static analysis and on-the-fly determinization are applied. However, siege_v4 is based on a randomized algorithm and the running times for a particular instance may vary depending on the seed value given to the random number generator. To remove this potential source of bias, the running times for siege_v4 are the arithmetic average for 5 runs using different seed values. In all tables, the running time of the better solver is highlighted.

The presented results justify the fact that if the tests cases are run using only a single SAT solver, a source of bias is introduced. Namely, there are many examples where siege_v4 performs better than BCzChaff, for instance in all the test cases with interleaving semantics. For instance in the example **Dp(12)**, when interleaving semantics and static analysis are applied, the running time of over one hour of BCzChaff reduces to 51.6 seconds. However, there are also examples where BCzChaff performs better that siege_v4. It should be noted that also with siege_v4, the running times using step semantics compare favorably to those using interleaving and process semantics.

Table 7.4: Comparison between BCzChaff and siege_v4

Problem	Il. (B)	Il. (s)	St. (B)	St. (s)	Pr. (B)	Pr. (s)
Dp(12)	$> 1h$	**51.6**	0.02	**0.00**	0.02	**0.01**
Elev(4)	268	**15.0**	**4.0**	5.1	**3.1**	4.3
Key(4)	962	**108**	**9.9**	13.9	264	**80.0**
Key(5)	1610	**491**	**6.2**	50	348	**122**

In Table 7.7 results are presented when local transition merging is applied. The results are presented for both the implementation with the full (cubic) consistency check and the implementation applying iterative strengthening. In addition, the table contains again the results for step semantics from Table 7.1. These are given to be able to easier compare the cubic and iterative encodings to the fastest applicable (not applying static analysis and on-the-fly determinization) linear encoding. The columns of the table are as follows:

Table 7.5: Comparison between BCzChaff and siege_v4 (Static Analysis)

Problem	Il. (B)	Il. (s)	St. (B)	St. (s)	Pr. (B)	Pr. (s)
Dp(12)	$> 1h$	**385**	0.05	**0.01**	0.01	0.01
Elev(4)	119	**19.9**	1.9	**0.62**	1.7	**1.0**
Key(4)	2151	**168**	**3.1**	23.8	**26.3**	99.9
Key(5)	1169	**187**	**5.6**	26.5	341	**157**

Table 7.6: Comparison between BCzChaff and siege_v4 (Static Analysis and On-the-Fly Determinization)

Problem	Il. (B)	Il. (s)	St. (B)	St. (s)	Pr. (B)	Pr. (s)
Dp(12)	$> 1h$	**1636**	**0.00**	0.01	**0.00**	0.01
Elev(4)	2.6	**0.93**	1.6	**0.69**	1.22	**0.96**
Key(4)	98	**45.2**	52	**36.9**	**36.3**	36.4
Key(5)	451	**43.3**	**21.0**	52.6	36.5	**20.2**

- Pt. k, the bound with local transition merging,
- Pt. s, the running time in seconds of BCzChaff with local transition merging,
- Pt.(ite) s, the running time in seconds when iterative strengthening is applied,
- ite, the number of iterations needed to detect a rule out spurious counterexamples,
- St. k, the bound using step semantics from Table 7.1, and
- St. s, the running time in seconds using step semantics from Table 7.1.

Compared to interleaving and partial order execution models, the bounds in Table 7.7 are dramatically smaller. Indeed, there are many cases where a single step suffices to detect a deadlock. This suggests that the benchmarks contain many deadlocks of a simple nature. However, when the complete consistency check is encoded, the reduction in the size of the formula due to the smaller bound is quickly offset by the complexity of the consistency check.

Compared to the already presented execution models, the performance of local transition merging varies depending on the example. There are examples, like **Hartstone(100)** and **Sentest(100)** where the significant reductions in the bound compared to step semantics result in lower running times, even with the full consistency check. However, in the case of the example **Sentest(n)**, the alphabet of the example obtained with parameter value 400 is prohibitely large. Therefore, the value is reduced to 100 for the transition merging model (the running time for step semantics is obviously also for **Sentest(100)** in Table 7.7).

Table 7.7: Test Results of Local Transition Merging Model

Problem	Pt. k	Pt. s	Pt.(ite) s	ite	St. k	St. s
Dac(200)	1	514	**1.2**	0	3	0.98
Dartes	1	57.0	**0.41**	0	32	3.4
Dp(12)	1	0.73	0.06	0	1	**0.02**
Elev(4)	1	60.2	530	22	10	**4.0**
Hart.(100)	1	0.36	**0.25**	0	201	81.2
Key(4)	16	868	3600	0	37	**9.9**
Key(5)	16	2231	$> 1h$	N/A	38	**6.2**
Mmgt(4)	1	0.56	**0.21**	0	8	0.43
Q(1)	1	3.2	0.67	3	9	**0.36**
Sentest(100)	6	1.5	13.8	0	108	7.7
Speed(1)	1	0.05	**0.02**	0	4	0.03
Tree(100)	1	107	54.1	98	100	**19.8**

Table 7.7 also presents the experimental results obtained from the implementation where the cubic complex constraint to detect inconsistent executions is initially left out. This introduces spurious counterexamples and rightmost column presents the number of refinement cycles needed to obtain a correct counterexample. The results indicate that when zero or only a few iterative cycles are needed, the implementation using iterative strengthening is superior to that encoding the full consistency check. However, for instance with the example **Elev(4)**, so many iterations are needed that the encoding with the full consistency check (not to mention step semantics) performs better. Finally, it should be noted that the example **Key(4)** is actually solved exactly within the time limit.

Table 7.8: Test Results of Local Transition Merging Model (Static Analysis)

Problem	Pt. k	Pt. s	Pt.(ite) s	ite	St. k	St. s
Dac(200)	1	486	1.1	0	3	**0.17**
Dartes	1	59.9	**0.41**	0	32	1.8
Dp(12)	1	0.74	0.07	0	1	**0.05**
Elev(4)	1	8.1	**0.36**	0	10	1.9
Hart.(100)	1	17.7	**0.36**	0	201	75.1
Key(4)	16	896	3600	0	37	**3.1**
Key(5)	16	2630	3600	0	38	**5.6**
Mmgt(4)	1	0.20	**0.04**	0	8	0.42
Q(1)	1	2.6	0.43	3	9	**0.22**
Sentest(100)	6	2.6	**1.7**	1	108	5.6
Speed(1)	1	0.02	0.01	0	4	**0.00**
Tree(100)	1	109	50.3	98	100	**12.6**

Table 7.8 presents the results of the same execution model as in Table 7.7,

however, this time with the static analysis algorithm presented in Section 6.1 activated. These are compared against the step execution model from Table 7.2 where static analysis is also applied. Comparison of tables 7.7 and 7.8 show that when full consistency check is encoded, applying static analysis does not seem to affect the performance.

When the encoding is performed iteratively, comparison of Tables 7.7 and 7.8 show that the number of iterations needed is in some cases reduced. For instance, in the example **Elev(4)**, the 22 refinement cycles of Table 7.7 are reduced to zero in Table 7.8. This phenomenon can be explained as follows. An analysis of the spurious counterexamples of the test cases presented in Table 7.7 shows that they are spurious because unreachable transition cycles are executed. However, these transition cycles are "deep" in the components and thus the impossibility of the transitions being executed is detected by the static analysis algorithm.

The experimental part concludes with a comparison to the state-of-the-art model checking tools NuSMV and SPIN. NuSMV contains both a BDD based symbolic model checking back end and a SAT based BMC back end. The comparison is performed against both of them. From the BMC encodings the most efficient (arguably) non-iterative semantics is chosen for comparison, namely step semantics with static analysis and on-the-fly determinization. In addition, NuSMV and SPIN are compared against the iterative local transition merging model. Since both BDD based NuSMV and SPIN are complete model checking techniques it is not fair to a compare them against a BMC run of a single propositional formula corresponding to the smallest depth where a deadlock is found. Therefore, in this case, the running time for the BMC techniques is the sum it takes to solve successive propositional instances corresponding to increasing depth starting from zero.

The general convention of applying a time limit of one hour is still applied. In some cases, this time limit is not sufficient to solve all the propositional instances until the depth of the deadlock is reached. If this is the case, the bound is reported in the form $> k$ where k is the depth corresponding to the last instance whose unsatisfiability could be established within the time limit. Compared to the presented BMC implementations, NuSMV and SPIN seem to require large amounts of memory. In these comparisons the maximum amount of virtual memory and maximum resident set size are limited to 900 megabytes. With this limitation, SPIN reports running out of memory with two examples, **Dartes** and **Dp(12)**. With these examples, the running time is marked with a dagger (†).

```
set input_file file.dlcheck.smv;
set enable_reorder 1;
go;
compute_reachable;
check_fsm;
quit;
```

Figure 7.1: NuSMV BDD command file

When NuSMV's BDD based model checking is used, NuSMV is started

in interactive mode and the commands given in Figure 7.1 are executed. The first command sets the input file containing the SMV specification. The second allows the dynamic reordering of BDD variables. This can potentially reduce the size of the BDDs. The command "`go;`" initializes the system for verification. Then, the reachable states are computed and finally, the command "`check_fsm;`" checks the transition relation for totality, i.e., whether there is a reachable deadlock.

```
set input_file file.dlcheck.smv;
set sat_solver zchaff;
go_bmc;
check_ltlspec_bmc -p "G (Live)" -k 100 -l X;
quit;
```

Figure 7.2: NuSMV BMC command file

When NuSMV's BMC is applied, NuSMV is started in interactive mode and the commands given in Figure 7.2 are executed. Again, the first command gives the input file containing the SMV specification created using the encodings from Corbett. The second sets the used SAT solver, in this case `zChaff`. The command "`go_bmc;`" initializes the system for BMC. Finally, the command "`check_ltlspec_bmc`" verifies whether the given LTL specification holds for all execution up to the given bound. The specification is "`G (Live)`" where `Live` is a predicate that evaluates to true iff some action can be executed in the reached state. The bound follows after the switch "`-k`" and is in Figure 7.2 set to 100. The argument "`-l X`" limits the search of counterexamples to non-looping executions.

The results of the comparison of step semantics with on-the-fly determinization against NuSMV's BDD based and BMC implementations are presented in Table 7.9. The columns of the table are as follows:

- Problem, the name of the test case,
- St. k, the depth reached using step semantics with static analysis and on-the-fly determinization,
- Σ St. s, the running time in seconds using step semantics with static analysis and on-the-fly determinization,
- SMV s, the running time in seconds of NuSMV BDD,
- SMV(B) k, the depth reached using NuSMV BMC, and
- Σ SMV(B) s, the running time in seconds of NuSMV BMC.

The results in Table 7.9 indicate that many of the examples are so small that they can be easily solved using NuSMV with BDDs. However, notable exceptions to this trend are the examples **Dac(200)**, **Dartes** and **Sentest(400)**. When step semantics with on-the-fly determinization is compared against the interleaving BMC implementation of NuSMV, it compares favorably. Obviously, the translation from the LTS specifications to the input

Table 7.9: Test Results of Comparison Against NuSMV

Problem	St. k	Σ St. s	SMV s	SMV(B) k	Σ SMV(B) s
Dac(200)	2	**0.15**	663	7	623
Dartes	31	**15.3**	272	> 28	1944
Dp(12)	1	**0.00**	0.21	> 8	549
Elev(4)	9	**2.6**	4.7	> 8	2063
Hart.(100)	> 184	3544	**6.2**	> 97	957
Key(4)	> 27	3153	**1.2**	> 21	3574
Key(5)	> 24	111	**3.4**	> 19	1758
Mmgt(4)	8	4.4	**0.32**	9	1660
Q(1)	9	**0.41**	4.9	> 13	1927
Sentest(400)	> 249	3597	$> 1h$	> 8	201
Speed(1)	4	**0.03**	0.13	7	0.23
Tree(100)	100	300	**6.5**	> 90	994

syntax of NuSMV adopted from Corbett [26] can introduce bias. Corbett translates the synchronizing LTSs to a NuSMV transition relation. This translation was devised before BMC was introduced. The translation used is quite straightforward and might be done more efficiently in many ways.

Table 7.10 compares NuSMV's BDD based model checking to the iterative implementation of the execution model with local transition merging. The results in the table indicate that with `BCzChaff`, iterative strengthening compares favorably to NuSMV with the examples where a single step suffices to detect a deadlock. An exception to this is the example **Tree(100)** that contains lot of inconsistent executions. With the examples **Key(4)** and **Key(5)**, however, NuSMV clearly outperforms even the iterative strengthening technique.

When the presented methods are compared to SPIN, SPIN is used as follows. Firstly, a verifier is created from the PROMELA specification using the command "`spin -a file.prom`". The resulting model checker is then compiled with the following options:

- `-DBFS` to use breadth first search. This is needed to detect the shortest deadlocks.
- `-DCOLLAPSE` to reduce memory usage by compressing state vectors, and
- `-DSAFETY` to optimize the performance for reachability properties.

Table 7.11 presents the results of the comparison to SPIN. The columns of the table are:

- Problem, the name of the test case,
- St. k, the depth reached using step semantics with static analysis and on-the-fly determinization,

Table 7.10: Test Results of Iterative Encoding against NuSMV BDD

Problem	Pt.(ite) k	Σ Pt.(ite) s	SMV s
Dac(200)	1	**1.0**	663
Dartes	1	**0.41**	272
Dp(12)	1	**0.07**	0.21
Elev(4)	1	**0.36**	4.7
Hart.(100)	1	**0.36**	6.2
Key(4)	> 11	2002	**1.2**
Key(5)	> 10	1922	**3.4**
Mmgt(4)	1	**0.04**	0.32
Q(1)	1	**0.43**	4.9
Sentest(100)	6	**29.3**	$> 1h$
Speed(1)	1	**0.02**	0.13
Tree(100)	1	109	**6.52**

- Σ St. s, the running time in seconds using step semantics with static analysis and on-the-fly determinization,
- Pt.(ite) k, the depth reached using local transition merging with static analysis,
- Σ Pt.(ite) s, the running time in seconds using local transition merging with static analysis,
- SPIN k, the depth of the shortest found deadlocking execution using SPIN, and
- SPIN s, the running time in seconds of SPIN.

Table 7.11: Test Results of Comparison Against SPIN

Problem	St. k	Σ St. s	Pt.(ite) k	Σ Pt.(ite) s	SPIN k	SPIN s
Dartes	31	15.3	1	**0.41**	-	†
Dp(12)	1	**0.00**	1	0.07	-	†
Elev(4)	9	2.6	1	0.4	34	**0.06**
Hart.(100)	> 184	3544	1	0.36	502	**0.06**
Key(4)	> 27	3153	> 11	2002	158	**5.9**
Key(5)	> 24	111	> 10	1922	183	**72.8**
Mmgt(4)	8	4.4	1	**0.04**	28	0.90
Q(1)	9	0.41	1	**0.43**	171	39.2
Speed(1)	4	0.03	1	**0.02**	30	0.19

The results presented in Table 7.11 reiterate the result that there are systems, where explicit-state model checking (even with partial order reduction) requires too much memory, whereas using BMC, it is possible to find errors.

This is the case with the examples **Dartes** and **Dp(12)**. However, when available memory permits, SPIN seems to be very efficient. In Table 7.11, the depths of the deadlocking executions are reported also for SPIN. These have been obtained using breadth first search (switch `-DBFS`) in order to detect the shortest deadlocks.

However, even though SPIN applies interleaving semantics, the reported bounds are not the same as for the interleaving semantics reported in Table 7.1. This difference can be traced to the way, the LTS (fsa) to PROMELA translation is done in [26]. The execution of a visible action in the LTS domain corresponds to a rendez-vous communication in the SPIN domain. Thus, every visible action introduces a single step to the LTS counterexample and two steps to the SPIN counterexample. In addition, when SPIN is used, process creation introduces additional steps to the counterexample, one for each created process.

Finally, for some of the examples that have been scaled up or created from scratch, no PROMELA model is available. These are the examples **Dac(200)**, **Sentest(400)** and **Tree(100)** that have therefore been left out from Table 7.11.

8 CONCLUSIONS

The research goal of this dissertation has been to develop efficient bounded model checking techniques for concurrent systems composed of labeled transition systems. The idea is to replace the standard interleaving semantics with non-standard execution models. The potential added efficiency of these new execution models comes from two sources. Firstly, the counterexamples are shortened and secondly, the search space of the SAT solver can potentially be reduced by limiting the number of executions of the system.

When standard interleaving semantics is applied, in each component exactly one action is executed in each time step. The counterexamples of this standard semantics are shortened using three techniques. Firstly, a partial order semantics, referred to as step semantics, is applied. Step semantics allows the simultaneous execution of independent actions. Thus, several actions can be executed in a single time step, however, at most one from a single component. The second technique to shorten counterexamples is by determinizing the components (LTSs) of the system. When this is done, the executions of the concurrent system no longer contain steps where transitions internal to a component are executed. The final technique to shorten counterexamples is to allow local transitions to be merged. This execution model lifts the limitation that at most one action can be executed in a single time step from each component.

The second potential source for added BMC efficiency is based on limiting the number of executions of the system. In this dissertation, this is done using two techniques. Firstly, the components of the system can be determinized. Besides shortening the counterexamples, this also reduces the number of executions. Secondly, when step semantics is applied, it is possible to disallow executions that are not in a certain normal form. The resulting partial order semantics is called process semantics.

To be applicable for efficient BMC, the initial states and the transition relation of the non-standard execution models should not be too difficult to encode compared to those of the interleaving semantics. In addition, the resulting formula should not be too large or too difficult for the used SAT solvers. It turns out that it is easy to encode the execution model using step semantics. Furthermore, limiting the executions to the studied normal form (process semantics) requires only a single additional constraint to the formula for step semantics. Similarly, it is easy to encode the execution model modeling the system where the original components are determinized. In addition, combining the two techniques above, i.e. partial order semantics with determinized components is also easy. All the BMC formulas from these execution models are linear in the size of the original system, as is the case with interleaving semantics.

The situation is not so easy when local transition merging is applied. Firstly, it is likely that an execution model allowing arbitrary local transition merging can not be encoded with a formula whose size is polynomial in the size of the system. Namely, if this were the case, the shortest counterexample demonstrating a violation of a reachability property would be of length one. Then, however, the **PSPACE**-complete problem of the reachability of

a global state would be solved in **NP**.

The dissertation demonstrates that a polynomial encoding applying limited local transition merging is possible. The limitations are such that (i) from each component at most one transition labeled with a given visible action can be executed and that (ii) each component may execute a loop at most once. However, even with these limitations, the encoding is no more linear. This is due a complex constraint, presented in Section 5.1.1, needed to detect that executions from different components are consistent.

The dissertation applies local transition merging together with partial order semantics. Local transition merging can, in principle, also be applied with on-the-fly determinization. This construct could be beneficial in creating an efficient BMC technique. However, as demonstrated in Section 5.3, it seems hard to conceive a simple encoding for this execution model. Another way of obtaining the product from this composition operator is to determinize the components offline using the standard subset construction. Even though it might work with some examples, the determinization construct can in the worst case result in an exponential blow-up of the components.

The dissertation presents some experimental results comparing the presented non-standard execution models to an implementation of the interleaving semantics. With the studied test cases and with the used SAT solver, the following conclusions can be drawn. In many cases the lower bound obtained using step semantics results in a lower running time of the SAT solver. However, it is a bit surprising that limiting the executions to a normal form, i.e., applying process semantics, typically results in longer running times than those for step semantics. As a second conclusion, applying on-the-fly determinization seems to yield lower running times. Also in this case, applying process semantics seems not to be a good idea.

When local transition merging is applied, with these test cases drastic reductions in the bound are achieved. Indeed, many deadlocks are found in a single step. However, this is not fully reflected in the running times of the SAT solver. This is due to the fact that the encodings for execution models applying step / process semantics and on-the-fly determinization are linear whereas the encoding for the execution model applying local transition merging is cubic in the size of the system's alphabet. With these test cases, the running times are comparable to those of the other execution models.

The cubic complexity of the encoding for local transition merging is due to a single constraint modeling the consistency of the executed actions (see Section 5.1.1). The experimental section contains also data from an iterative approach (presented in Section 5.1.3) where this constraint is initially omitted and thus a linear formula is obtained. Omitting a constraint has the obvious impact of potentially introducing spurious counterexamples. To rule these out, iteration is needed. With the studied test cases, the iterative approach performs very well, when only a few steps of iteration is needed. However, there are examples where the iterative technique is clearly worse than the step / process execution models and also the local transition merging model with full consistency check.

Finally, the non-standard execution models are compared to state-of-the-art model checking tools. This comparison reiterates the already known results concerning BMC, namely that compared to complete model checking

techniques:

- BMC is at its best in finding shallow errors and
- there can be system models whose transition relation is too large for, for instance, BDD based techniques but from which BMC techniques can find errors.

8.1 FUTURE WORK

The presented work suggest interesting topics for future research. Some of them are as follows:

- This dissertation does not combine process semantics with local transition merging. A topic for future work is to define a suitable process criterion and see if any gains in the running time can be obtained.
- Given a tuple of LTSs, their process product and determinized process product can contain fewer transitions than their synchronized product. This could be used in creating an efficient explicit state model checking technique.
- Work has been carried out to develop solvers that in addition to propositional logic are able to solve integer equations (see e.g. [12]). Using such a solver, it is possible to extend the studied system model, LTSs, with more PROMELA like features, like integer variables and FIFOs. Then, however, the condition of independence of actions becomes more complex than in the case of simple action labels. The same holds for the process condition.
- In this dissertation, the properties to be verified are limited to reachability properties. An extension to handle, for instance, full LTL is interesting. This work could use the ideas presented in [41] as a starting point.
- By using on-the-fly determinization, it seems possible to modify a BMC encoding so that the models of the formula represent actions sequences that are *not* executions of the system (complementation). This could be used to solve bounded trace containment between two concurrent systems.

In general, taking an alternative view on the transition relation of a concurrent system, as is done in this dissertation, is by no means limited to BMC. Therefore, the presented composition operators can also be applied to create for instance a BDD based model checker. An interesting topic for further work is the use of the proposed methods inside a counterexample based abstraction refinement framework.

Bibliography

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'2000)*, volume 1785 of *Lecture Notes in Computer Science*, pages 411 – 425. Springer-Verlag, 2000.

[2] N. Amla, R. Kurshan, K. L. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. In *Proceedings of the 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, volume 2619 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[3] A. Arnold. *Finite Transition Systems*. Prentice Hall, 1994.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[5] M. Awedh and F. Somenzi. Increasing the robustness of bounded model checking by computing lower bounds on the reachable states. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'2004)*, volume 3312 of *Lecture Notes in Computer Science*, pages 230 – 240. Springer-Verlag, 2004.

[6] E. Best and R. Devillers. Sequential and concurrent behaviour in petri net theory. In *Theoretical Computer Science 55*, volume 1, pages 87–136, 1987.

[7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Zhu Y. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Conference on Design Automation (DAC'1999)*, pages 317 – 320. ACM Press, 1999.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[9] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Proceedings of the 3rd International Conference on Formal Methods in Computer Aided Design (FMCAD'2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 2000.

[10] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 372 – 389. Springer-Verlag, 2000.

[11] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer Aided Verification (CAV'2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.

[12] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Proceedings of the 11th Interanational Conference on Tools and Algorithms for the Construction Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 2005.

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[14] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205 – 213, 1991.

[15] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293 – 318, 1992.

[16] J. Burch, E. M. Clarke, K. L. McMillan, D. Dill, and L. Hwang. Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[17] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[18] P. Chaudan, E. M. Clarke, J. Kukula, S. Sapra, G. Vieth, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'2002)*, volume 2517 of *Lecture Notes in Computer Science*, pages 33 – 51. Springer-Verlag, 2002.

[19] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Journal on Software Tools for Technology Transfer*, 2(4):410 – 425, 2000.

[20] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the encoding of LTL model checking into SAT. In *Proceedings of the 3rd International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2002)*, number 2294 in Lecture Notes in Computer Science, pages 196–207. Springer-Verlag, 2002.

[21] E. M. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logics. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52 – 71. Springer-Verlag, 1981.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[23] E. M. Clarke, O. Grunberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th international conference on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[24] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer-Verlag, 2004.

[25] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tachhella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–452. Springer-Verlag, 2001.

[26] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[27] J. C. Corbett and G. Avrunin. Using integer programming to verify general safet and liveneses properties. *Formal Methods in System Design*, 6(1):97 – 123, 1995.

[28] W. Craig. Linear reasoning: A new form of Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[29] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201 – 215, 1960.

[30] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction (CADE'2002)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, 2002.

[31] J. Desel and W. Reisig. Place/transition Petri nets. In *Lectures on Petri Nets I : Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 122 – 173. Springer-Verlag, 1998.

[32] D. Dill. The Murϕ verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390 – 393. Springer-Verlag, 1996.

[33] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'1999)*, number 1664 in Lecture Notes in Computer Science, pages 2–20. Springer-Verlag, 1999.

[34] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[35] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC'2003)*, July 2003.

[36] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems.* PhD thesis, University of Liege, 1995.

[37] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exposition (DATE'02)*, pages 142 – 149. IEEE Computer Society, 2002.

[38] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[39] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, volume 2421 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2001.

[40] K. Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-safe Petri Nets.* PhD thesis, Helsinki University of Technology, 2002.

[41] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming (TPLP)*, 3(4&5):519–550, 2003.

[42] G. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2003.

[43] J. HoonSang, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 519 – 522. Springer-Verlag, 2004.

[44] J. HoonSang and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC'2004)*, 2004.

[45] ITU-T. CCITT Specification and Descrtiption Language (SDL). Technical Report Z.100 (1993), ITU-T, 1994.

[46] T. A. Junttila. Boolean circuit tools (including BCzChaff), May 2003. `http://www.tcs.hut.fi/~tjunttil/circuits`.

[47] T. A. Junttila. *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms.* PhD thesis, Helsinki University of Technology, 2003.

[48] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability testing. In *Proceedings of the 1st International Conference on Computational Logic - CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567. Springer-Verlag, 2000.

[49] T. Jussila. BMC via dynamic atomicity analysis. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD'2004)*. IEEE Computer Society, June 2004.

[50] T. Jussila. Efficient bounded reachability through iterative strengthening. In *Concurrency, Specification and Programming CS&P'2004*, 2004.

[51] T. Jussila, K. Heljanko, and I. Niemelä. BMC via on-the-fly determinization. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC'2003)*, 2003.

[52] T. Jussila, K. Heljanko, and I. Niemelä. BMC via on-the-fly determinization. *Journal on Software Tools for Technology Transfer (STTT)*, 7(2), 2005.

[53] S. Katz and H. Miller. Saving space by fully exploiting invisible transitions. *Formal Methods in System Design*, 14(3):311–332, 1999.

[54] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Proceedings of the Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer-Verlag, 1988.

[55] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th Conference on Artificial Intelligence (ECAI'1992)*, pages 359–363. John Wiley and Sons, 1992.

[56] V. Khomenko and M. Koutny. Branching processes of high-level Petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 458 – 472. Springer-Verlag, 2003.

[57] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2003)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer-Verlag, 2003.

[58] O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Israel Institute of Technology, 1995.

[59] R. Kurshan. *Computer-Aided Verification of Co-ordinating Processes*. Princeton University Press, 1994.

[60] R. Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 569–581. Springer-Verlag, 2002.

[61] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'2004)*, volume 2937 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[62] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the Fourth International Workshop on Computer Aided Verification (CAV'1992)*, volume 663 of *Lecture Notes in Computer Science*, pages 164 – 177. Springer-Verlag, 1992.

[63] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[64] K. L. McMillan. Applying SAT methods in unbounded model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264, 2002.

[65] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th international conference on Computer Aided Verification (CAV'2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2003.

[66] K. L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2005.

[67] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS'2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 411 – 425. Springer-Verlag, 2003.

[68] R. Milner. *Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[69] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[70] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'2001)*, pages 530 – 535. ACM, 2001.

[71] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13(1):85 – 108, 1980.

[72] S. Ogata, T. Tsuchiya, and T. Kikuno. SAT-based verification of safe Petri nets. In *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'2004)*, volume 3299 of *Lecture Notes in Computer Science*, pages 79 – 92. Springer-Verlag, 2004.

[73] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[74] C. Papadimitriou and H. Lewis. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[75] W. Penczek, B. Wózna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 50:1 – 22, 2002.

[76] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[77] J. P. Quielle and J. Sifakis. Specification and verification of concurrents systems in CESAR. In *Proceedings of the 5th International Symposium of Programming*, pages 337 – 350, 1981.

[78] J. Rintanen, K. Heljanko, and I. Niemelä. Parallel encodings of classical planning as satisfiability. In *Logics in Artificial Intelligence: 9th European Conference*, volume 3225 of *Lecture Notes in Artificial Intelligence*, pages 307 – 319. Springer Verlag, 2004.

[79] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: Parallel plans and algorithms for plan search. Technical Report 216, Institute of Computer Science at Freiburg University, 2005.

[80] L. Ryan. Efficient algorithm for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2002.

[81] S. Sheeran, S. Singh, and G. Stålmarck. Checking safety properities using induction and a SAT-solver. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108 – 125. Springer-Verlag, 2000.

[82] O. Strichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480 – 494. Springer-Verlag, 2000.

[83] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of the 11th Advanced Reseach Working Conference on Correct Hardware Design and Verification Methods (CHARME'2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58 – 70. Springer-Verlag, 2001.

[84] O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):1 – 24, 2004.

[85] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.

[86] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[87] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, 1986.

[88] K. Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, 1998.

[89] C. Wang, H. Jin, G. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proceedings of the 41th Design Automation Conference (DAC'2004)*, pages 535 – 538. ACM, 2004.

[90] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124 – 138. Springer-Verlag, 2000.

[91] Karen Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, Israel Institute of Technology, 2000.

[92] E. Zarpas. Simple yet efficient improvements of SAT based bounded model checking. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'2004)*, volume 3312 of *Lecture Notes in Computer Science*, pages 174–185. Springer-Verlag, 2004.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A84 Johan Wallén
On the Differential and Linear Properties of Addition. December 2003.

HUT-TCS-A85 Emilia Oikarinen
Testing the Equivalence of Disjunctive Logic Programs. December 2003.

HUT-TCS-A86 Tommi Syrjänen
Logic Programming with Cardinality Constraints. December 2003.

HUT-TCS-A87 Harri Haanpää, Patric R. J. Östergård
Sets in Abelian Groups with Distinct Sums of Pairs. February 2004.

HUT-TCS-A88 Harri Haanpää
Minimum Sum and Difference Covers of Abelian Groups. February 2004.

HUT-TCS-A89 Harri Haanpää
Constructing Certain Combinatorial Structures by Computational Methods. February 2004.

HUT-TCS-A90 Matti Järvisalo
Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking.
March 2004.

HUT-TCS-A91 Mikko Särelä
Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004.

HUT-TCS-A92 Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila
Simple Bounded LTL Model Checking. July 2004.

HUT-TCS-A93 Tuomo Pyhälä
Specification-Based Test Selection in Formal Conformance Testing. August 2004.

HUT-TCS-A94 Petteri Kaski
Algorithms for Classification of Combinatorial Objects. June 2005.

HUT-TCS-A95 Timo Latvala
Automata-Theoretic and Bounded Model Checking for Linear Temporal Logic. August 2005.

HUT-TCS-A96 Heikki Tauriainen
A Note on the Worst-Case Memory Requirements of Generalized Nested Depth-First Search.
September 2005.

HUT-TCS-A97 Toni Jussila
On Bounded Model Checking of Asynchronous Systems. October 2005.

ISBN 951-22-7903-7
ISSN 1457-7615