Technische Universität München
Department of Computer Science

Master's Thesis in Computer Science

Implementation and Verification of Partial Order Reduction for On-The-Fly Model Checking

Implementierung und Verifizierung von Partial Order Reduction für On-The-Fly Model Checking

Author:	Julian Brunner
Supervisor:	Prof. Tobias Nipkow
Advisors:	Prof. Javier Esparza
	Dr. Peter Lammich
Date:	July 15, 2014

Declaration

I confirm that this master's thesis is my own work and that I have documented all sources and materials used.

Acknowledgments

I would like to thank my supervisor Prof. Tobias Nipkow for providing me with the topic of and the opportunity to write this thesis. I would also like to thank my advisor Prof. Javier Esparza for recommending the literature that made this thesis possible, as well as for supporting me with his knowledge on both automata theory and model checking throughout my work on the thesis. Furthermore, I would like to thank my advisor Dr. Peter Lammich for always providing me with answers to my questions on both the refinement framework for monadic programs as well as more general aspects of the proof assistant Isabelle.

My thanks also go to Prof. Doron Peled for taking the time to reply to our inquiries regarding one of his papers and for recommending further literature.

Finally, I want to express my gratitude towards my friends Julian Asamer and Yannick Mahlich, who agreed to proofread this thesis and who provided me with valuable feedback about it.

Abstract

In this thesis, we present our effort of formally verifying the on-the-fly partial order reduction technique described in [Pel96] using the proof assistant Isabelle in conjunction with the HOL logic library.

First, we briefly introduce the ideas of automata-based model checking, partial order reduction, and formal verification. The on-the-fly partial order reduction algorithm we want to formalize is built on top of an off-line partial order reduction algorithm. Thus, we first present the implementation and verification of significant parts of this off-line partial order reduction algorithm. Next, we describe the architecture of the correctness proof for the on-the-fly partial order reduction algorithm. There, we point out a problem with one of the proof steps and provide a counterexample showing that the proof of this step is not valid. Finally, we give an overview of the formal theories and present ideas for further work.

Contents

Contents ix

1 Introduction 1

2 Automata-Based Model Checking 4
- 2.1 Inputs 4
  - 2.1.1 System 4
  - 2.1.2 Correctness Property 5
  - 2.1.3 Semantic Connection 6
- 2.2 Problem 7
- 2.3 Algorithm 8
  - 2.3.1 Formula Automaton 8
  - 2.3.2 Product Automaton 8
  - 2.3.3 Reachable Accepting Cycles 9
- 2.4 Off-Line vs. On-The-Fly 9
- 2.5 Verification 11

3 Partial Order Reduction 12
- 3.1 Fairness Assumptions 13
- 3.2 Static Analysis 13
- 3.3 Off-Line Model Checking 13
- 3.4 On-The-Fly Model Checking 14
  - 3.4.1 Search Stack Projection 15
  - 3.4.2 Sequential Product 16

4 Formal Verification 18
- 4.1 Informal Proofs 18
- 4.2 Formal Proofs 19
- 4.3 Proof Assistants 20
- 4.4 Formally Verified Model Checking 21

5 Automaton Generation **22**
5.1 Nondeterminism and Refinement 22
5.2 Generalized Depth-First Search 23
5.3 Generated Büchi Automata 24

6 Auxiliary Definitions **26**
6.1 Stutter Invariance 26
6.2 Independence Relation 27
6.3 Equivalence Relations 28
6.4 Visibility Sets 30
6.5 Independent Occurrence 30
6.6 Independent Decomposition 31

7 Verification of Off-Line Partial Order Reduction **32**
7.1 Assumptions 32
7.2 Lemma 3.7 34
7.3 Theorem 3.9 35
7.4 Theorem 3.11 36
7.4.1 Construction Scheme 37
7.4.2 Construction Invariants 39
7.4.3 Construction Step 40
7.4.4 Transition to Infinite Sequences 42
7.4.5 Final Consequences 43
7.5 Theorem 3.12 44
7.6 Correctness Theorem 45

8 Verification of On-The-Fly Partial Order Reduction **46**
8.1 Automaton Completion 46
8.2 The Reduced Completed Product Automaton 48
8.3 Proof Architecture 50
8.4 Reduced Product Automaton Language 51
8.4.1 Setup 52
8.4.2 Reduction 53
8.4.3 Product 54
8.4.4 Completion 56
8.4.5 Contradiction 58
8.4.6 Consequences 59

9 Verification Results **60**

10 Conclusion **62**

A Definitions **64**
A.1 Sequences 64
A.2 Linear Temporal Logic 65
A.3 Büchi Automata 66
A.3.1 Deterministic Automata 67
A.3.2 Product Automata 68

Bibliography **69**

Chapter 1

Introduction

Many of the things we take for granted in our everyday lives critically depend on computer systems working correctly. While a user interface glitch in a smartphone game may not have any significant consequences, computer systems are also used in many safety or security critical environments, such as aviation and banking. Correctness also becomes important when the system in question is used heavily, like an operating system kernel or a web server. For some systems, testing may be sufficient for fulfilling the correctness requirements. However, when the consequences of a system working incorrectly are severe, a more rigorous method of ensuring its correctness may be in order.

In these cases, it seems desirable to verify the system to gain more trust in its correctness. The verification of a system amounts to the construction of mathematical proofs asserting that certain statements about its behavior during execution hold. Unfortunately, constructing these proofs manually is time-consuming and tedious at best, and error-prone at worst. Model checking is aimed at solving these problems, describing a set of techniques that allow for fully automatic construction of proofs for certain correctness properties of systems [CGP99; Hol03].

A model checker takes as input a model of the system to be checked, as well as a formal description of the correctness property that should be verified. In this context, a model is a formal description of the system's behavior. For hardware systems, these models have to be derived from the way the system is built. Given a software system, there already exists a formal description of its behavior, namely, its source code. Nonetheless, it is usually necessary to construct a separate model, even for software systems, since the source code often contains too many technical details while lacking semantic information about the system that can improve the model checking process significantly, such as variable bounds. The model of a

given system can be viewed as a formal description of the possible execution runs of this system. The formal description of the correctness property is usually realized using formulae in temporal logic. A formula corresponding to a specific property can then be used to decide if the property holds for a given execution run.

The problem of model checking consists of deciding whether the correctness property holds for all execution runs of the system. We know from Rice's theorem that this is undecidable for general systems and nontrivial semantic properties. Because of this, model checking is usually restricted to finite state systems and certain classes of logical formulae, such that it becomes decidable. This allows the model checker to fully automatically check whether the given property holds for all execution runs of the given system, or, in the case that it does not, supply an execution run for which the property does not hold to serve as a counterexample.

Given the restriction of using finite state systems, a common use case for model checking are systems that do not involve large natural numbers or collections, but rather focus on control flow and parallelism. This is incidentally also an area where humans are likely to make mistakes, so it is a natural fit for verification.

Since a single model checker may be used to assess the correctness of many systems, it is important for the model checker itself to be correct. An incorrect model checker can at best not yield any additional trust in the checked systems, and at worst, convince the user of the model checker that their system is correct when it actually is not. This is why we are concerned with verifying the model checker itself. As mentioned before, manual verification of software is a very time-consuming task. As such, verified model checkers like the one developed as part of the CAVA project [NEN; Esp+13; Esp+14] cannot yet compete with unverified ones like the SPIN model checker [Hol03] when it comes to performance. This is due to the fact that many of the optimizations that have been implemented in unverified model checkers have not been formally proven correct yet.

This thesis sets out to implement and verify one such optimization, which is called partial order reduction [Pel98]. Partial order reduction is designed to counteract what is known as the state space explosion problem, a phenomenon caused by the many possible ways of interleaving the operations of processes executing in parallel. The general idea of the optimization is that even though there may be many possible ways of interleaving, most of them will produce the same result and thus, do not have to be considered when model checking.

Partial order reduction is complex. Because of this, mistakes have been found in the algorithms and/or correctness proofs of early partial order reduction techniques, and some were even found in more recent algorithms [HPY96]. For this reason, verifying partial order reduction seems like a worthwhile effort, as it would significantly

increase the trustworthiness of the techniques that are involved. To the best of our knowledge, the only other attempt at verifying a partial order reduction algorithm was made in [CP96]. However, this attempt merely covers the case of off-line model checking under strong fairness assumptions. Furthermore, it reasons about partial order reduction on a very abstract level and thus does not result in an executable algorithm. This thesis aims at implementing and verifying the partial order reduction algorithm presented in [Pel96]. More specifically, we consider the version of the algorithm which covers on-the-fly model checking without fairness assumptions. Our goal is also to produce an executable algorithm in the end.

We will start by giving an overview of automata-based model checking in chapter 2. Chapter 3 introduces the specific partial order reduction algorithm that this thesis is concerned with. Next, chapter 4 gives an overview of the formal verification techniques used for this thesis. In chapters 5 and 6, we introduce some concepts needed for the correctness proofs that make up the main part of the thesis in chapters 7 and 8. Finally, we conclude the verification effort and the thesis in chapters 9 and 10, respectively. Finally, appendix A introduces some of the more basic concepts needed for this thesis.

Chapter 2

Automata-Based Model Checking

In this thesis, we focus on the automata-based approach to model checking. The idea here is to represent both the system and the correctness property to be checked using finite automata. It is then possible to check whether the correctness property holds for the system by examining the product of those two automata. The following sections describe the details of this process.

2.1 Inputs

As mentioned before, the first step of the model checking process consists of deriving a formal representation from the inputs, which are comprised of the system and the correctness property. Thus, we start by describing these inputs as well as the formal entities that we use to represent them.

2.1.1 System

The primary input of the model checking process is the system to be checked. We want to derive a formal description, or model, from the system. Since we only consider finite state systems, a finite automaton can be used to model the system, with the automaton states corresponding to the system states and the automaton labels corresponding to the system operations which cause transitions between the system states. This automaton is called the system automaton. The derivation of a system automaton from some system is heavily dependent on the type of system that is considered and as such, is not part of the generic model checking process covered in this thesis. We will thus assume that there is some procedure in place

for performing this step and consider the system automaton to be available to the remainder of the model checking process.

For the system automaton, Büchi automata as defined in section A.3 are used, which take infinite words as input. This allows modeling systems that keep executing indefinitely. Systems with finite executions can also be modeled using a cyclic halting state.

We require the system automaton to be a deterministic Büchi automaton as defined in section A.3.1. Note that the system automaton being deterministic does not imply that the system itself is deterministic. This is because multiple operations may be enabled at the same state, causing the system automaton to nondeterministically choose between them.

Given a system automaton S, we define the initial state ι, the set of enabled operations en, and the execution function ex as follows:

$$\{\iota\} = \mathcal{I}(S) \tag{2.1a}$$

$$a \in \mathrm{en}(q_1) \iff \exists\, q_2.\ (q_1, a, q_2) \in \Delta(S) \tag{2.1b}$$

$$\mathrm{ex}(a, q_1) = q_2 \iff (q_1, a, q_2) \in \Delta(S) \tag{2.1c}$$

We also require all states of the system automaton to be accepting. This makes sense, considering that when constructing the system automaton from the system, we are not interested in any specific properties that the runs in the system fulfill, but rather only in which runs are possible at all.

2.1.2 Correctness Property

The second input of the model checking process is the correctness property. As with the system, it is necessary to derive a formal representation from the correctness property first. The version of automata-based model checking we consider in this thesis uses formulae in linear temporal logic as defined in section A.2 to represent the correctness property. LTL formulae allow for the concise specification of common correctness properties of parallel systems such as being free of deadlocks, staying responsive, and processes always making progress. They are thus well-suited to formally represent correctness properties to be used in model checking. However, since the correctness property is usually described informally at first, there is no generic procedure for deriving such a formula from it. Thus, as with the system automaton, we will assume that the formula corresponding to the correctness property is available to the remainder of the model checking process.

2.1.3 Semantic Connection

So far, the formula, like all LTL formulae, is just a predicate on sequences of propositional variables. However, the propositional variables are just names without any meaning attached to them. In order for the formula to describe a property of runs in the system automaton, some semantic connection has be established between the system states and the propositional variables that occur in the formula. Thus, we expect as third and final input of the model checking process the interpretation function, which maps system states to sets of propositional variables.

Given an interpretation function P, for every system state q, we call $P(q)$ the interpretation of the state q. This way, the interpretation function determines the meaning of the propositional variables in each system state. For instance, we may know that the system has crashed when it reaches the state q_4 or the state q_7, and that it has not crashed while it is in any other state. We can then use the interpretation function to attach this meaning to the propositional variable p_2. To do so, we define the interpretation function in such a way that $p_2 \in P(q_4)$ and $p_2 \in P(q_7)$, with p_2 not being a member of the interpretation of any other system state. With this definition, we know that the system has crashed whenever the propositional variable p_2 is part of a system state's interpretation. This way, when the formula makes a statement about the propositional variable p_2, it indirectly also makes a statement about system crashes. The set of propositional variables that makes up the interpretation of some system state can thus be taken to contain exactly those propositional variables that hold in this system state.

We extend the interpretation function P to work not just with individual system states, but with whole sequences of system states as well. Given a sequence of system states r, its interpretation $P(r)$ is the sequence of sets of propositional variables obtained by applying P to each of the system states in r. It is then possible to check whether an interpreted sequence of system states is accepted by the formula.

Next, given an automaton S, we introduce the concept of the interpreted language $\mathcal{L}_P(S)$ of the automaton S, which is defined as follows:

$$\mathcal{L}_P(S) = P[\text{runs}(S)] = \{P(r) \mid r \in \text{runs}(S)\} \tag{2.2}$$

That is, the interpreted language of an automaton is the image of the automaton's runs under the interpretation function, or, equivalently, the interpreted language of an automaton consists of the interpretations of all the runs in the automaton.

This definition allows us to more easily make the connection between the system automaton and the formula. This is because the kind of sequences contained in the interpreted language of the system automaton are the same as those contained in the language of the formula, that is, sequences of sets of propositional variables.

Our final definition concerning the interpretation function P consists of extending it to work with whole automata. Given an automaton S, we define the interpreted automaton $P(S)$ to be identical to the original automaton S, except for the set of labels and the transition relation, which are defined as follows:

$$\Sigma(P(S)) = P[\mathcal{Q}(S)] = \{P(q) \mid q \in \mathcal{Q}(S)\} \tag{2.3a}$$

$$\Delta(P(S)) = \{(p, P(p), q) \mid (p, a, q) \in \Delta(S)\} \tag{2.3b}$$

Here, the set of labels of the interpreted automaton is the image of the set of labels of the original automaton under the interpretation function, or, equivalently, the set of labels of the interpreted automaton is equal to the set of interpreted states of the original automaton. Furthermore, for each transition, we replace the original automaton's labels with the interpretation of the source state of the transition, discarding all information about the labels of the original automaton in the process.

The motivation behind this construction is similar to that of the interpreted language and just like it, the language of the interpreted automaton also contains sequences of sets of propositional variables. Indeed, the following identity can be proven:

$$\mathcal{L}(P(S)) = \mathcal{L}_P(S) \tag{2.4}$$

That is, the language of the interpreted automaton is equal to the interpreted language of the original automaton.

2.2 Problem

Having established a formal representation for each of the inputs involved, we can now state the model checking problem. Given a system automaton S, a formula φ, and an interpretation function P, it consists of deciding whether or not the following proposition holds:

$$\mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) \tag{2.5}$$

That is, model checking answers the question of whether or not the interpreted language of the system automaton is contained in the language of the formula. Alternatively, the model checking problem can be stated as follows:

$$r \in \text{runs}(S) \implies P(r) \models \varphi \tag{2.6}$$

This implication states that for all runs of the system automaton, their interpretation is accepted by the formula. The intuitive meaning here is that all executions of the system are accepted by the formula representing the checked property.

2.3 Algorithm

In the previous section, we have stated the problem of model checking, which tells us which question needs to be answered. However, the way it was stated did not give any indication as to how it can be decided. To this end, we present an executable algorithm which decides the model checking problem in this section.

2.3.1 Formula Automaton

An important property of LTL formulae is that for each of them, there is a corresponding Büchi automaton that accepts precisely those words that are accepted by the formula [Ger+96; SL14]. Moreover, constructing the corresponding Büchi automaton for some formula can be performed algorithmically. We can thus assume that there is a function F performing this construction, called the formula automaton function. This way, the Büchi automaton corresponding to the formula φ is denoted by $F(\varphi)$. The correctness theorem for F then states that for all formulae φ, the following holds:

$$\mathcal{L}(F(\varphi)) = \mathcal{L}(\varphi) \tag{2.7}$$

That is, the language of the Büchi automaton corresponding to the formula is equal to the language of the formula itself.

We then define the formula automaton to be $F(\neg\varphi)$. That is, the formula automaton is the Büchi automaton corresponding to the negation of the formula that represents the correctness property. It accepts precisely those words that violate the property expressed by the formula.

2.3.2 Product Automaton

Let S be the system automaton, let φ be the formula, and let P be the interpretation function. Using the product construction for Büchi automata given in section A.3.2, we define the product automaton to be the product of the interpreted system automaton and the formula automaton:

$$P(S) \times F(\neg\varphi) \tag{2.8}$$

This automaton accepts precisely those words that are interpreted runs of the system and which also violate the formula. Thus, if it accepts any word, that is, its language is non-empty, we know that the system has a run that violates the formula. Hence, the problem of model checking has been reduced to the problem of deciding language emptiness for Büchi automata.

2.3.3 Reachable Accepting Cycles

According to the previous section, deciding the model checking problem is equivalent to deciding language emptiness for Büchi automata. This problem can be solved by searching for reachable accepting cycles in the graph induced by the Büchi automaton. A reachable accepting cycle is also called a lasso, since its graphical representation looks just like one. From the acceptance condition of Büchi automata described in section A.3, we can conclude that each reachable accepting cycle corresponds to some word in the language. Thus, a Büchi automaton's language is empty if and only if it has no reachable accepting cycles.

Searching for reachable accepting cycles can be done using nested depth-first-search [Cou+92; SE05]. We can thus assume that a function $\mathrm{has_lasso}$ exists which returns $\top$ if and only if a reachable accepting cycle exists in the given automaton.

Using the function $\mathrm{has_lasso}$ in conjunction with the product automaton, we can then define the executable model checking function as follows:

$$\mathrm{check}(S, \varphi, P) = \neg\mathrm{has_lasso}(P(S) \times F(\neg\varphi)) \tag{2.9}$$

This function checks if the product automaton contains a reachable accepting cycle using the $\mathrm{has_lasso}$ function and then returns $\bot$ if there is one and $\top$ if there is not.

2.4 Off-Line vs. On-The-Fly

There are two basic ways of implementing the model checking algorithm described in the previous section. The naive approach is called off-line model checking and consists of explicitly constructing the system automaton, formula automaton, and product automaton in order to perform the nested depth-first search. This means that all the states and transitions of these automata first have to be constructed and then stored somewhere. Doing so can have a significant impact on the performance of the model checker, especially when the system automaton is large.

An improved approach known as on-the-fly model checking avoids the explicit construction of the system automaton and product automaton. To do so, it represents Büchi automata using transition functions instead of transition relations. The transition function of the system automaton can be computed on the fly using a high-level representation of the system, which is usually some kind of program source code. The product automaton is constructed from this implicit representation of the system automaton and the formula automaton according to the product construction for Büchi automata given in section A.3.2. This algorithm evaluates the transition

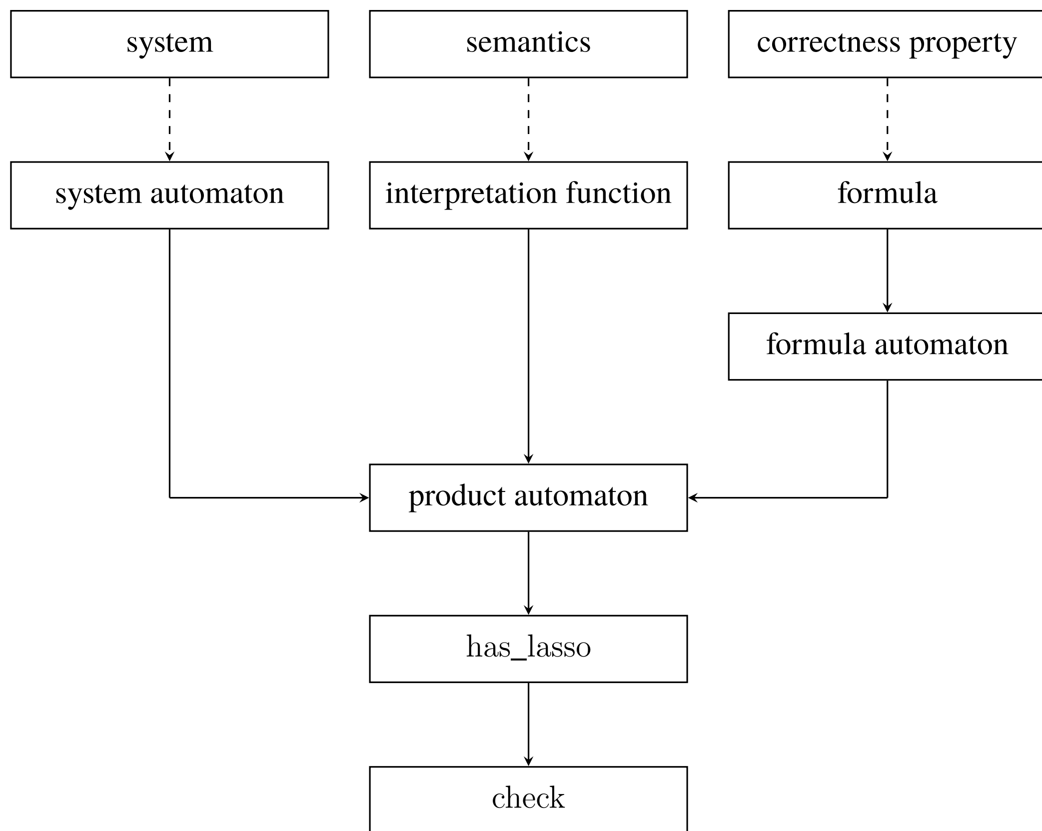


Figure 2.1: Overview of the model checking process. The dashed arrows represent the process of modeling, which derives formal representations from the raw and possibly informal inputs. The solid arrows show which of the formal entities are used to derive other formal entities or results.

function of the product automaton on the fly, using the transition functions of the system automaton and the formula automaton.

On-the-fly model checking thus allows performing nested depth-first search on the product automaton without ever constructing it or the system automaton explicitly. Instead, the nested depth-first search algorithm explores the product automaton on the fly, starting with the set of initial states and adding states as transitions are discovered via the product automaton's transition function. This has various advantages. Firstly, if the property does not hold, a counterexample may be found before the whole product automaton is explored. Secondly, even if the checked property does hold, parts of the system automaton that are not relevant with respect to the checked property may not be explored at all, since these states may not appear in the product automaton.

2.5 Verification

We now present a proof of the fact that the function check actually decides the model checking problem stated in equation 2.5. We arrive at the following steps:

$$\begin{aligned}
\mathrm{check}(S, \varphi, P) &\iff \neg\mathrm{has_lasso}(P(S) \times F(\neg\varphi)) && \text{(2.10a)} \\
&\iff P(S) \times F(\neg\varphi) \text{ has no reachable accepting cycle} && \text{(2.10b)} \\
&\iff \mathcal{L}(P(S) \times F(\neg\varphi)) = \{\} && \text{(2.10c)} \\
&\iff \mathcal{L}(P(S)) \cap \mathcal{L}(F(\neg\varphi)) = \{\} && \text{(2.10d)} \\
&\iff \mathcal{L}_P(S) \cap \overline{\mathcal{L}(\varphi)} = \{\} && \text{(2.10e)} \\
&\iff \mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) && \text{(2.10f)}
\end{aligned}$$

In the following, we present some explanation for each step.

2.10a This step follows directly from the definition of the model checking function given in equation 2.9.

2.10b The function $\mathrm{has_lasso}$ uses nested depth-first search to determine if an automaton has a reachable accepting cycle. Given that it is implemented correctly, $\mathrm{has_lasso}(P(S) \times F(\neg\varphi))$ does not hold if and only if there is no reachable accepting cycle in the product automaton.

2.10c From the acceptance condition of Büchi automata, we can conclude that each reachable accepting cycle corresponds to some word in the language. Thus, the lack of reachable accepting cycles in the product automaton is equivalent to the emptiness of its language.

2.10d This step follows from the correctness theorem about the Büchi automaton product given in equation A.6, stating that for all automata A and B, where for at least one of them, all states are accepting, it holds that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$.

2.10e This step is justified by equation 2.4, which states that for all automata S, it holds that $\mathcal{L}(P(S)) = \mathcal{L}_P(S)$, equation 2.7, which states that for all formulae φ, it holds that $\mathcal{L}(F(\varphi)) = \mathcal{L}(\varphi)$, and equation A.4, which states that for all formulae φ, it holds that $\mathcal{L}(\neg\varphi) = \overline{\mathcal{L}(\varphi)}$.

2.10f A mere set theoretic transformation is performed in this step.

Chapter 3

Partial Order Reduction

In this thesis, we set out to formally verify a partial order reduction algorithm for on-the-fly model checking. In this chapter, we introduce the basic ideas behind partial order reduction, as well as the specific algorithm that is considered in the formal verification effort of this thesis.

As mentioned before, model checking is often applied to parallel systems. With different processes executing in parallel, there are many possible ways of interleaving their atomic operations. Since one usually cannot make any strong assumptions about the scheduling of these processes, all possible interleaving orders have to be considered in order to perform model checking. Due to this, a large number of system states are possible, a phenomenon known as state space explosion. This is unfortunate, considering that the checked correctness property may be largely insensitive to the order of interleaving. For instance, two processes may operate independently of each other, only performing local tasks whose results are not part of the correctness property. In this example, it does not matter in which order the two sequences of atomic operations are interleaved, since the are independent of both each other and the correctness property.

This scenario strongly hints at a possible optimization for reducing the number of system states by choosing a canonical interleaving order for independent operations like local assignments. This is exactly what partial order reduction does. Using static program analysis, partial order reduction determines at which system states, operations can be omitted. This way, redundant interleavings are discarded, greatly reducing the number of system states.

Various specific instances of this general idea have been proposed [Pel98; Val91; Pel93; God96]. In this thesis, we consider the algorithm presented in [Pel96], which performs ample set partial order reduction for on-the-fly model checking.

3.1 Fairness Assumptions

When model checking, certain properties concerning process scheduling are sometimes assumed. In the case of partial order reduction, a fairness assumption called F, introduced in [Pel96, Page 43], can both improve reduction performance and simplify the correctness proof of the algorithm. However, as we want to verify partial order reduction in its most generic form, no fairness assumptions are employed throughout the rest of the thesis and the more complex correctness proof is used.

3.2 Static Analysis

Ample set partial order reduction as described in [Pel96] consists of two phases, the static analysis phase and the reduction phase which uses the results from the static analysis phase. During the static analysis phase, control flow information about the system is used to determine which operations can be omitted at which system states. Since this control flow information is hard to extract from the system automaton, the static analysis phase is performed on a higher-level representation of the system. In the case of a software system, this is typically the program source code. Thus, the static analysis phase is specific to the system's representation and as such, not part of this thesis. In the following, we will simply assume that its results are available to the reduction phase and that they exhibit the required properties.

The primary result of the static analysis phase is the ample function, written ample. For all system states q, the ample set $\mathrm{ample}(q)$ is a subset of the set of enabled operations $\mathrm{en}(q)$. If, at some system state, the ample set is a proper subset of the set of enabled operations, we say that a reduction has taken place. Intuitively speaking, the constants ι, en, and ex describe the system automaton, while the constants ι, ample, and ex describe the reduced system automaton.

3.3 Off-Line Model Checking

Once the ample function has been obtained from the static analysis phase, the reduction phase can be performed. At first glance, it may seem as if all of the work is already done once the ample function has been obtained, as it can simply be substituted for the function returning the set of enabled operations, resulting in the reduced system automaton.

However, there is an additional complication. For the partial order reduction algorithm to work, the reduced system automaton has to be constructed using depth-first search, with the ample function behaving differently depending on the system states that are on the current search stack. Specifically, a reduction is invalid if one of the operations in the ample set is on the search stack. Thus, the ample function actually has to take the search stack s as an additional parameter, causing the ample set at state q to be $\mathrm{ample}(s, p)$.

The reduced system automaton R can then be constructed using depth-first search exploration. It is subsequently used instead of the system automaton S in the model checking process. With the reduced system automaton being smaller than the system automaton, this can improve both time performance and memory consumption of the model checker. Since the reduced system automaton has to be constructed up front, this partial order reduction technique can be considered an algorithm for off-line model checking.

3.4 On-The-Fly Model Checking

As described in section 2.4, on-the-fly model checking has various advantages over off-line model checking, making it desirable to use this technique in conjunction with partial order reduction. At first glance, partial order reduction seems like a natural fit for on-the-fly model checking, since both it and the search for reachable accepting cycles in on-the-fly model checking are performed using depth-first search. However, the construction turns out to be somewhat tricky.

When doing on-the-fly model checking, we want to avoid explicitly constructing the system automaton and the product automaton. If partial order reduction is not used, this can be done by representing Büchi automata using transition functions, thereby preventing the explicit construction of the transition relations of the system and the product automaton. While the ample function already constitutes an implicit representation of the reduced system automaton, it is not immediately possible to use it as the transition function of a factor of the product automaton. The problem is that the ample function, as it was introduced in the previous sections, takes a search stack consisting of system states as its first parameter. However, when performing depth-first search on the product automaton, the search stack contains product states, so the types of the items on these search stacks are incompatible.

Another possible problem of using the ample function as part of a transition function in nested depth-first search is that the reductions performed during the outer search may not be the same as the ones performed during the inner search. This can

happen as the search stack at some state during the outer search may not be equal to the search stack at the same state during the inner search. If this happens, it can compromise the algorithm's correctness, as reachable accepting cycles in the reduced product automaton may not be found by the nested depth-first search, thereby causing the model checker to deliver the wrong result [HPY96].

The following sections describe some possible adaptations of various parts of the model checking process and the partial order reduction algorithm in order to perform partial order reduction together with on-the-fly model checking.

3.4.1 Search Stack Projection

The primary issue with using partial order reduction together with on-the-fly model checking is that the ample function cannot be used as the transition function of a factor of the product automaton due to the search stack incompatibility mentioned previously. The most straightforward remedy for the situation is to project each product state on the search stack to its first component before passing it to the ample function. This way, the stack of product states is converted to a stack of system states, making it compatible with the ample function.

Of course, the parameters being compatible does not imply that the resulting algorithm is correct. However, we argue that every path the depth-first search can take when exploring the reduced system automaton can also be taken when exploring the product automaton using the ample function. From the viewpoint of the ample function, the only difference is the search stack, whose projection can now contain more system states than in off-line partial order reduction. This can be caused by the depth-first search on the product automaton visiting multiple product states with the same system state component, but different formula state components. However, the ample function will only be more conservative in its reductions when it is passed a search stack with additional states. This way, the reduction algorithm should still be correct.

However, due to the additional system states present on the search stack, this algorithm may reduce less than the off-line partial order reduction algorithm. This may result in a larger product automaton than if off-line partial order reduction was performed in conjunction with the regular Büchi automaton product, thus diminishing the performance gains of the partial order reduction optimization. Another issue is that while the previous paragraph suggests some reasons for the algorithm's correctness, this is still far from an actual proof, developing which may still require substantial work. With these considerations, we decided not to pursue this approach further.

3.4.2 Sequential Product

The approach presented in the previous section is too conservative, weakening the reduction in the process. This is because a transition to one of the system states on the projected search stack does not necessarily close an actual cycle in the product automaton, as the transition in the product automaton may lead to a product state with a different formula state component. The idea behind the sequential product approach is again to process the search stack consisting of product states before passing it to the ample function. However, we now want to do it in such a way that the processed search stack actually contains those system states to which making a transition would close a cycle in the product automaton.

A transition in the product automaton represents transitions in both the system automaton and the formula automaton. Thus, in order to decide if a transition in the system automaton closes a cycle in the product automaton, one also needs to know the transition in the formula automaton that is being made alongside of it. This requires us to determine which transition is being made in the formula automaton before evaluating the ample function. Thus, we can no longer use the regular Büchi automaton product, but instead have to use a custom product construction. We call this construction the sequential product, as it makes its transitions sequentially, first the one in the formula automaton and then the one in the system automaton.

Once the successor state of the formula automaton is known, it is possible to construct the processed search stack. It contains exactly those system states, which, when paired with the successor state of the formula automaton, form a product state that is on the search stack. We can then use this processed search stack and pass it to the ample function in order to obtain the successor state of the system automaton. Finally, the two successor states are combined to form the successor state of the sequential product. A more formal presentation of the sequential product algorithm can be seen in figure 3.1.

for all s, such that $(r, P(p), s) \in \Delta(F(\neg\varphi))$ **do**
  $c' \leftarrow$ the processed search stack with respect to c and s
  for all $a \in \mathrm{ample}(c', p)$ **do**
    return $(P(p), (\mathrm{ex}(a, p), s))$
  end for
end for

Figure 3.1: The transition function of the sequential product. Given the system automaton S, a formula φ, an interpretation function P, a search stack c, a system state p, and a formula state r, the listing shows how to calculate the successor transitions $\delta'(c, (p, r))$ of the sequential product.

Note that in [Pel96, Section 4], the same algorithm is described, albeit in a different fashion. There, instead of processing the search stack to make it compatible with the ample function, the ample function itself is modified.

Also note that this algorithm not only performs the product construction, but also takes care of interpreting the system automaton, as this is a prerequisite for defining the product automaton for model checking.

We mentioned earlier that using the ample function in conjunction with nested depth-first search can compromise the search algorithm's correctness since different reductions may be performed during the outer and the inner search. A simple solution to this problem is to incrementally construct the automaton during the outer depth-first search. The inner depth-first search then uses this partially constructed automaton instead of invoking the ample function, thus guaranteeing that the outer and the inner search runs are performed on the same automaton.

For our formal verification effort, we decided to use this approach for adapting partial order reduction to be used in conjunction with on-the-fly model checking. During the remainder of this thesis, we will call the automaton that is defined using the sequential product the reduced product automaton. The notation used for the sequential product is the same as the one used for the regular Büchi automaton product, resulting in the reduced product automaton being written as $P(R) \times F(\neg\varphi)$. This notation somewhat conceals the fact that this is not a regular Büchi automaton product, but a custom product construction, with the only notational difference being that the reduced system automaton R is being used instead of the system automaton S. However, this detail will not be of much importance throughout the rest of the thesis, such that it is unlikely to cause much confusion.

With the reduced product automaton established, we can now define the model checking function for this on-the-fly partial order reduction approach:

$$\text{check}'(S, \varphi, P) = \neg\text{has_lasso}(P(R) \times F(\neg\varphi)) \tag{3.1}$$

Note that this definition is in complete analogy to that of the regular model checking function in equation 2.9.

Our goal throughout the rest of the thesis is then to prove a correctness theorem similar to the one shown in the set of equations 2.10, but for the on-the-fly partial order reduction model checking function defined in equation 3.1.

Chapter 4

Formal Verification

The main focus of this thesis lies on the formal verification of the partial order reduction algorithms introduced in chapter 3. As mentioned previously, verification describes the construction of mathematical proofs showing the correctness of some algorithm. When formal proofs are used to do so, the process is called formal verification. The following sections give an introduction to formal proofs as well as the tools used to construct them.

4.1 Informal Proofs

The majority of all mathematical proofs are informal. An informal proof is presented in natural language, and is intended to be read and understood by humans. These proofs usually discuss concepts on a very high level, leaving out many details. They sometimes also appeal to the reader's intuition about the involved concepts. This applies especially when these concepts lend themselves to being represented visually, as is the case with finite automata, graphs and geometrical concepts. The intention is to leave out those details that are not important for the understanding or the validity of the proof, and which can also be easily reconstructed if needed.

The advantages of informal proofs are that in leaving out details and appealing to intuition, they are shorter, potentially easier to understand, and that they can focus on the relevant thoughts involved in the proof, without causing distraction by dealing with technical details. However, this also often leads to problems. For instance, what the author of the proof might consider a technical detail could be crucial for the reader's understanding of the proof. Steps which the author might consider trivial or intuitive due to their deep understanding of the topic may be

puzzling for the reader. Without a more detailed description to fall back on, the reader may not be able to follow if the deduction gaps are too large. An even bigger problem, however, is that sometimes, the author of a proof may consider a statement trivial or intuitive which does, in fact, not hold. The reason for this is often that the intuitive understanding of the involved concepts fails to consider pathological cases like an automaton that is not connected or a graph without any vertices, since one usually only thinks about the typical case. This way, errors in the proof may go unnoticed since the proof step in which they occur was simply left out.

4.2 Formal Proofs

Conceptually, formal proofs are syntactic derivations of formulae using the axioms and inference rules of some formal system. This way, it is precisely specified which proof steps are valid and which are not. Thus, formal proofs cannot leave out any details or resort to intuitive reasoning.

This causes formal proofs to have certain advantages over informal proofs. Firstly, they are easy to follow, since there cannot be any gaps in the reasoning. While the whole proof is often lengthy, each step is immediately justified since it is just the instantiation of some inference rule. Of course, this also makes for a simple way of checking the validity of a formal proof. One simply needs to check if each step is a valid application of some inference rule. Another advantage is that being forced to formalize concepts without omitting details or using intuitive reasoning often leads to new insights and a better understanding of the concepts at hand. For instance, one might uncover additional proof obligations or pathological cases that were concealed by informal definitions or proofs. There have also been cases where formalization has led to simplification, generalization, and/or further abstraction of the involved concepts.

Unfortunately, formal proofs have their own set of issues. Most importantly, constructing them takes a lot of effort, since every little detail needs to be taken care of, often leading to lengthy and tedious proofs. This is especially true when verifying software, where the proofs tend to be full of technical details and ridden with many instances of the same trivial proof obligation. Another issue is that while each step of the proof is simple enough to comprehend, the high-level reasoning behind the proof is often obscured by technical details. This may make it a lot harder to see the idea that lies behind the proof. Finally, even though checking the correctness of some formal proof is simple in theory, doing so by hand is still very error prone due to the length of most formal proofs. Thus, errors in the proof may still not be caught, albeit for different reasons than with informal proofs.

4.3 Proof Assistants

A proof assistant is a piece of software that allows the user to interactively construct formal mathematical models and proofs. The proof assistant both helps with the construction and ensures the correctness of these proofs at all times.

Using a proof assistant can alleviate some of the issues of formal proofs described in the previous section. For instance, most proof assistants enable the user to modularize their formalization. Some of them also feature powerful automation tools enabling the proof assistant to find less complex proofs automatically. Both of these features help with either removing technical details altogether, or at least allowing the user to hide them in separate modules. This helps with keeping the focus on high-level reasoning and the idea behind the proof, while still benefiting from the fact that a formal proof is maintained by the proof assistant in the background. This is important since the most significant advantage of using a proof assistant lies in the fact that all the proofs are automatically checked. This makes the proofs much more trustworthy than informal proofs or manually checked formal proofs.

That being said, developing a formal proof with the help of a proof assistant is still significantly more work than developing the same proof informally. This is because the proof steps that are skipped in informal proofs are often quite large, making it hard for the proof assistant to fill in the gaps. Thus, initial proof development for conjectured statements is usually done informally. Once an informal proof has been found, it can be used to guide the construction of the formal proof.

```
theorem cantor:
  fixes f :: "'a ⇒ 'a set"
  shows "¬ surj f"
proof
  assume "surj f"
  then obtain x where "f x = {y. y ∉ f y}" by force
  then have "x ∈ f x ⟷ x ∈ {y. y ∉ f y}" by blast
  also have "x ∈ {y. y ∉ f y} ⟷ x ∉ f x" by blast
  finally have "x ∈ f x ⟷ x ∉ f x" by this
  then show "False" by simp
qed
```

Figure 4.1: Example of an Isabelle proof, showing a formalization of Cantor's theorem in Isabelle/HOL. Note how through the use of the structured proof language Isar, the formal proof is actually fairly readable without any knowledge of the language of the proof assistant, as it is similar to informal mathematical notation.

For this thesis, the proof assistant Isabelle [PNW; NPW02] was used together with the Higher-Order Logic (HOL) system, together called Isabelle/HOL. Isabelle's architecture is built in such a way that the correctness of the whole proof assistant depends only on a small logical core. The idea is for this core to be rarely modified but tested extensively over time, which justifies the trust in its correctness. Figure 4.1 shows an example of a formal proof in Isabelle/HOL.

4.4 Formally Verified Model Checking

A formally verified model checker has already been developed as part of the CAVA project [NEN; Esp+13; Esp+14]. As it was also developed using Isabelle/HOL, we were able to use parts of this formalization to develop our theories on partial order reduction. Namely, we use the library for finite automata [Lam14], and the formalization of the conversion from LTL formulae to generalized Büchi automata [SL14] that is needed for the formula automaton as described in section 2.3.1.

The formal theories developed as part of this thesis are still much more detailed than what is adequate for the presentation in a thesis. Furthermore, formal theories in general cannot convey abstract ideas and intuition behind concepts and proofs as well as natural language and diagrams can. Because of this, we chose not to present the formal theories in this thesis. Instead, we will present the abstract ideas and the intuition on which they are built, as well as the insights that were gained during the formalization effort. The formal theories then serve as reference material and correctness certificate, an important role in itself.

Chapter 5

Automaton Generation

In this chapter, we start by establishing some prerequisites for the formalization of algorithms like depth-first search. We then present a generalized version of depth-first search that is needed to accommodate for the search stack parameter of the ample function. Finally, we introduce the concept of generated Büchi automata in order to be able to reason about the reduced system automaton in the correctness proofs for the partial order reduction algorithm.

5.1 Nondeterminism and Refinement

When performing depth-first search, the traversal may take different routes, depending on the order in which the successors at each vertex are explored. Since the successors of a vertex are usually not ordered, the exploration order may depend on implementation details of the specific data structures that are used for the implementation. As it is not desirable for the correctness proof to be dependent on the specific implementation, we employ a technique called refinement using the nondeterminism monad [Lam12].

When using this technique, the algorithm is at first specified using abstract data structures like sets, which are assumed to be unordered. Therefore, an operation that retrieves a member of the set behaves nondeterministically. In our case, for example, the specification of the depth-first search algorithm does not make any statement about the order in which the successor vertices are explored.

In the abstract correctness proof, it is then proven that some property holds for all possible sequences of nondeterministic choices. That is, it is proven that the exploration order does not have any effect on the algorithm's result. For instance,

the fact that depth-first search discovers all vertices which are reachable through the given successor function is independent of the exploration order.

Then, the abstract version of the algorithm can be made executable with minimal effort. This is because replacing the abstract data structures with concrete ones does not affect the correctness proof, as the algorithm was proven correct for all possible sequences of nondeterministic choices. This way, the results carry over to the executable version of the algorithm. This step is called refinement.

Refinement allows to separate the proof showing that the algorithm itself is correct from the proof showing that the refinement to executable data structures is correct. This improves readability and maintainability of both the definitions and the proofs. For instance, it is possible to replace the specific implementation of an abstract data structure without having to modify the abstract correctness proof at all.

Since model checking absolutely requires the involved algorithms to be executable, we can make good use of these refinement techniques. We thus use the refinement framework implemented in [Lam12] to define our algorithms.

5.2 Generalized Depth-First Search

Regular depth-first search algorithms take as inputs a start vertex and a successor function, where the successor function takes as parameter a vertex and yields the set of edges from this vertex. As the ample function introduced in section 3.3 needs the search stack of the depth-first search as an additional parameter, it cannot be used as a regular successor function in regular depth-first search. To accommodate for this, we have to implement a generalized depth-first search algorithm, which passes the current search stack as a parameter to the successor function when evaluating it at each vertex. We call such a combination of start vertex and generalized successor function a generator for generalized depth-first search, which serves as input to the generalized depth-first search algorithm.

The generalized depth-first search algorithm has two operating modes. The first one simply explores the graph specified by the given generator by performing depth-first search and keeping track of all the discovered vertices and edges. The result is then the explored graph. Given a generator, this mode can be used to obtain the graph which the generator represents. The other mode uses user-supplied hooks that are invoked when entering or leaving a vertex. Together with a user-defined state type, this enables implementing algorithms such as nested depth-first search [Cou+92; SE05] using this generic algorithm template for generalized depth-first search.

Unfortunately, the correctness properties of this algorithm are not easily stated in an abstract way. For instance, with regular depth-first search, given a generator consisting of a start vertex v_0 and a successor function s, the graph (V, E) resulting from depth-first search exploration is easily defined inductively as follows:

$$v_0 \in V \tag{5.1a}$$

$$v \in V \implies (v, u) \in s(v) \implies u \in V \tag{5.1b}$$

$$v \in V \implies e \in s(v) \implies e \in E \tag{5.1c}$$

However, the addition of the search stack as a parameter of the successor function makes this impossible. It is necessary to actually perform the search to find out with which search stack the successor function is invoked at each node. Thus, all the properties about the explored graph have to be proven from the generalized depth-first search algorithm directly.

At this point, it may seem tempting to skip the step of defining a complete correctness property for the algorithm, and to simply use the most abstract version of algorithm itself, implemented using the constructs of the refinement framework for that purpose. However, this causes issues later on as the refinement framework was not designed to be used this way. Thus, we have to actually recreate the generalized depth-first search algorithm using inductive predicates and then use the refinement framework to prove that the most abstract version of the algorithm behaves as specified by those predicates. The predicates are then used to prove various statements about the generalized depth-first search algorithm and the automata that it explores.

5.3 Generated Büchi Automata

For the correctness proof of the partial order reduction algorithm, we want to reason about properties of the reduced system automaton. Unfortunately, this automaton is only defined implicitly using the ample function and generalized depth-first search. To make matters even worse, when using on-the-fly model checking, it may never actually get constructed in its entirety.

Thus, we make this notion of a reduced system automaton explicit by introducing the concept of Büchi automaton generators. Basically, we take the definition of regular Büchi automata and replace the transition function δ with the generalized transition function δ'. The generalized transition function takes as parameters the search stack and a state, and returns a set of transitions, each represented using a tuple consisting of the label and target state of the transition. We can then derive a generalized depth-first search generator from a Büchi automaton generator, and use

generalized depth-first search to explore the automaton's graph. From this graph, we can then construct a regular Büchi automaton, which we will call the generated automaton.

Note that since generalized depth-first search is nondeterministic, a Büchi automaton generator may generate many different automata. Thus, we cannot assign a single language to some Büchi automaton generator. Because of this, it makes sense to consider properties that hold for the languages of all the generated Büchi automata of some generator.

One further complication is that since generalized depth-first search needs a single start vertex, we need to adjust Büchi automaton generators with more than one initial state to fit this requirement. Fortunately, this is always possible by adding a special initial state with transitions to all the initial states of the original automaton generator.

With this setup, it is now possible to use the ample function together with the initial and accepting states of the system automaton to define a Büchi automaton generator for the reduced system automaton. We can then make statements about the languages of the Büchi automata generated from this generator.

Chapter 6

Auxiliary Definitions

So far, we have only defined various algorithms for performing partial order reduction. In order to reason about the correctness of these algorithms, a few auxiliary definitions are needed, which are introduced in this chapter.

6.1 Stutter Invariance

An important predicate on temporal properties is stutter invariance. Intuitively speaking, a property is stutter-invariant if it cannot distinguish between sequences that differ only with respect to character replication, also called stuttering. We call pairs of such sequences stutter-equivalent. For instance, the sequences $ddddcba$ and $dccccbaa$ are stutter-equivalent.

Stutter-invariant temporal properties are important in parallel systems since they allow different parts of the system to execute more independently from each other. For instance, consider a system where the correct functioning of subsystem A depends on subsystem B responding to its queries exactly 17 execution cycles after they were placed. Here, subsystem B cannot be replaced by a faster version later on, since that will break subsystem A. If the correctness of subsystem B were specified using a stutter-invariant property, these strict timing assumptions could not have been made by subsystem A. Stutter-invariant properties also capture the behavior of schedulers very well, since those usually do not give any guarantees about when exactly each process will be scheduled execution time. This may cause other processes to observe stuttering in some shared state.

We call an LTL formula next-free if it does not make use of the next operator. There is a connection to be observed between stutter-invariant linear temporal

properties and next-free LTL formulae. Namely, next-free LTL formulae express stutter-invariant properties, and every stutter-invariant linear temporal property can be expressed using a next-free LTL formula [PW97].

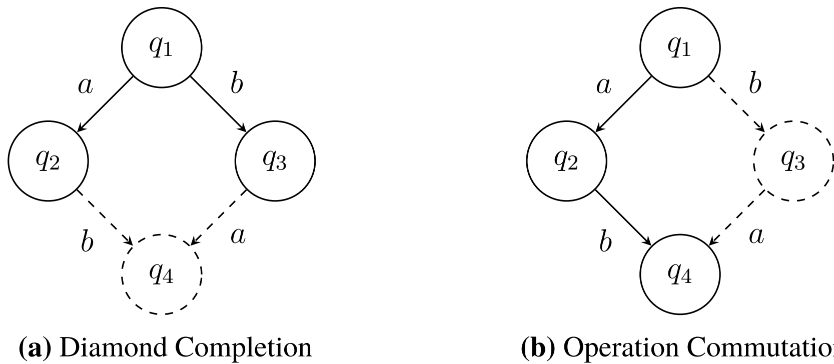
The partial order reduction algorithm considered in this thesis works with next-free LTL formulae. The fact that next-free LTL formulae express stutter-invariant properties will be used in the correctness proof in section 7.5.

6.2 Independence Relation

One of the key concepts in partial order reduction is that of an independence relation between operations of the system [Pel96, Definition 2.1]. We say that a relation I is an independence relation if and only if it exhibits the following properties:

1. I is irreflexive
2. I is symmetric
3. if $(a, b) \in I$ and $a \in \mathrm{en}(q)$, then $b \in \mathrm{en}(q)$ if and only if $b \in \mathrm{en}(\mathrm{ex}(a, q))$
4. if $(a, b) \in I$, $a \in \mathrm{en}(q)$, and $b \in \mathrm{en}(q)$, then $\mathrm{ex}(b, \mathrm{ex}(a, q)) = \mathrm{ex}(a, \mathrm{ex}(b, q))$

Intuitively, property 3 states that independent operations may not influence each other's enabledness, that is, executing one of the operations should not enable or disable the other. Property 4 specifies that independent operations should commute, that is, the order in which they are executed should not matter. Some common consequences of two operations being independent are shown in figure 6.1.



(a) Diamond Completion **(b)** Operation Commutation

Figure 6.1: Independence gives rise to diamond structures. If the operations a and b are independent, then the solid parts of the diagrams imply the dashed parts. In diagram a, both operations are enabled at state q_1 and executing either of them cannot disable the other. Since the operations also commute, the paths are guaranteed to join at state q_4. In diagram b, operation b being enabled at state q_2 implies that it is also enabled at state q_1, and thus the diagram can be completed as shown.

A dependence relation is the complement of an independence relation. We will often consider the independence relation I together with its complement D.

Given an independence relation, the notion of independence can easily be extended to sets of operations. We define two sets to be independent if and only if all pairs of operations in the Cartesian product of these sets are independent. Using the notion of independence between sets of operations, we can observe what happens when the operations in a sequence are independent of the operations in another sequence. Some of these consequences are illustrated in figure 6.2.

(a) Finite Sequence – Finite Sequence **(b)** Infinite Sequence – Finite Sequence

Figure 6.2: Independence properties carry over to sequences of operations. If the operations in w are independent of those in v, then the solid parts of the diagrams imply the dashed parts. Diagram a is essentially equivalent to diagram 6.1a, with single operations being replaced by finite sequences of operations. Diagram b illustrates the fact that the completion even works when the sequence w is infinite.

6.3 Equivalence Relations

Given an independence relation I and its corresponding dependency relation D, we define various equivalence relations between finite and infinite sequences of operations [Pel96, Page 41].

We say that two finite sequences w_1 and w_2 are equivalent, written as $w_1 \equiv_D w_2$, if and only if they can be transformed into one another by repeatedly commuting adjacent independent operations. The idea here is that two independent operations can always be executed in reverse order, as guaranteed by property 3 from the definition of independence relations. Furthermore, property 4 from the definition of independence relations states that the order in which two independent operations are executed does not matter, as the state that is reached in the end is the same either way. As these properties still hold true for repeated commutations of adjacent independent operations, it is usually possible to substitute a sequence of operations with a different sequence that is equivalent to the first one.

Next, we define the notion of prefix equivalence between two sequences w_1 and w_2, written as $w_1 \preceq_D w_2$. We give definitions for all sensible combinations of finite and infinite sequences, with the definitions successively building upon each other. Given finite sequences w_1 and w_2, the definition is as follows:

$$w_1 \preceq_D w_2 \iff \exists\, v_1.\ w_1 \cdot v_1 \equiv_D w_2 \tag{6.1}$$

Given a finite sequence w_1 and an infinite sequence w_2, the definition is as follows:

$$w_1 \preceq_D w_2 \iff \exists\, v_2 \sqsubseteq w_2.\ w_1 \preceq_D v_2 \tag{6.2}$$

Given infinite sequences w_1 and w_2, the definition is as follows:

$$w_1 \preceq_D w_2 \iff \forall\, v_1 \sqsubseteq w_1.\ v_1 \preceq_D w_2 \tag{6.3}$$

Intuitively, w_1 is prefix equivalent to w_2 if and only if w_1 is the prefix of a sequence that is equivalent to w_2. Note however, that this intuition may not hold in the case of infinite sequences, where the actual definitions are a little more complicated.

Having established the notion of prefix equivalence, it can now be used to extend the definition of equivalence to infinite sequences w_1 and w_2 as follows:

$$w_1 \equiv_D w_2 \iff w_1 \preceq_D w_2 \wedge w_2 \preceq_D w_1 \tag{6.4}$$

That is, an infinite sequence is equivalent to another infinite sequence if and only if both sequences are prefix equivalent to each other. This definition is possible since the notion of prefix equivalence was also defined on pairs of infinite sequences. It may also be worth noting that equation 6.4 constitutes a valid theorem when w_1 and w_2 are finite sequences, in which case the notion of prefix equivalence defined on pairs of finite sequences is invoked by the identity.

We define one last equivalence relation for both finite and infinite sequences of operations. Given a set of operations A, as well as sequences w_1 and w_2, where either both are finite or both are infinite, we write $w_1 \preceq_D^A w_2$ if and only if there exists a selection function s, such that the following statements hold:

1. $w_1 \equiv_D s(w_2)$
2. the operations in $\bar{s}(w_2)$ are a subset of A
3. s is an independent decomposition of w_2

The notion of independent decomposition will be introduced in section 6.6. Intuitively, $w_1 \preceq_D^A w_2$ means that w_1 is equivalent to a selection of w_2, where only operations in A have been removed, and where the removed operations can be deferred indefinitely in w_2.

6.4 Visibility Sets

As mentioned in the previous section, replacing a sequence of operations with an equivalent one usually has no adverse effects as it does not change the final state that is reached after executing the whole sequence. However, the correctness property may still be sensitive to the order in which certain operations are executed within the sequence. If this is the case, a sequence of operations cannot simply be replaced by an equivalent one as that could cause a change in whether the sequence is accepted by the formula or not.

To deal with this kind of situation, we define the notion of a visibility set [Pel96, Definition 3.4]. A set V is called a visibility set if and only if the following holds for all system states q_1 and q_2, and for all operations a:

$$(q_1, a, q_2) \in \Delta(S) \implies a \notin V \implies P(q_1) = P(q_2) \tag{6.5}$$

That is, for every transition in the system automaton whose associated operation is invisible, the source state and the target state of the transition have the same interpretation.

With this, a visibility set is a conservative approximation of the set of those operations whose execution can cause a change in the interpretation of the states along some run in the system automaton. Given that the order of the visible operations within a run is preserved, all the other operations can be commuted freely without causing changes in the interpretation of the run, and therefore, also without causing changes in the acceptance of this run by the formula.

Furthermore, the execution of invisible operations is merely observed as additional stuttering when considering interpreted runs. As mentioned in section 6.1, the partial order algorithm that is covered in this thesis works with formulae that represent stutter-invariant properties. Thus, the execution of additional operations that are invisible cannot be detected by the formula.

6.5 Independent Occurrence

We introduce the notion of independent occurrence. Given a set of operations A and a finite or infinite sequence w, we say that A is an independent occurrence set for w, if and only if all the operations in w before the first occurrence of an operation in A are independent of the operations in A. In the case that no operation in A ever occurs in w, all the operations in w have to be independent of the operations in A.

The idea behind this definition is explained in the following. If A is an independent occurrence set for w, then there are two cases. First, we consider the case that some operation in A occurs in w. In this case, we can call this operation a and split the sequence w into the three parts w_1, a, and w_2. Since A is an independent occurrence set for w, we know that the operation a, being the first occurrence of an operation in A, is independent of all the operations in w_2, which yields the following equality:

$$w = w_1 \cdot a \cdot w_2 \equiv_D a \cdot w_1 \cdot w_2 \tag{6.6}$$

In the other case, no operation in A ever occurs in w. Since A is an independent occurrence set for w, this means that all the operations in w have to be independent of the operations in A. This results in the following equivalence:

$$a \cdot w \equiv_D w \cdot a \tag{6.7}$$

6.6 Independent Decomposition

Independent decomposition is a property of a selection function s with respect to a finite or infinite sequence of operations w. We say that s is an independent decomposition of w, if and only if every operation in w that is not selected by s is independent of all the operations in w occurring after it that are selected by s.

Essentially, the statement that s is an independent decomposition of w means that the deselected occurrences of operations can be deferred indefinitely. In the case of finite sequences, this means that the deselected operations can be commuted to the end of the sequence, resulting in the following decomposition of w:

$$w \equiv_D s(w) \cdot \bar{s}(w) \tag{6.8}$$

It is also possible to do the reverse and construct a function that is an independent decomposition from an arbitrary decomposition of a sequence. However, the definition is even more useful in the case of infinite sequences, as this case cannot be characterized using a decomposition. Here, it means that the deselected operations can be commuted as far back as needed, or, equivalently, arbitrarily many selected operations can be made to occur before the first deselected operation.

Chapter 7

Verification of Off-Line Partial Order Reduction

The proof presented in [Pel96, Section 4] reduces the correctness of on-the-fly partial order reduction to that of off-line partial order reduction. Thus, our first formal verification effort concerns off-line partial order reduction. In the following sections, we show that the off-line partial order reduction algorithm presented in section 3.3 is correct, given that the ample function fulfills certain properties. To do so, we adopt the proof architecture from [Pel96, Section 3.2]. It consists of proving a series of theorems which build upon each other, culminating in the final correctness property.

Throughout this chapter, we will assume the existence of a system automaton S, a formula φ, an interpretation function P, and a reduced system automaton R, where the reduced system automaton was generated using the algorithm described in section 3.3.

7.1 Assumptions

In order to prove the correctness of the partial order reduction algorithm presented in section 3.3, we need to be able to assume certain properties about the ample function that is used in this algorithm. Usually, these properties would be obtained from theorems about the static analysis phase of the partial order reduction algorithm. However, as mentioned in section 3.2, the static analysis phase will not be covered in this thesis. Thus, in this section, we will simply state the properties that we expect the static analysis phase to guarantee.

These properties are stated with respect to an independence relation I as defined in section 6.2. We will call the corresponding dependence relation D. We also assume that there is a visibility set V as defined in section 6.4. As stated in section 3.1, we will consider the case of partial order reduction without any fairness assumptions. We thus use the following conditions adapted from [Pel96, Section 3]:

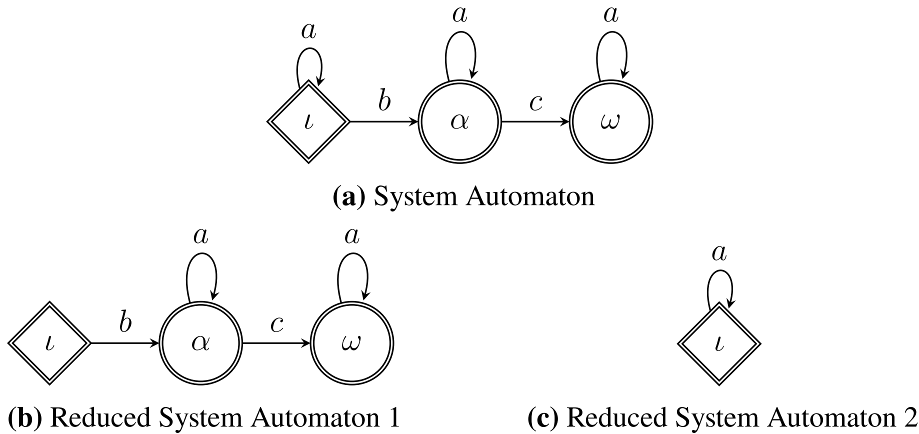
C1 If v is the word corresponding to a run starting at state q, then $\mathrm{ample}(s, q)$ is an independent occurrence set for v.

C2 If $\mathrm{ample}(s, q)$ is a proper subset of $\mathrm{en}(q)$, then for all operations a in $\mathrm{ample}(s, q)$, it holds that $\mathrm{ex}(a, q) \notin s$.

C3′ If $\mathrm{ample}(s, q)$ is a proper subset of $\mathrm{en}(q)$, then all of the operations in $\mathrm{ample}(s, q)$ are invisible.

Note that we present the conditions a little differently than is done in [Pel96], taking advantage of having defined the notion of independent occurrence in section 6.5, as well as making use of the additional search stack parameter s. In the case that $\mathrm{ample}(s, q) = \mathrm{en}(q)$, these conditions are fulfilled trivially.

The purpose of condition C1 is as follows. If v is the word corresponding to a run starting at state q, then $\mathrm{ample}(s, q)$ may omit some of the operations enabled at state q. However, condition C1 guarantees that all of the operations that occur in v before the first operation in $\mathrm{ample}(s, q)$ are independent of the operations in $\mathrm{ample}(s, q)$. Thus, the system might as well execute one of the operations in $\mathrm{ample}(s, q)$ before these operations, justifying the reduction.



(a) System Automaton

(b) Reduced System Automaton 1

(c) Reduced System Automaton 2

Figure 7.1: Example of partial order reduction failing without condition C2. Shown are the system automaton and its two possible reductions. Condition C2 prevents degenerate reductions like the one seen in diagram c, which would wrongly alter the result of the model checking process.

In order to fulfill condition C2, the ample function has to take the search stack as an additional parameter. Condition C2 is thus indirectly responsible for a lot of problems that arise in both the algorithm and its correctness proof due to this additional parameter. However, as illustrated in figure 7.1, it is needed for the algorithm to be correct. The system automaton presented in figure 7.1a contains the operations a, b, and c, where a is independent of both b and c. We assume that the operation c is visible. We consider the reduction at state ι. Without condition C2, either operation a or operation b can be omitted. Figure 7.1b shows the effect of omitting operation a. In this case, the reduction does not adversely affect the model checking process, and operation c is still included. However, the reduction shown in figure 7.1c, which omits operation b, causes the automaton to degenerate completely, including the omission of operation c, which is visible. Condition C2 ensures that this cannot happen by only allowing a reduction to take place if none of the operations in the resulting ample set are on the search stack. In this example, the omission of operation b is thus not a valid reduction.

Condition C3$'$ requires all the operations in a reduced ample set to be invisible. The idea here is that since visible operations can have an effect on the checked property, they cannot be commuted, as that may incorrectly alter the outcome of the model checking process. Condition C3$'$ effectively augments the dependency relation D, adding dependencies between all visible operations. We thus introduce a new dependency relation D' which represents this [Pel96, Page 50]:

$$D' = D \cup (V \times V) \tag{7.1}$$

This dependency relation makes the augmentation of the original dependency relation that is caused by condition C3$'$ explicit. We will use D' as the primary dependency relation in the remainder of the off-line correctness proof. Thus, when we mention that two operations are independent, we mean independent with respect to D', unless explicitly stated otherwise.

7.2 Lemma 3.7

Let there be an infinite transition sequence that is accepted by the system automaton with a corresponding transition run r starting at state q and a corresponding transition sequence v. Then, lemma 3.7 [Pel96, Page 50] states that one of the following statements holds:

a) there exists an operation a in $\mathrm{ample}(s, q)$, such that $a \preceq_{D'} v$
b) the operations in $\mathrm{ample}(s, q)$ are invisible and independent of those in v

For the proof, we first consider the case that $\mathrm{ample}(s, q) = \mathrm{en}(q)$. Let a be the first operation in v. Since a has to be in $\mathrm{en}(q)$, it is also in $\mathrm{ample}(s, q)$. Since a is the first operation in v, it also holds that $a \preceq_{D'} v$, proving case a of theorem 3.7.

We now assume that $\mathrm{ample}(s, q) \neq \mathrm{en}(q)$ holds. We know from condition C1 that $\mathrm{ample}(s, q)$ is an independent occurrence set for v. Furthermore, condition C3′ implies that the operations in $\mathrm{ample}(s, q)$ are invisible as we are considering the case that a reduction takes place. We consider two subcases. If none of the operations in $\mathrm{ample}(s, q)$ ever occur in v, we know that they are both invisible and independent of the operations in v, proving case b of theorem 3.7. If, on the other hand, some operations in $\mathrm{ample}(s, q)$ do occur in v, we can assume a to be the first of them and then decompose v as outlined in equation 6.6:

$$v = v_1 \cdot a \cdot v_2 \equiv_D a \cdot v_1 \cdot v_2 \tag{7.2}$$

Since a is invisible, this still holds when replacing D with D'. We can thus conclude that $a \preceq_{D'} v$, completing the proof of case a of theorem 3.7.

7.3 Theorem 3.9

Given an infinite word $v_1 \cdot a \cdot v_2$ accepted by the system automaton, theorem 3.9 [Pel96, Page 51] states that there exist sequences u, t, and w, such that the following statements hold:

1. $u \cdot a$ extends v_1 in the reduced system automaton
2. the operations in u are independent of a
3. the operations in u are invisible
4. t is an independent decomposition of u
5. $t(u) \cdot w \equiv_{D'} v_2$
6. the operations in $\bar{t}(u)$ are independent of the operations in w

Figure 7.2 illustrates theorem 3.9. Intuitively, the statement of theorem 3.9 is the following. If an operation a is executed as part of a word $v_1 \cdot a \cdot v_2$ in the system automaton, then the reduced system automaton may not be able to execute the operation a at this point of the word due to this operation having been omitted during the reduction. However, the reduced system automaton can execute a sequence of operations $u \cdot a$ instead. The sequence u is decomposed into two parts by the selection function t. The operations in $t(u)$ will be executed as part of the word v_2 later (consequence 5), while the operations in $\bar{t}(u)$ may never occur in the word v_2. However, the operations in $\bar{t}(u)$ are independent of the operations in w (consequence 6), which represents the part of v_2 that is not covered by $t(u)$.

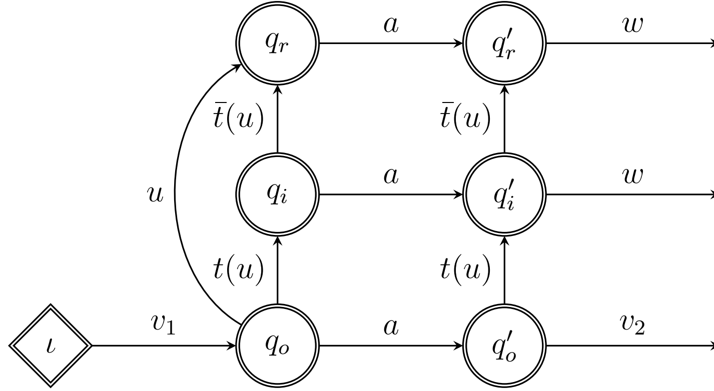


Figure 7.2: Illustration for theorem 3.9. The word $v_1 \cdot a \cdot v_2$ is given. Theorem 3.9 guarantees the existence of a word $u \cdot a$ in the reduced system automaton, which extends the word v_1. Since a is independent of the operations in u (and thus also the operations in $t(u)$ and $\bar{t}(u)$), various different, equivalent paths can be taken, leading to the grid-like structure seen in the diagram.

Due to time constraints, theorem 3.9 could not be formally proven in this thesis. In [Pel96, Page 51], the proof of theorem 3.9 is omitted with the justification being that it is similar to the proof of theorem 3.2, which is the corresponding theorem for partial order reduction with fairness assumptions. The proof of theorem 3.2 is given in [Pel96, Page 45] and uses induction on the order in which the depth-first search algorithm closes the states it discovers in the reduced system automaton.

7.4 Theorem 3.11

The central theorem of off-line partial order reduction is theorem 3.11 [Pel96, Page 52]. It makes the following statements:

1. if a transition sequence is accepted by the reduced system automaton, then it is also accepted by the system automaton
2. if v is a transition word accepted by the system automaton, there exists a transition word w that is accepted by the reduced system automaton and a set of invisible operations A, such that $v \preceq_{D'}^{A} w$

Consequence 1 can be proven fairly easily. Since all partial order reduction does is restrict the set of enabled operations at some states, both the set of states and the transition relation of the reduced system automaton are subsets of the corresponding entities of the system automaton. Thus, according to the definition of accepting

transition sequences given in section A.3, any transition sequence that is accepted by the reduced system automaton is also accepted by the system automaton.

The proof of consequence 2, however, is rather tricky. We formalize the informal proof given in [Pel96, Page 52]. The first step is, given a word accepted by the system automaton, to inductively construct finite prefixes of a word accepted by the reduced system automaton. For every operation executed in the original word, the construction scheme takes a step in building the word for the reduced system automaton. We consider three aspects of this construction. Firstly, section 7.4.1 describes the construction scheme that specifies in which state the construction starts, as well as in which direction it can take a step in any given state. Secondly, section 7.4.2 proves certain invariants that hold throughout the construction and which will be needed later in the proof. Thirdly and lastly, since we want to construct arbitrarily long prefixes of the word for the reduced system automaton, section 7.4.3 proves that at any point of the construction, it is possible to take another step. The second step in proving consequence 2 of theorem 3.11 consists of making the transition from the arbitrarily long but finite prefixes that were constructed in the first step, to the final, infinite word accepted by the reduced system automaton. This step is described in section 7.4.4. The third and final step described in section 7.4.5 draws the theorem's conclusions concerning the infinite word established in the second step, whose prefixes are known to exhibit properties established in the last part of the first step.

The following sections will assume the existence of a word v that is accepted by the system automaton and describe the construction of a word w that is accepted by the reduced system automaton.

7.4.1 Construction Scheme

Formally, the construction scheme for theorem 3.11 is represented using an inductive predicate [NPW02, Section 7] called C. Basically, this predicate describes the valid states of the construction by specifying an initial state as well as rules for taking steps. We write $C(v_1, v_2, w_1, w_2, s_1, l)$ to state that a construction state is valid. The meaning of the involved variables is described in the following:

v_1	the prefix of v that has been processed so far
v_2	the suffix of v that has not yet been processed
w_1	the prefix of the reduced word constructed so far
w_2	one possible suffix of the reduced word constructed so far
s_1	the selection function for operations that appear in w_1 but not in v
l	the operations that have been appended to w_1 but did not appear in v_1

The predicate itself is described using the following rules:

initial $C(\epsilon, v, \epsilon, v, \epsilon, \epsilon)$ holds

absorb $C(v_1 \cdot a, v_2, w_1, w_2, s_1, l')$ holds, given that

1. $C(v_1, a \cdot v_2, w_1, w_2, s_1, l)$ holds
2. a does occur in l
3. l' is l with the first occurrence of a removed

extend $C(v_1 \cdot a, v_2, w_1 \cdot u \cdot a, w_2', s_1 \cdot t \cdot \top, l \cdot t(u))$ holds, given that

1. $C(v_1, a \cdot v_2, w_1, w_2, s_1, l)$ holds
2. a does not occur in l
3. $u \cdot a$ extends w_1 in the reduced system automaton
4. the operations in u are independent of a
5. the operations in u are invisible
6. t is an independent decomposition of u
7. $a \cdot t(u) \cdot w_2' \equiv_{D'} w_2$
8. the operations in $\bar{t}(u)$ are independent of the operations in w_2'

To facilitate proper understanding of this predicate, the idea behind the construction scheme is explained. Starting with the initial rule, operations are read from v_2 and appended to v_1 using the absorb and extend rules ($\ldots v_1, a \cdot v_2 \ldots$ becomes $\ldots v_1 \cdot a, v_2 \ldots$). At the same time, operations are appended to w_1, constructing the reduced word, while a possible suffix of the reduced word is kept in w_2. Since v_1 will never be empty again after the first step, the initial rule can only be applied once, at the very beginning. Given that the absorb rule can only be applied if a does occur in l, while for the extend rule, the opposite is true, we already know that at all points of the construction, at most one of these rules can be applied.

The extend rule extends the reduced word that has been constructed so far. It uses theorem 3.9, which does the extensions in batches, possibly appending multiple operations to w_1 that will either appear in v_2 later on or not at all. The absorb rule takes care of processing those operations that have been appended to w_1 ahead of time, with l keeping track of which operations are still queued up. Once an operation is encountered that is not in the queue, the extend rule is applied once again, appending more operations to both w_1 and the queue l. The selection function s keeps track of those operations that will never appear in v_2. This way, the absorb and extend rules ensure steady progress, building arbitrarily long words w_2 in the reduced system automaton.

It is worth noting that there are multiple ways of specifying the behavior of the construction scheme using inductive predicates. Since the predicate is merely a tool used for giving structure to the proof, we only care about the invariants it entails and about the steps it can take. This allows for some flexibility in defining the predicate.

For instance, in section 7.4.2, we will prove invariant 1, which states that at all points of the construction, w_1 is a transition word in the reduced system automaton. This can be proven inductively, since premise 3 of the extend rule states that the sequence $u \cdot a$, which is appended to w_1 when applying the extend rule, is also a transition word in the reduced system automaton which extends w_1. This causes the whole sequence $w_1 \cdot u \cdot a$ to be a transition word in the reduced system automaton.

Alternatively, we could have required the composite sequence $w_1 \cdot u \cdot a$ to be a transition word in the reduced system automaton in premise 3. This would have shifted the proof obligation for the fact that the concatenation of the sequences w_1 and $u \cdot a$ is a transition word in the reduced system automaton from the proofs of invariants of the construction in section 7.4.2 to the proof about the construction scheme always being able to take a step in section 7.4.3.

Intuitively speaking, we have the choice between a weak predicate, leading to difficult invariant proofs and simple step proofs, and a strong predicate, leading to simple invariant proofs and hard step proofs. We chose to make the predicate as weak as possible. Firstly, this simplifies the statement of the predicate a little. Secondly, it shifts most of the work to the invariant proofs, which are easier to structure and more easily decomposed into multiple lemmata, while the step proof can remain as simple as possible.

7.4.2 Construction Invariants

In order to be able to work with the construction scheme, we first need to prove certain invariants about the construction which will be needed for the later parts of the proof. Given that $C(v_1, v_2, w_1, w_2, s_1, l)$ holds, we state the following invariants:

1. w_1 is a transition word in the reduced system automaton
2. the first operation of v_2 is an independent occurrence in l
3. s_1 is an independent decomposition of w_1
4. $s_1(w_1) \equiv_{D'} v_1 \cdot l$
5. $l \cdot w_2 \equiv_{D'} v_2$
6. $s_1(w_1) \cdot w_2 \equiv_{D'} v_1 \cdot v_2$
7. the operations in $\overline{s_1}(w_1)$ are independent of the operations in w_2
8. the operations in $\overline{s_1}(w_1)$ are invisible

Figure 7.3 illustrates some of these invariants. Note that invariant 2 is never actually used in the proof of theorem 3.11, serving merely as an intermediate property needed to inductively prove invariants 4 and 5. Also note that invariant 6 follows immediately from invariants 4 and 5. The rest of the invariants are used at various points of the proof.

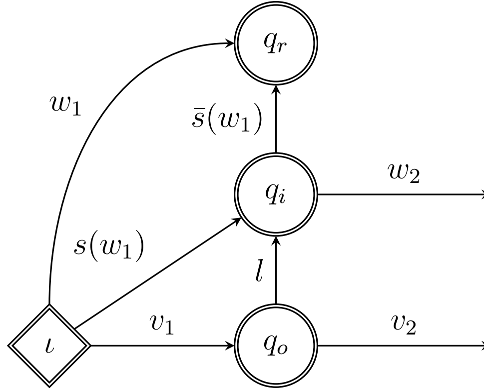


Figure 7.3: Illustration for the Construction state $(v_1, v_2, w_1, w_2, s_1, l)$. Some of the invariants can be seen in the diagram. For instance, the equivalence $s_1(w_1) \equiv_{D'} v_1 \cdot l$ from invariant 4 and the equivalence $w_1 \equiv_{D'} s_1(w_1) \cdot \overline{s_1}(w_1)$ from invariant 3 lead to the triangles visible in the diagram. The equivalence $l \cdot w_2 \equiv_{D'} v_2$ from invariant 5 is less visible, since the runs associated with the words v_2 and w_2 may never meet, continuing indefinitely, even though they can be considered equivalent.

The proofs of these invariants are done using rule induction [NPW02, Section 7.1.3] on the inductively defined predicate C. With this scheme, the proofs of the invariants are very straightforward. It is merely necessary to apply the composition rules of the involved concepts (transition words, independent occurrence, independent decomposition, etc.) using the induction hypotheses. Thus, we decided to not present these proofs here.

7.4.3 Construction Step

In the previous section, we established various properties that hold for valid construction states. To make these properties useful, we need to show that their premises are actually met. That is, we need to prove that the construction can produce arbitrarily many valid construction states by consuming the word v step by step. More specifically, we show that in each construction state, the construction scheme can take a step towards a successor state, consuming an operation from v in the process.

With v being infinite, we can assume the suffix of v that has not yet been processed to be of the form $a \cdot v_2$. Thus, we can assume that $C(v_1, a \cdot v_2, w_1, w_2, s_1, l)$ holds. The proof then works by case distinction on whether or not a occurs in l. Assuming that a does occur in l, applying the absorb rule is simple, since premise 3 is really just a definition of l'. Thus, by removing the first occurrence of a from l, all the premises are fulfilled and the absorb rule can be applied.

Let us now consider the case that a does not occur in l. In this case, our goal is to apply the extend rule in order to perform a step of the construction. In order to instantiate the extend rule we need to find a sequence of operations u, which, together with the operation a, forms an extension of the word w_1. To do so, we want to use theorem 3.9 from section 7.3. Thus, the first step is to prove that there is an infinite word starting with $w_1 \cdot a$. This would fulfill the premises of theorem 3.9. Figure 7.4 illustrates the steps necessary for this proof.

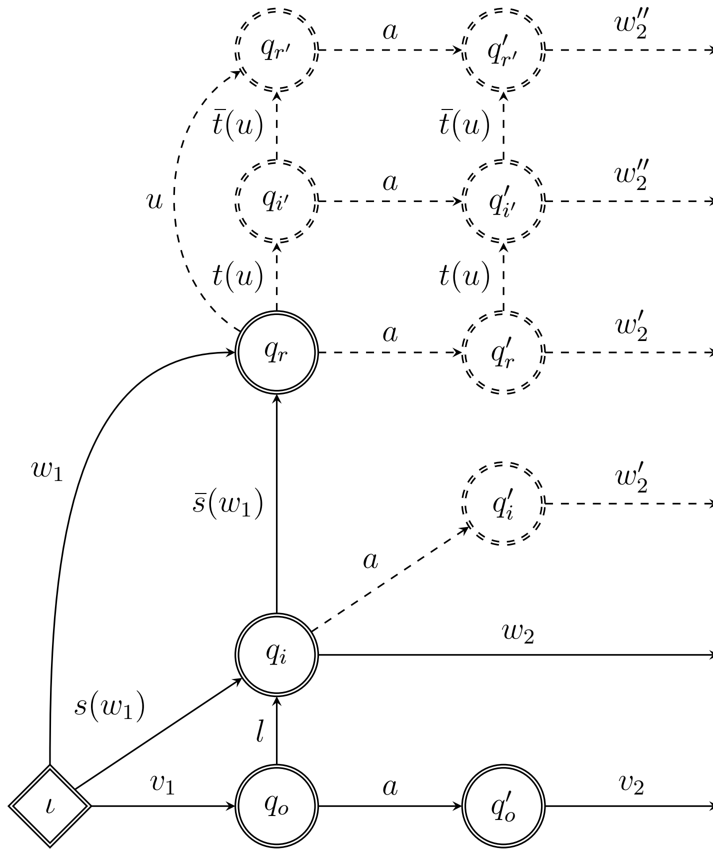


Figure 7.4: Various proof steps are performed towards applying theorem 3.9. First, w_2 is decomposed into $a \cdot w'_2$, resulting in the inclusion of the part of the automaton associated with state q'_i. Secondly, $a \cdot w'_2$ is appended to w_1, yielding the part of the automaton associated with the state q'_r. This allows us to instantiate theorem 3.9 using the run $w_1 \cdot a \cdot w'_2$, resulting in the runs u and w''_2 as well as the selection function t, displayed in the uppermost part of the diagram.

The solid parts of the automaton in figure 7.4 are identical to the automaton shown in figure 7.3, with the exception of v_2 having been replaced with $a \cdot v_2$, since a different construction state is being used.

The first step towards fulfilling the premises of theorem 3.9 is as follows. As a does not occur in l, but $l \cdot w_2 \equiv_{D'} a \cdot v_2$ due to invariant 5, we know that a must occur in w_2. Furthermore, a being at the very beginning of the sequence $a \cdot v_2$ implies $a \preceq_{D'} w_2$. Thus, we know that there exists a sequence w_2', such that $a \cdot w_2' \equiv_{D'} w_2$. This insight is indicated in figure 7.4 by adding the state q_i' together with its transitions.

Next, due to invariant 7, we know that the operations in $\overline{s_1}(w_1)$ are independent of the operations in w_2. From this, it follows that w_2, or, equivalently, $a \cdot w_2'$, can also be appended to w_1, as shown in figure 7.4 using the state q_r'.

With this, we have established an infinite word starting with $w_1 \cdot a$. This allows us to instantiate theorem 3.9 using the word $w_1 \cdot a \cdot w_2'$, yielding the uppermost part of the automaton in figure 7.4 consisting of the states $q_{i'}$, $q_{i'}'$, $q_{r'}$, and $q_{r'}'$.

We then prove that $C(v_1 \cdot a, v_2, w_1 \cdot u \cdot a, w_2'', s_1 \cdot t \cdot \top, l \cdot t(u))$ holds using the extend rule. With the exception of premise 7, all premises of the extend rule follow directly from either assumptions made in this section or from the consequences of theorem 3.9. Premise 7 requires that $a \cdot t(u) \cdot w_2'' \equiv_{D'} w_2$, which follows from our earlier statement $a \cdot w_2' \equiv_{D'} w_2$, as well as from consequence 5 of theorem 3.9, which states that $t(u) \cdot w_2'' \equiv_{D'} w_2'$.

With $C(v_1 \cdot a, v_2, w_1 \cdot u \cdot a, w_2'', s_1 \cdot t \cdot \top, l \cdot t(u))$ established, we have proven the existence of a successor state with a appended to v_1, concluding the proof for the case of a not occurring in l.

This proves that at each point of the construction, at least one of the rules can be applied, consuming an operation from v_2 in the process. Together with the observation made in section 7.4.1, stating that the absorb and the extend rules are mutually exclusive, this leads to the conclusion that at each point of the construction, exactly one of the rules can be applied.

7.4.4 Transition to Infinite Sequences

In the previous section, we have proven that at each point of the construction, the construction scheme can take a step, consuming an operation from the transition word in the system automaton in the process. While doing so, it constructs a corresponding transition word in the reduced system automaton. However, at each point of the construction, the transition word in the reduced system automaton is finite, while in order to prove theorem 3.11, it is necessary to construct an infinite transition word in the reduced system automaton.

Intuitively, it should be possible to define an infinite transition word in the reduced system automaton by taking infinitely many steps using the construction scheme.

As the construction is deterministic, it should be possible to obtain a sequence of finite transition words in the reduced system automaton, one for each successive construction state. It should then be possible to get an infinite transition word in the reduced system automaton as the limit of this sequence.

However, as is often the case with intuitive reasoning, there are additional assumptions hidden in this argument. Firstly, in order for this limit to exist at all, the words in the sequence cannot keep changing forever. For every natural number, there has to be a point in the sequence such that the prefixes of words of that length stay fixed. Luckily, this is guaranteed by the fact that with each step of the construction, operations are simply appended to the word in the reduced system automaton. This way, once an operation has been appended to this word, it never changes.

Secondly, in order for the limit to be an infinite sequence, we need to show that the words in the sequence grow to arbitrary length. Since the absorb rule does not append anything to the word in the reduced system automaton, it is not immediately obvious why this should be the case. Fortunately, we have proven invariant 4, which states that $s_1(w_1) \equiv_{D'} v_1 \cdot l$. Since finite sequences which are equivalent also have the same length, and applying a selection function to a sequence can only make it shorter, this implies the following lower bound on the length of w_1:

$$\mathrm{length}(w_1) \geq \mathrm{length}(v_1) \tag{7.3}$$

Since each step of the construction scheme appends an operation to v_1, and it can take an arbitrary number of steps, this lower bound shows that the words w_1 in the reduced system automaton grow to arbitrary length along the sequence.

This allows us to define an infinite word w and an infinite selection function s, such that for every decomposition $v = v_1 \cdot v_2$, there exist a prefix w_1 of w, a prefix s_1 of s, a sequence w_2, and a sequence l, such that $C(v_1, v_2, w_1, w_2, s_1, l)$ holds. Using the invariants about valid construction states proven in section 7.4.2, we can then derive statements about arbitrarily long prefixes of w and s.

7.4.5 Final Consequences

To conclude the proof of the second part of theorem 3.11, we need to prove that:

1. the sequence w is an infinite word in the reduced system automaton
2. there exists a set of invisible operations A, such that $v \preceq^A_{D'} w$

The proof for consequence 1 is rather straightforward. Applying the properties of w established in the previous section, we can use invariant 1 to prove that arbitrarily long prefixes of w are words in the reduced system automaton. An infinite word in

the reduced system automaton merely requires all the operations in it to be enabled at their respective states in the sequence. For any operation a occurring in w, we can thus obtain a prefix w_1 of w that includes all the operations up to and including this occurrence of a. As w_1 is a word in the reduced system automaton, this proves that this occurrence of a is enabled in the reduced system automaton, showing that w is an infinite word in the reduced system automaton.

Consequence 2 is also proven readily. The definition of the relation $\preceq_D^A$ in section 6.3 implies that in order to prove that there exists a set of invisible operations A, such that $v \preceq_{D'}^A w$, we need to prove that there exists a selection function s, such that the following statements hold:

1. $v \equiv_{D'} s(w)$
2. all the operations in $\bar{s}(w)$ are invisible
3. s is an independent decomposition of w

Statement 1 can be proven by showing that both $v \preceq_{D'} s(w)$ and $s(w) \preceq_{D'} v$ hold. Since the relation $\preceq_D$ is defined in terms of a universally quantified statement about prefixes, both statements follow from the invariants in a straightforward fashion.

From invariant 8 we can derive that, for arbitrarily long prefixes w_1 of w, the operations in $\bar{s}(w_1)$ are invisible. Thus, the same holds true for w itself, which proves statement 2.

Finally, invariant 3 asserts that, for arbitrarily long prefixes w_1 of w and s_1 of s, s_1 is an independent decomposition of w_1. From the way independent decomposition is defined, this property carries over to w and s, proving statement 3.

7.5 Theorem 3.12

Let A be a set of invisible operations and let v and w be words in the system automaton such that $v \preceq_{D'}^A w$. Let furthermore r be the run corresponding to the word v and s be the run corresponding to the word w. With these assumptions, theorem 3.12 [Pel96, Page 52] states the following:

$$P(r) \models \varphi \iff P(s) \models \varphi \tag{7.4}$$

In the context of partial order reduction and with the consequences obtained from theorem 3.11, this means that the formula accepts the interpretation of the original run if and only if it accepts the interpretation of the reduced run.

The proof of theorem 3.12 consists of two parts. The first part shows that $P(r)$ and $P(s)$ are equivalent up to stuttering. It was not possible to formally prove

this within the time frame of this thesis. A description of the proof can be found in [Pel96, Page 52]. The second part shows that next-free LTL formulae cannot distinguish between words that are equivalent up to stuttering. This property was already established in section 6.1.

7.6 Correctness Theorem

In this section, we assemble the results proven previously and use them to obtain the final correctness theorem for off-line partial order reduction.

From consequence 1 of theorem 3.11 from section 7.4 we know that every transition sequence accepted by the reduced system automaton is also accepted by the system automaton. Thus, the same is also true for the accepted transition runs. Together with the definition of the interpreted language from equation 2.2, this proves that the interpreted language of the reduced system automaton is a subset of the interpreted language of the system automaton:

$$\mathcal{L}_P(R) \subseteq \mathcal{L}_P(S) \tag{7.5}$$

Using consequence 2 of theorem 3.11 together with the consequence of theorem 3.12 from section 7.5, we know that for each transition run accepted by the system automaton, there is a potentially different transition run accepted by the reduced system automaton whose interpretation satisfies the formula if and only if the interpretation of the original run does so, too. Thus, if for all transition runs accepted by the reduced system automaton, it holds that their interpretations satisfy the formula, then the same is also true for the system automaton:

$$\mathcal{L}_P(R) \subseteq \mathcal{L}(\varphi) \implies \mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) \tag{7.6}$$

Putting together equations 7.5 and 7.6, we obtain the final correctness theorem of off-line partial order reduction:

$$\mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) \iff \mathcal{L}_P(R) \subseteq \mathcal{L}(\varphi) \tag{7.7}$$

As stated in equation 2.5, the model checking problem consists of deciding the validity of the relation $\mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi)$. Thus, the final correctness theorem from equation 7.7 shows that replacing the system automaton with the reduced system automaton does not change the result of the model checking process, justifying the use of this partial order reduction optimization.

Chapter 8

Verification of On-The-Fly Partial Order Reduction

The primary goal of this thesis was to formally verify an algorithm for on-the-fly partial order reduction. In section 3.4, we gave two possible ways of adapting the off-line reduction algorithm for on-the-fly model checking. We decided to pursue the proof of the algorithm using the sequential product, which is described in section 3.4.2. This is also the approach chosen in [Pel96, Section 4].

During the process of formalizing the correctness proof of this algorithm, we discovered that one of the steps does not seem plausible. Indeed, we found what we believe to be a counterexample for one of the identities used in the proof. In the following sections, we introduce the concepts that are needed for the correctness proof. We will then give an overview of the proof itself as well as a detailed discussion concerning the step that we were not able to verify.

8.1 Automaton Completion

The central concept of the proof described in [Pel96] is that of automaton completion. It gives rise to the completed product automaton, which serves as an intermediate step between the interpreted system automaton and the product automaton. It is structurally similar to the product automaton, while still accepting the same language as the interpreted system automaton. It is intended to help in proving statements about the language of the product automaton. We chose to present the construction of the completed product automaton a little differently than is done in [Pel96] as we believe this alternative presentation to be more accessible.

We will introduce the idea of automaton completion using an example. We assume that there is a formula φ which contains only one propositional variable called p. This yields the following universe of sets of propositional variables:

$$\mathcal{P}(\Omega(\varphi)) = \mathcal{P}(\{p\}) = \{\{\}, \{p\}\} \tag{8.1}$$

We define the interpretation function P as follows:

$$P(q) = \begin{cases} \{p_1\} & \text{if } q = \omega \\ \{\} & \text{otherwise} \end{cases} \tag{8.2}$$

The idea is then illustrated in figure 8.1. Note that in this example, no partial order reduction takes place, as it is supposed to merely illustrate the idea of automaton completion. Thus, there are no generators and there is no reduced system automaton.

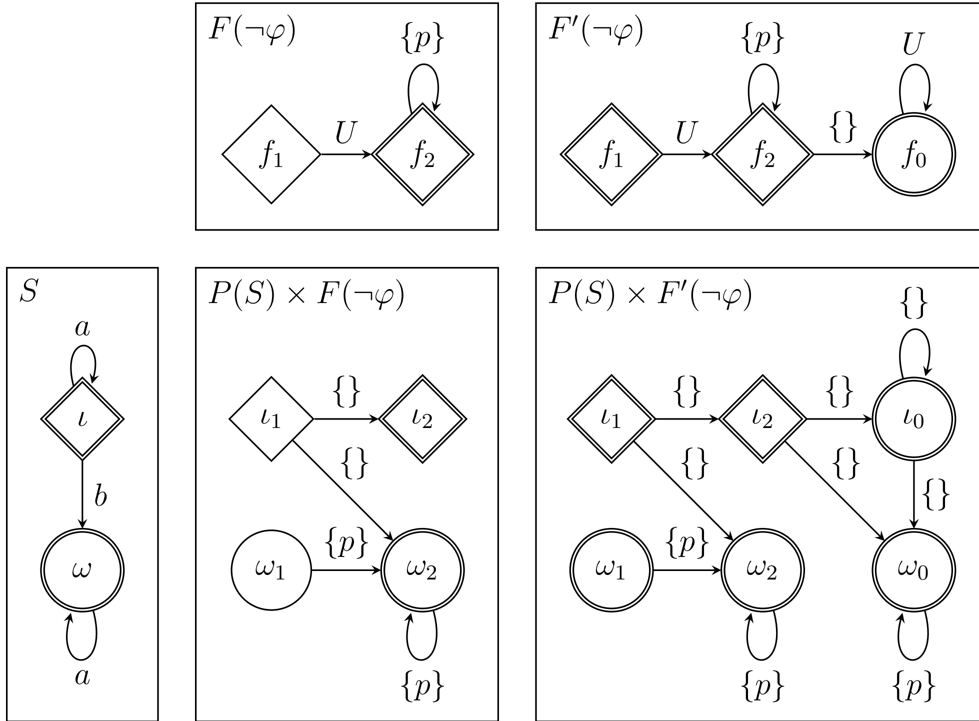


Figure 8.1: Example for automaton completion. Shown are the system automaton S, the formula automaton $F(\neg\varphi)$, the product automaton $P(S) \times F(\neg\varphi)$, the completed formula automaton $F'(\neg\varphi)$, and the completed product automaton $P(S) \times F'(\neg\varphi)$. We use shorthand state notation for the product automata. For instance, the product state (ω, f_1) is simply written as ω_1. Furthermore, we abbreviate the universe of sets of propositional variables $\mathcal{P}(\Omega(\varphi))$ with the variable U in this diagram.

The product automaton is constructed from the interpreted system automaton and the formula automaton as described in section 2.3.2. Note how this results in two instances of the system automaton with indices 1 and 2, respectively. The product automaton accepts fewer runs than the interpreted system automaton. This happens for two reasons. Firstly, not every state of the formula automaton is accepting. Secondly, the formula automaton does not contain outgoing transitions for every combination of formula state and set of propositional variables. For instance, the system automaton in figure 8.1 has an outgoing transition at state ι. This means that the interpreted system automaton also has an outgoing transition at this state, which is labeled with the interpretation of this state, in this case, $\{\}$. However, the state ι_2 in the product automaton is a dead end, since at state f_2, the formula automaton does not have an outgoing transition labeled $\{\}$.

This observation leads to the idea of the completed formula automaton. Here, an additional completion state f_0 is added. Whenever a state does not contain an outgoing transition for a specific set of propositional variables, a transition to this completion state is added. This also holds for the completion state itself, causing reflexive transitions for all sets of propositional variables to be added to it. Additionally, all the states in the completed formula automaton are accepting. In the example in figure 8.1, since at state f_2, the formula automaton does not have an outgoing transition labeled $\{\}$, such a transition is added to the completed formula automaton, targeting the completion state f_0.

We then define the completed product automaton to be the product of the interpreted system automaton and the completed formula automaton. The completed product automaton accepts the same language as the interpreted system automaton, while being structurally similar to the product automaton. The only difference lies with the accepting states as well as with states and transitions involving the completion state. Note that the part of the completed product automaton which involves the completion state is an exact replica of the interpreted system automaton itself.

8.2 The Reduced Completed Product Automaton

We now combine partial order reduction and automaton completion. Section 3.4.2 explains how the product automaton construction can be modified to incorporate the ample function instead of the transition relation of the system automaton, resulting in the reduced product automaton. Automaton completion can be considered a different modification to the product automaton construction, which uses the completed formula automaton instead of the formula automaton, resulting in the completed product automaton. Since these two modifications are independent, we

can apply both of them, yielding the reduced completed product automaton. Figure 8.2 gives an overview of the different product automaton constructions.

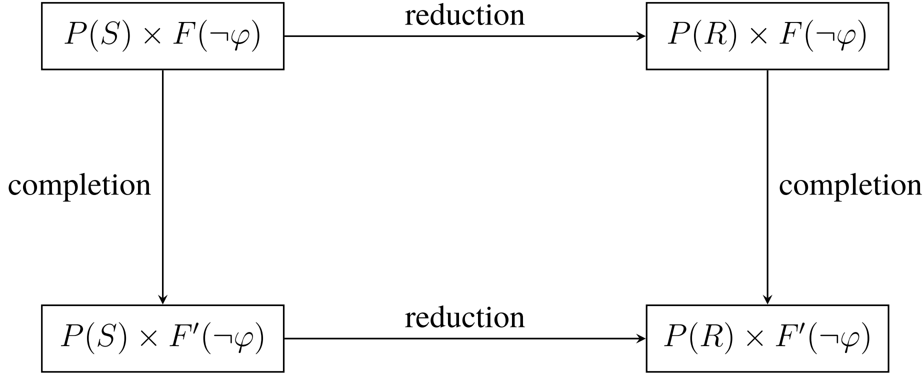


Figure 8.2: Overview of the various product automaton constructions. Shown are the product automaton $P(S) \times F(\neg\varphi)$, the reduced product automaton $P(R) \times F(\neg\varphi)$, the completed product automaton $P(S) \times F'(\neg\varphi)$, and the reduced completed product automaton $P(R) \times F'(\neg\varphi)$, as well as the relations between them.

As mentioned in the previous section, our way of defining the reduced completed product automaton is quite different from the one employed in [Pel96]. However, the resulting automaton is the same. What we call the reduced completed product automaton is called $\mathcal{A}'$ in [Pel96], an automaton constructed by a nondeterministic algorithm controlled by the formula automaton.

In [Pel96], it is then claimed that all the proofs about off-line partial order reduction from chapter 7 stay the same when replacing the reduced system automaton with the reduced completed product automaton, or rather, an uninterpreted variant thereof. However, this claim has not been formally proven and is thus only assumed to be true in our presentation of the proof.

From this assumption, we know that the final correctness theorem of off-line partial order reduction from equation 7.7 carries over when using the reduced completed product automaton instead of the interpreted reduced system automaton. This leads to the following statement:

$$\mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) \iff \mathcal{L}(P(R) \times F'(\neg\varphi)) \subseteq \mathcal{L}(\varphi) \tag{8.3}$$

That is, the interpreted language of the system automaton is included in the language of the formula if and only if the language of the reduced completed product automaton is included in the language of the formula. This theorem will be needed for the correctness proof.

8.3 Proof Architecture

Having taken care of all the prerequisites, we now want to prove the correctness of the on-the-fly partial order reduction algorithm described in section 3.4.2. Thus, we have to show that the model checking function implementing this algorithm actually decides the model checking problem stated in equation 2.5. We adapt the proof architecture from [Pel96, Section 4] and arrive at the following steps:

$$
\begin{aligned}
\mathrm{check}(S, \varphi, P) &\iff \neg\mathrm{has_lasso}(P(R) \times F(\neg\varphi)) && \text{(8.4a)}\\
&\iff P(R) \times F(\neg\varphi) \text{ has no reachable accepting cycle} && \text{(8.4b)}\\
&\iff \mathcal{L}(P(R) \times F(\neg\varphi)) = \{\} && \text{(8.4c)}\\
&\iff \mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi)) = \{\} && \text{(8.4d)}\\
&\iff \mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \overline{\mathcal{L}(\varphi)} = \{\} && \text{(8.4e)}\\
&\iff \mathcal{L}(P(R) \times F'(\neg\varphi)) \subseteq \mathcal{L}(\varphi) && \text{(8.4f)}\\
&\iff \mathcal{L}_P(S) \subseteq \mathcal{L}(\varphi) && \text{(8.4g)}
\end{aligned}
$$

Note that both the reduced product automaton as well as the reduced completed product automaton are generated automata. As such, close attention has to be paid to whether some step is performed on the generator or the explored automaton. During the formal verification effort, this has given rise to many proof obligations that are easily overlooked when appealing to intuition for the reasoning about generated automata. Nonetheless, we will not make this distinction here since the main focus of this chapter is not on the presentation of a formal proof, but rather on the issues with the proof of step 8.4d. Thus, we will simply treat these automata as if they were fully generated at all times.

In the following, we present a short explanation for each step.

8.4a This step follows directly from the definition of the model checking function given in equation 3.1.

8.4b The function $\mathrm{has_lasso}$ uses nested depth-first search to determine if an automaton has a reachable accepting cycle. Given that it is implemented correctly, $\mathrm{has_lasso}(P(R) \times F(\neg\varphi))$ does not hold if and only if there is no reachable accepting cycle in the reduced product automaton.

8.4c From the acceptance condition of Büchi automata, we can conclude that each reachable accepting cycle corresponds to some word in the language. Thus, the lack of reachable accepting cycles in the reduced product automaton is equivalent to its language being empty.

8.4d The proof for this step given in [Pel96, Section 4] uses the identity $\mathcal{L}(P(R) \times F(\neg\varphi)) = \mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi))$. We think that this identity does not hold and will discuss a counterexample in section 8.4. However, as argued in section 8.4.6, the step itself may still be valid.

8.4e This step is justified by equation 2.7, which states that for all formulae φ, it holds that $\mathcal{L}(F(\varphi)) = \mathcal{L}(\varphi)$ and equation A.4, which states that for all formulae φ, it holds that $\mathcal{L}(\neg\varphi) = \overline{\mathcal{L}(\varphi)}$.

8.4f A mere set-theoretic transformation is performed in this step.

8.4g As mentioned in section 8.2, [Pel96] claims that all the theorems about off-line partial order reduction carry over when using the reduced completed product automaton instead of the reduced system automaton. Equation 8.3, the transferred version of the final correctness theorem of off-line partial order reduction from equation 7.7, is used to prove this step.

8.4 Reduced Product Automaton Language

In order to complete the proof outlined in the previous section, we need to prove step 8.4d. The proof for this step given in [Pel96] uses the following identity:

$$\mathcal{L}(P(R) \times F(\neg\varphi)) = \mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi)) \tag{8.5}$$

That is, the language of the reduced product automaton is equal to the intersection of the languages of the reduced completed product automaton and the formula automaton. This identity immediately proves step 8.4d.

We consider the two inclusions that make up the equality statement. The proof of the inclusion $\mathcal{L}(P(R) \times F(\neg\varphi)) \subseteq \mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi))$ is straightforward. Every transition sequence accepted by the reduced product automaton is also accepted by the reduced completed product automaton. Additionally, a transition sequence accepted by the formula automaton can be extracted from a transition sequence accepted by the reduced product automaton by projecting every state in the sequence onto its formula state component.

The proof for the other direction, however, is rather tricky. In order to show that the inclusion $\mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi)) \subseteq \mathcal{L}(P(R) \times F(\neg\varphi))$ holds, we have to prove that every word which is accepted by both the reduced completed product automaton and the formula automaton is also accepted by the reduced product automaton.

The first intuition here could be that any transition sequence that is accepted by the reduced completed product automaton, and whose corresponding transition word is accepted by the formula automaton, is also accepted by the reduced product automaton. The idea here is that since the corresponding transition word is accepted by the formula automaton, the transition sequence never has to fall back on completion states or states that are only accepting in the reduced completed product automaton, but not in the reduced product automaton. However, if the formula automaton is nondeterministic, there may be a transition sequence whose corresponding transition word is accepted, while the sequence itself is not. This means that a transition sequence accepted by the reduced completed product automaton may not be accepted by the reduced product automaton.

At this point, is seems like it might be possible to construct a transition sequence that is accepted by the reduced product automaton from transition sequences accepted by the reduced completed product automaton and the formula automaton, respectively. The plan is to take the transition sequence accepted by the reduced completed product automaton, which is not necessarily accepted by the reduced product automaton, and replace the formula state component with the states in the transition sequence accepted by the formula automaton. Surely, this will yield a transition sequence that is accepted by the reduced formula automaton, as the sequence itself is accepted by the reduced completed product automaton, while its projection is accepted by the formula automaton. Unfortunately, this constructed transition sequence may not be a transition sequence in the reduced product automaton at all, as the instances of the system automaton in the reduced product automaton corresponding to each formula automaton state may have been reduced differently. Because of this, a transition sequence in the system automaton may be a transition sequence in the product automaton when paired with one transition run in the formula automaton, but not when paired with another one.

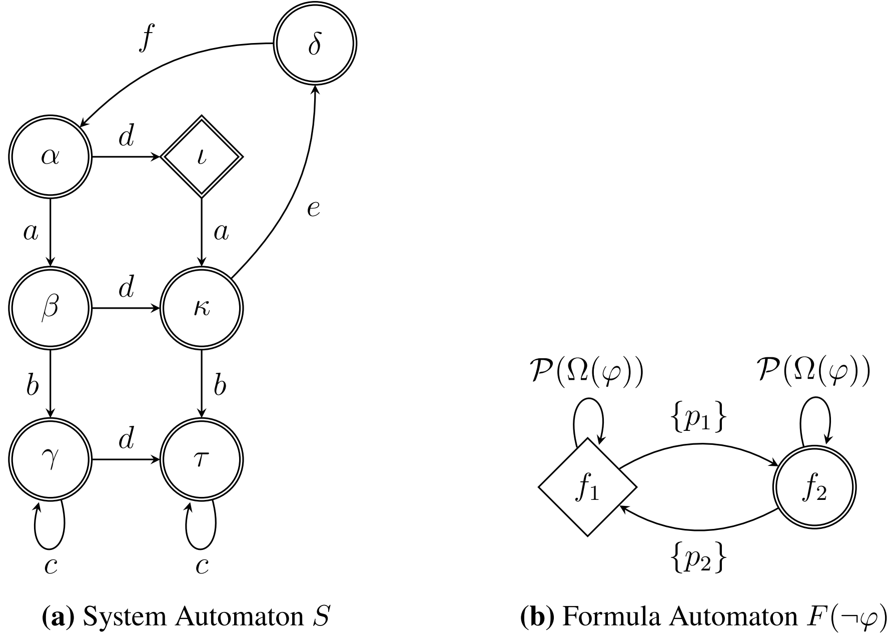
In the following sections, we will use this insight to construct a counterexample for the second inclusion of the identity from equation 8.5.

8.4.1 Setup

Figure 8.3 shows the setup for the counterexample, consisting of the system automaton S and the formula automaton $F(\neg\varphi)$.

We assume that the formula φ contains the propositional variables p_1 and p_2. This yields the following universe of sets of propositional variables:

$$\mathcal{P}(\Omega(\varphi)) = \mathcal{P}(\{p_1, p_2\}) = \{\{\}, \{p_1\}, \{p_2\}, \{p_1, p_2\}\} \tag{8.6}$$



(a) System Automaton S

(b) Formula Automaton $F(\neg\varphi)$

Figure 8.3: The setup for the counterexample.

We define the interpretation function P as follows:

$$P(q) = \begin{cases} \{p_1\} & \text{if } q = \delta \\ \{p_2\} & \text{if } q = \gamma \\ \{p_2\} & \text{if } q = \tau \\ \{\} & \text{otherwise} \end{cases} \tag{8.7}$$

With this definition, the operations b, e, and f are the only ones that change the interpretation of the current system state. We can thus assume that all other operations are invisible and define the set of visible operations V as follows:

$$V = \{b, e, f\} \tag{8.8}$$

8.4.2 Reduction

Next, we establish the prerequisites for performing partial order reduction on the given automata. We employ the following independence relation:

$$I = \{(a, d), (d, a), (b, d), (d, b), (c, d), (d, c)\} \tag{8.9}$$

Using the definition presented in section 6.2, it can be verified that this is indeed a valid independence relation for the given system automaton. The corresponding

diamond constellations can be seen in figure 8.3a. Note that the diamond constellation corresponding to the independence of the operations c and d is degenerate due to the reflexivity of operation c.

We now define the function ample as follows:

$$\mathrm{ample}(s, q) = \begin{cases} \{d\} & \text{if } q = \alpha \text{ and } \iota \notin s \\ \{a\} & \text{if } q = \alpha \text{ and } \beta \notin s \\ \mathrm{en}(q) & \text{otherwise} \end{cases} \tag{8.10}$$

The rules are applied in the given order. This means that a reduction is performed at state α, if possible. In all other cases, the ample function simply returns the set of enabled operations, meaning that no reduction takes place.

We have to verify that this ample function fulfills the requirements established in section 7.1. Since these properties always hold in the case that no reduction takes place, only the state α has to be considered.

C1 We verify that the two reduction possibilities given in equation 8.10 are both valid. In both cases, starting at state α, all the operations encountered before the first operation in the ample set are independent of all the operations in the ample set with respect to the independence relation introduced in equation 8.9.

C2 The guard conditions employed in equation 8.10 guarantee that in case of a reduction, none of the operations in the resulting ample set lead to a state in s when executed.

C3′ The only operations that can be part of a reduced ample set are a and d. Since they are both invisible according to the set of visible operations established in equation 8.8, there can never be a reduced ample set containing visible operations.

8.4.3 Product

We now construct the reduced product automaton. We use the transition relation of the formula automaton together with the ample function defined in equation 8.10 to build the transition function of the reduced product automaton according to the definition of the sequential product construction, which was introduced in section 3.4.2. With this transition function, it is then possible to define the generator of the reduced product automaton, which can be explored using generalized depth-first search. Figure 8.4 shows what the fully explored automaton looks like.

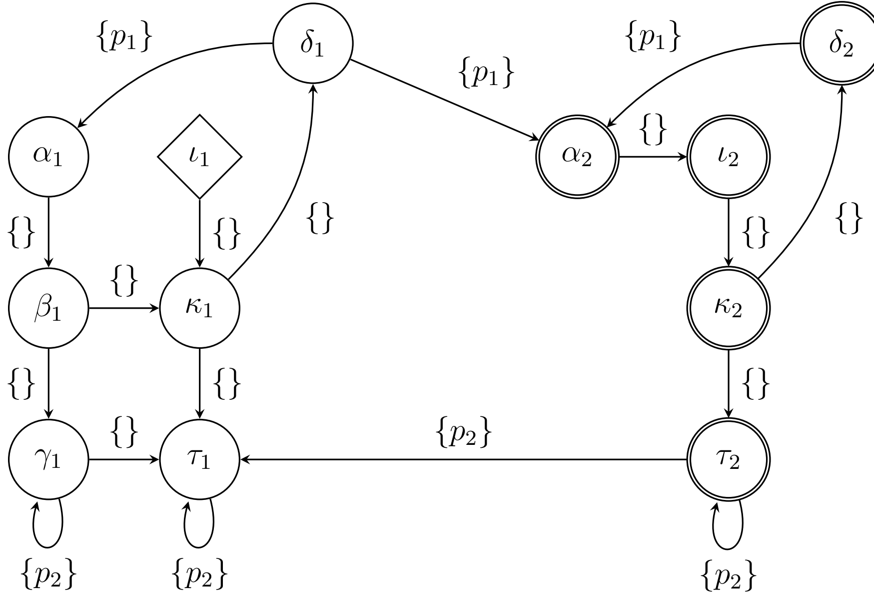


Figure 8.4: The reduced product automaton $P(R) \times F(\neg\varphi)$ for the counterexample. We use shorthand state notation. For instance, the state (α, f_1) is simply written as α_1. Note that since the reduced product automaton is already interpreted, its labels are sets of propositional variables instead of operations.

Using the shorthand state notation from figure 8.4, we describe the interesting parts of the depth-first search exploration, which occur at the states α_1, α_2, δ_1, and τ_2. The former two states are interesting because they are the only states at which reduction can possibly take place, the latter two are interesting because they are the only states at which non-reflexive transitions can be taken in the formula automaton. At all other states, the search algorithm merely explores an instance of the system automaton.

When the search algorithm reaches the states δ_1 and τ_2, the formula automaton can, in addition to the reflexive transitions, take a transition between its two states. This means that from state δ_1, there are transitions to both α_1 and α_2, and from state τ_2, there are transitions to both τ_1 and τ_2.

The search algorithm can only reach the state α_1 via the states ι_1, κ_1, and δ_1, resulting in the search stack $[\iota_1, \kappa_1, \delta_1, \alpha_1]$. With $P(\alpha) = \{\}$, the state f_1 is the only successor state of the formula automaton at α_1. Using this successor state of the formula automaton, we obtain the processed search stack as described in section 3.4.2, causing the ample function to be invoked as $\mathrm{ample}([\iota, \kappa, \delta, \alpha], \alpha)$. The ample function then tries to make a reduction according to the rules given in equation 8.10. The first rule cannot be chosen since the state ι is on the processed

search stack. However, the second rule can be applied since the state β is not on the processed search stack. Thus, we can conclude that the ample set is $\{a\}$, leading to the discovery of the state β_1. The transition (α, d, ι), which is also enabled in the system automaton, is never discovered in the part of the reduced product automaton that corresponds to the formula state f_1.

The state α_2 is also reached via the states ι_1, κ_1, and δ_1, resulting in the similar search stack $[\iota_1, \kappa_1, \delta_1, \alpha_2]$. As before, the fact that $P(\alpha) = \{\}$ holds forces the formula automaton to use the reflexive transition. However, as the formula state component of α_2 is f_2, the successor state of the formula automaton is also f_2. Thus, processing the search stack now results in the ample function being invoked as $\mathrm{ample}([\alpha], \alpha)$, since all the states with a formula state component other than f_2 were removed from the search stack. With this, the ample function can now use the first rule from equation 8.10, since the state ι is not on the processed search stack, resulting in the ample set $\{d\}$. Because of this, the transition (α, a, β) is never discovered in the part of the reduced product automaton that corresponds to the formula state f_2. This, in turn, causes large parts of the system automaton to be omitted from this part of the reduced product automaton.

The important thing to note about the reduced product automaton shown in figure 8.4 is that the two instances of the system automaton that it is comprised of have been reduced differently. The instance corresponding to the formula state f_1 omits operation d at state α, while the instance corresponding to the formula state f_2 leaves out operation a at state α. This fact will be used in section 8.4.5 to derive a contradiction for the identity given in equation 8.5.

8.4.4 Completion

Our next goal is to construct the reduced completed product automaton. To this end, we first consider the completed formula automaton. As can be seen in figure 8.3b, the formula automaton has outgoing transitions for all combinations of states and labels. This means that there is no need for an additional completion state in the completed formula automaton. Thus, the completed formula automaton shown in figure 8.5 looks exactly like the formula automaton, except for the fact that all states are accepting.

The reduced completed product automaton is shown in figure 8.6. As it is built by replacing the formula automaton in the product construction with the completed formula automaton, it is very similar to the reduced product automaton. Indeed, the only difference is that all states are accepting in the reduced completed product automaton, while they are not in the reduced product automaton.

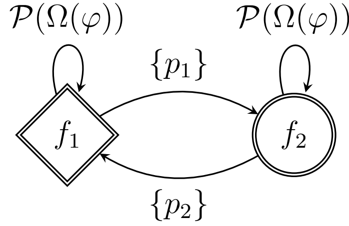


Figure 8.5: The completed formula automaton $F'(\neg\varphi)$ for the counterexample. Since the formula automaton shown in figure 8.3b already has outgoing transitions for all combinations of states and labels, its only difference to the completed formula automaton shown in this diagram is the fact that in the completed version of the automaton, all the states are accepting.

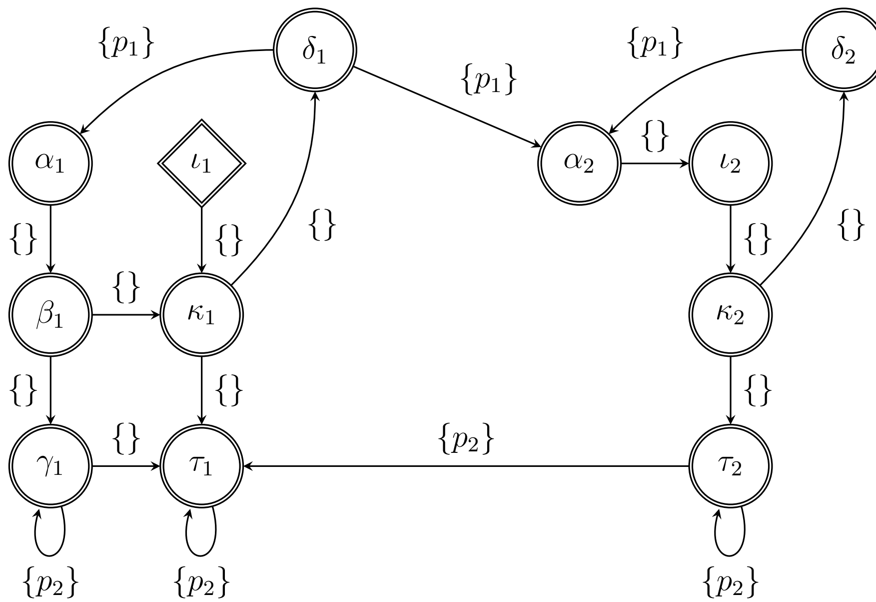


Figure 8.6: The reduced completed product automaton $P(R) \times F'(\neg\varphi)$ for the counterexample. We use shorthand state notation. For instance, the state (α, f_1) is simply written as α_1. As with the completed formula automaton, the only difference between the reduced completed product automaton shown in this diagram and the reduced product automaton shown in figure 8.4 is the fact that in the completed version of the automaton, all the states are accepting.

8.4.5 Contradiction

We can now use the automata constructed in the previous sections to contradict equation 8.5, which states that the language of the reduced product automaton is equal to the intersection of the languages of the reduced completed product automaton and the formula automaton. We define the word w:

$$w = \{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\} \cdot \{p_2\}^\omega \tag{8.11}$$

Then, we show that the word w is accepted by both the reduced completed product automaton and the formula automaton, but not by the reduced product automaton. We use the shorthand state notation for states in the product automata again.

We prove that the word w is accepted by the reduced completed product automaton. To this end, we define the run r as follows:

$$r = \iota_1 \cdot \kappa_1 \cdot \delta_1 \cdot \alpha_1 \cdot \beta_1 \cdot \gamma_1^\omega \tag{8.12}$$

A quick look at figure 8.6 confirms that the run r and the word w together form a transition sequence in the reduced completed product automaton. Since all the states in the reduced completed product automaton are accepting, every transition sequence starting at ι_1 is accepted. This proves that the word w is indeed accepted by the reduced completed product automaton.

The formula automaton shown in figure 8.3b simply accepts all those words that contain at least one occurrence of the set of propositional variables $\{p_1\}$. Since this is the case for the word w, we know that the formula automaton accepts it.

However, in contrast to the reduced completed product automaton, the run r is not accepted by the reduced product automaton due to the change in accepting states. Indeed, as can be seen in figure 8.4, if a word is to be accepted by the reduced product automaton, the corresponding run needs to contain at least one of the states corresponding to the formula state f_2, as the other part of the automaton does not contain any accepting states. In our case, the word w ends with $\{p_2\}^\omega$. This means that any run that is both accepted by the reduced product automaton and which corresponds to the word w has to end with τ_2^ω. However, it is impossible to reach the state τ_2 while consuming the label sequence $\{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\}$, since the shortest path to the state τ_2 consumes the label sequence $\{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\} \cdot \{\}$. Thus, there can be no run, which, together with the word w, forms a transition sequence that is accepted by the reduced product automaton, which proves that the reduced product automaton does not accept the word w.

Thus, w is accepted by both the reduced completed product automaton and the formula automaton, but not by the reduced product automaton, contradicting the statement from equation 8.5.

8.4.6 Consequences

Even though we have found a counterexample for the identity in equation 8.5, step 8.4d, for which it is used, may still be valid. This is because step 8.4d does not require the language of the the reduced product automaton to be equal to the intersection of the languages of the reduced completed product automaton and the formula automaton. Rather, it suffices if the emptiness of the former is equivalent to the emptiness of the latter.

In section 8.4, we have proven one of the inclusions of the identity in equation 8.5. Thus, the proof of step 8.4d could be completed using the following implication:

$$\mathcal{L}(P(R) \times F'(\neg\varphi)) \cap \mathcal{L}(F(\neg\varphi)) \neq \{\} \implies \mathcal{L}(P(R) \times F(\neg\varphi)) \neq \{\} \quad (8.13)$$

That is, if both the reduced completed product automaton and the formula automaton accept some word, then the reduced product automaton also accepts some word.

Note that this statement is weaker than the inclusion for which we provided the counterexample and that it is indeed not affected by the counterexample. While the word w defined in the previous section is not accepted by the reduced product automaton, we can construct a new word w' that is:

$$w' = \{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\} \cdot \{\} \cdot \{p_2\}^\omega \quad (8.14)$$

This word w' is, in a sense, equivalent to the word w, as it is obtained via the additional execution of the operation d in the system automaton, which is independent of all the operations occurring afterwards in w'.

We were not able to find a counterexample for the implication in equation 8.13 and suspect that it does indeed hold. However, even if it does, the counterexample has shown that there are cases in which the reduced completed product automaton and the formula automaton accept a word that is not accepted by the reduced product automaton. Thus, in order to prove the implication in equation 8.13, it will be necessary to use the transition sequence in the reduced completed product automaton to construct a completely new transition sequence with a different corresponding transition word. In particular, it is not possible to simply replace the formula state component of the states in the transition sequence with those in a transition run accepted by the formula automaton as one might be tempted to. This strongly indicates that if it exists at all, the proof of step 8.4d is non-trivial.

Chapter 9

Verification Results

As mentioned in section 4.4, we have decided not to present the actual formal theories in this thesis. However, most of the time spent on this thesis was spent working on them. Because of this, we want to at least give an overview of the results of the formal verification effort in this chapter. We organize the formal theories into the following six parts.

Basics Together, the HOL library and the CAVA project already provide formalizations for most of the basic concepts we needed. Nonetheless, a few definitions and theorems had to be added regarding basic data structures like sets, sequences, graphs, and relations. Some extensions were also made to automata-theoretic constructs like labeled transition systems and Büchi automata, as well as to the formalization of LTL formulae. Finally, a few additions were made to the formalizations concerning nondeterminism and the refinement of monadic programs.

Model Checking This part contains some basic definitions and theorems about automata-based model checking, concerning the system automaton, the formula, and the interpretation function together with all its extensions. We formalize the concepts described in chapter 2.

Automaton Generation As described in chapter 5, automata generated by generalized depth-first search play an important role in partial order reduction. As such, these and related concepts were formalized in this part. To do so, we use the refinement framework from [Lam12].

Partial Order Reduction Basics The basic concepts needed to prove the correctness of partial order reduction were introduced in chapter 6. As they provide the foundation for the formalizations of both off-line and on-the-fly partial order reduction, they were formalized in this part of the formal theories.

Off-Line Partial Order Reduction This part of the theories contains the formalization of the proof presented in chapter 7. All the theorems except for theorem 3.9 from section 7.3 and theorem 3.12 from section 7.5 have been formally proven.

On-The-Fly Partial Order Reduction As outlined in chapter 8, our efforts in verifying the on-the-fly partial order reduction approach presented in [Pel96] were unfortunately not successful. This part comprises the formal theories whose development led to the discovery of the issues described in section 8.4.

Finally, we give some statistics about the formal theories in figure 9.1. As with most software metrics, they cannot be taken as precise measurements of the volume or complexity of the theories. However, they can still convey a rough estimate of the effort that was required to develop those theories.

Part	Definitions	Theorems	Lines
Basics	73	342	3502
Model Checking	10	18	321
Automaton Generation	36	70	1279
Partial Order Reduction Basics	16	144	1851
Offline Partial Order Reduction	11	26	962
On-The-Fly Partial Order Reduction	46	49	1292
Total	192	649	9207

Figure 9.1: Statistics concerning the formal theories. Shown are the number of definitions, the number of theorems, and the lines of code in each part of the theories, as well as the total sum of each of these values.

Chapter 10

Conclusion

This thesis has produced two important results. Firstly, we were able to formalize significant parts of the correctness proof of off-line partial order reduction in chapter 7. These formalizations represent both a correctness certificate as well as a very detailed description of the proofs they implement. Secondly, attempting to formalize the correctness proof of on-the-fly partial order reduction in chapter 8 allowed us to discover a flaw in said proof which might otherwise not have been found. These two cases conveniently illustrate some of the possible outcomes of trying to formally prove a result that has only been informally proven so far.

At this point, there are multiple ways of continuing the work started in this thesis. An obvious choice is of course to complete the formalization of the correctness proof of off-line partial order reduction. While formally proving theorem 3.9 and theorem 3.12 may still require considerable effort, we would not expect to encounter any fundamental problems here.

Ultimately, it would be desirable to formally verify some on-the-fly partial order reduction algorithm. As mentioned in section 8.4.6, the overall soundness of the algorithm described in [Pel96] may not be affected by the counterexample. Thus, it may be possible to find a different proof which uses the run in the reduced completed product automaton to construct a different but equivalent run in the reduced product automaton. However, there is not much guidance for this proof and completing it will probably require considerable effort. Another possibility would be to use a different way of adapting the off-line partial order reduction algorithm for use in on-the-fly model checking. An approach which we have not yet explored consists of lazily constructing the reduced system automaton using off-line partial order reduction techniques. It is also possible that the insights gained in [CP96] concerning the use of a well-founded relation instead of generalized depth-

first search can help with the adaptation for on-the-fly model checking. Finally, there is the possibility of using a completely different approach such as static partial order reduction [Kur+98], which can avoid the issues with on-the-fly model checking altogether by not requiring the ample function to take the search stack as an additional parameter.

Once on-the-fly partial order reduction has been formally verified, it may be worthwhile to put some effort into refactoring the formal theories. Machine-checked theories are well-suited for refactoring, as the proof assistant can make sure that the proofs remain valid at all times. Through refactoring, it is then often possible to simplify both the definitions and the proofs involved in the theory, yielding a better understanding of the concepts in the process.

Finally, the optimization can be incorporated into the CAVA project [NEN; Esp+13; Esp+14] in order to obtain a fully verified model checker that makes use of on-the-fly partial order reduction. This should result in a significant performance improvement over the naive implementation when using model checking for parallel systems.

Appendix A

Definitions

In this appendix, we introduce some basic concepts and corresponding notation to be used throughout the thesis. These notions are not specific to model checking and the reader may already be familiar with some or all of them. Nonetheless, introducing them here enables us to agree on notation and terminology.

A.1 Sequences

Both finite and infinite sequences are considered. The empty sequence is denoted by ϵ. The concatenation of two sequences u and v is written $u \cdot v$. Iteration of v, that is, repeating the finite sequence v infinitely often, is denoted by v^ω. We write $v(i)$ to refer to the ith item in the sequence v. The suffix obtained by removing the first i items from v is written v^i.

We define the prefix relation $\sqsubseteq$ on all combinations of finite and infinite sequences. We write $u \sqsubseteq v$ if and only if u is a prefix of v. An infinite sequence can never be a prefix of a finite sequence.

We introduce the notion of a selection function from [Pel96, Definition 3.8] for both finite and infinite sequences. Given a sequence w, a selection function is a sequence s that assigns a truth value to each position in w. We can then write $s(w)$ to denote the sequence remaining after removing all the items occurring at positions in w which are assigned the value $\bot$ by s. We define $\bar{s}$ to be the inverse of s, that is, the selection function where the assigned truth values have been negated. We can thus write $\bar{s}(w)$ to denote the sequence remaining after removing all the items occurring at positions in w which are assigned the value $\top$ by s.

A.2 Linear Temporal Logic

Linear temporal logic (LTL) is an extension of propositional logic designed to include a temporal component. Formulae in linear temporal logic can be used to express properties of infinite sequences. Given that p is a propositional variable and φ and ψ are LTL formulae, the following are also LTL formulae:

$\top$	truth	(A.1a)
$\bot$	falsity	(A.1b)
p	propositional variable	(A.1c)
$\neg\varphi$	negation	(A.1d)
$\varphi \land \psi$	conjunction	(A.1e)
$\varphi \lor \psi$	disjunction	(A.1f)
$\varphi \,\mathrm{U}\, \psi$	until	(A.1g)
$\varphi \,\mathrm{R}\, \psi$	release	(A.1h)
$\mathrm{X}\, \varphi$	next	(A.1i)

Given a formula φ, we also define $\Omega(\varphi)$ to be the set of all those propositional variables that occur in the formula.

Next, we define the semantics of LTL formulae by introducing the satisfaction relation $\models$, which works with sequences of sets of propositional variables. Writing $w \models \varphi$ means that the formula φ accepts the sequence w, or, equivalently, that the sequence w satisfies the formula φ. Given a sequence w, the following rules recursively define the satisfaction relation for arbitrary LTL formulae:

$$w \models \top \iff \top \tag{A.2a}$$
$$w \models \bot \iff \bot \tag{A.2b}$$
$$w \models p \iff p \in w(0) \tag{A.2c}$$
$$w \models \neg\varphi \iff \neg w \models \varphi \tag{A.2d}$$
$$w \models \varphi \land \psi \iff w \models \varphi \land w \models \psi \tag{A.2e}$$
$$w \models \varphi \lor \psi \iff w \models \varphi \lor w \models \psi \tag{A.2f}$$
$$w \models \varphi \,\mathrm{U}\, \psi \iff \exists\, i.\; w^i \models \psi \land \forall\, j < i.\; w^j \models \varphi \tag{A.2g}$$
$$w \models \varphi \,\mathrm{R}\, \psi \iff \forall\, i.\; w^i \models \psi \lor \exists\, j < i.\; w^j \models \varphi \tag{A.2h}$$
$$w \models \mathrm{X}\, \varphi \iff w^1 \models \varphi \tag{A.2i}$$

As one would expect, $\top$ is the formula that accepts all sequences and $\bot$ is the formula that accepts no sequences. The formula consisting only of the propositional

variable p accepts w if and only if p is contained in the set of propositional variables that makes up the first item of w. The connectives $\neg$, $\wedge$, and $\vee$ are simply reduced to their counterparts from propositional logic. The formula $\varphi \operatorname{U} \psi$ accepts w if and only if ψ eventually becomes true, with φ holding until this is the case. The formula $\varphi \operatorname{R} \psi$ expresses that ψ has to remain true up to and including the point at which φ becomes true for the first time. Finally, the formula $\operatorname{X} \varphi$ accepts w if and only if φ is true in the next step of w.

Rule A.2c is concerned with the propositional variables in LTL formulae. It thus makes the semantic connection between a formula and some sequence that is accepted or rejected by this formula. With this rule, the sets of propositional variables that make up the sequence can be interpreted to contain exactly those propositional variables that hold at their respective point of the sequence.

We define the language $\mathcal{L}(\varphi)$ of an LTL formula φ to be the set of all those sequences that it accepts:

$$\mathcal{L}(\varphi) = \{w \mid w \models \varphi\} \tag{A.3}$$

Given these definitions, it is possible to prove the following equality concerning the language of the negation of an arbitrary formula φ:

$$\mathcal{L}(\neg\varphi) = \overline{\mathcal{L}(\varphi)} \tag{A.4}$$

A.3 Büchi Automata

A finite automaton that operates on infinite words is called an ω-automaton. A Büchi automaton is a type of ω-automaton. A Büchi automaton A consists of the following parts:

$\mathcal{Q}(A)$	the set of states
$\Sigma(A)$	the set of labels
$\Delta(A)$	the transition relation, a subset of $\mathcal{Q}(A) \times \Sigma(A) \times \mathcal{Q}(A)$
$\mathcal{I}(A)$	the set of initial states, a subset of $\mathcal{Q}(A)$
$\mathcal{F}(A)$	the set of accepting states, a subset of $\mathcal{Q}(A)$

If the transition relation contains the tuple (q_1, a, q_2), it means that from state q_1, there is a transition to state q_2 labeled a. Note that instead of the transition relation Δ, it is also possible to use a transition function δ that is either of type $\mathcal{Q}(A) \to \mathcal{P}(\Sigma(A) \times \mathcal{Q}(A))$ or of type $\mathcal{Q}(A) \times \Sigma(A) \to \mathcal{P}(\mathcal{Q}(A))$. All three of these representations are isomorphic, so we can freely choose which one to use based on their practicality or technical aspects.

We introduce the notion of a transition sequence, which is a finite or infinite sequence of transitions in the transition relation. For each transition sequence, there is a corresponding transition run, which is the sequence of states in the transition sequence, and a corresponding transition word, which is the sequence of labels in the transition sequence. Together, a transition run and a transition word uniquely identify the transition sequence to which they correspond.

We can use these transition sequences to define the acceptance condition for Büchi automata. We say that a transition sequence is accepted if and only if the corresponding transition run starts with an initial state and contains infinitely many accepting states. If a transition sequence is accepted, we say that the corresponding transition run and the corresponding transition word are accepted as well. The language $\mathcal{L}(A)$ of a Büchi automaton A is then defined to be the set of all those sequences that are accepted transition words. Similarly, we define $\mathrm{runs}(A)$ to be the set of all those sequences that are accepted transition runs.

In this thesis, we will often use diagrams to illustrate Büchi automata. One such diagram is shown in figure A.1 in order to introduce some of the conventions we use in these types of diagrams.

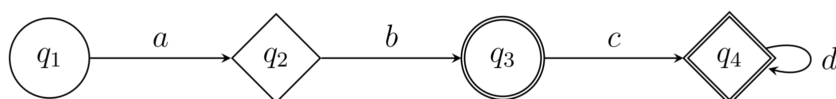


Figure A.1: Example of a diagram for Büchi automata. Shown are, from left to right, a regular state, an initial state, an accepting state, and a state that is both initial and accepting, as well as some transitions between these states.

A.3.1 Deterministic Automata

We say that a transition relation is deterministic if and only if for for each q_1 and a, there exists at most one q_2, such that (q_1, a, q_2) is in the transition relation.

A Büchi automaton is called deterministic if and only if its transition relation is deterministic and if it has exactly one initial state.

Given a deterministic Büchi automaton, for every sequence of labels, tracing the transitions induced by these labels starting at the initial state yields at most one transition sequence. Thus, in the context of deterministic Büchi automata, we will often simply talk about a transition word instead of a transition sequence, since the latter one is uniquely specified by the former, if it exists at all.

A.3.2 Product Automata

Let A and B be Büchi automata, such that for at least one of them, all states are accepting. We define the product automaton $A \times B$ as follows:

$$\mathcal{Q}(A \times B) = \mathcal{Q}(A) \times \mathcal{Q}(B) \tag{A.5a}$$
$$\Sigma(A \times B) = \Sigma(A) \cap \Sigma(B) \tag{A.5b}$$
$$\Delta(A \times B) = \{((p, r), a, (q, s)) \mid (p, a, q) \in \Delta(A) \wedge (r, a, s) \in \Delta(B)\} \tag{A.5c}$$
$$\mathcal{I}(A \times B) = \mathcal{I}(A) \times \mathcal{I}(B) \tag{A.5d}$$
$$\mathcal{F}(A \times B) = \mathcal{F}(A) \times \mathcal{F}(B) \tag{A.5e}$$

This definition can also be extended to work with transition functions instead of transition relations. However, for the sake of brevity, it is not included here.

With this definition, the following holds for the product automaton's language:

$$\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B) \tag{A.6}$$

That is, the language of the product automaton is the intersection of the languages of the factor automata.

Bibliography

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN: 978-0-262-03270-4. URL: http://mitpress.mit.edu/books/model-checking.

[Cou+92] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. "Memory-Efficient Algorithms for the Verification of Temporal Properties". In: *Formal Methods in System Design* 1.2-3 (Oct. 1992), pp. 275–288. ISSN: 0925-9856. DOI: 10.1007/BF00121128.

[CP96] Ching-Tsun Chou and Doron Peled. "Formal Verification of a Partial-Order Reduction Technique for Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 241–257. ISBN: 978-3-540-61042-7. DOI: 10.1007/3-540-61042-1_48.

[Esp+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. "A Fully Verified Executable LTL Model Checker". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 463–478. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_31.

[Esp+14] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. "A Fully Verified Executable LTL Model Checker". In: *Archive of Formal Proofs* (May 2014). Formal proof development. ISSN: 2150-914x. URL: http://afp.sf.net/entries/CAVA_LTL_Modelchecker.shtml.

[Ger+96] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. "Simple On-the-fly Automatic Verification of Linear Temporal Logic". In: *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*. London, UK: Chapman & Hall, Ltd., 1996, pp. 3–18. ISBN: 0-412-71620-8.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem*. Ed. by Patrice Godefroid. Vol. 1032. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996. ISBN: 978-3-540-60761-8. DOI: 10.1007/3-540-60761-7.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker. Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003. ISBN: 0-321-22862-6. URL: http://spinroot.com/spin/Doc/Book_extras/.

[HPY96] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. "On Nested Depth First Search". In: *Proceedings of the Second SPIN Workshop*. Vol. 32. 1996, pp. 81–89. URL: http://spinroot.com/spin/Workshops/ws96/papers.html.

[Kur+98] Robert Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. "Static Partial Order Reduction". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 345–357. ISBN: 978-3-540-64356-2. DOI: 10.1007/BFb0054182.

[Lam12] Peter Lammich. "Refinement for Monadic Programs". In: *Archive of Formal Proofs* (Jan. 2012). Formal proof development. ISSN: 2150-914X. URL: http://afp.sf.net/entries/Refine_Monadic.shtml.

[Lam14] Peter Lammich. "The CAVA Automata Library". In: *Archive of Formal Proofs* (May 2014). Formal proof development. ISSN: 2150-914X. URL: http://afp.sf.net/entries/CAVA_Automata.shtml.

[NEN] Tobias Nipkow, Javier Esparza, and Bernhard Nebel. *The CAVA Project*. Technische Universität München and Universität Freiburg. URL: https://cava.in.tum.de/.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: http://www.springer.com/978-3-540-43376-7.

[Pel93] Doron Peled. "All from One, One for All: On Model Checking Using Representatives". In: *Computer Aided Verification*. Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 409–423. ISBN: 978-3-540-56922-0. DOI: 10.1007/3-540-56922-7_34.

[Pel96] Doron Peled. "Combining Partial Order Reductions with On-the-Fly Model-Checking". In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64. ISSN: 0925-9856. DOI: 10.1007/BF00121262.

[Pel98] Doron Peled. "Ten Years of Partial Order Reduction". In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 17–28. ISBN: 978-3-540-64608-2. DOI: 10.1007/BFb0028727.

[PNW] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. *Isabelle*. University of Cambridge, Technische Universität München, and Université Paris-Sud. URL: http://isabelle.in.tum.de/.

[PW97] Doron Peled and Thomas Wilke. "Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator". In: *Information Processing Letters* 63.5 (1997), pp. 243–246. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(97)00133-6.

[SE05] Stefan Schwoon and Javier Esparza. "A Note on On-The-Fly Verification Algorithms". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 174–190. ISBN: 978-3-540-25333-4. DOI: 10.1007/978-3-540-31980-1_12.

[SL14] Alexander Schimpf and Peter Lammich. "Converting Linear-Time Temporal Logic to Generalized Büchi Automata". In: *Archive of Formal Proofs* (May 2014). Formal proof development. ISSN: 2150-914X. URL: http://afp.sf.net/entries/LTL_to_GBA.shtml.

[Val91] Antti Valmari. "Stubborn Sets for Reduced State Space Generation". In: *Advances in Petri Nets 1990*. Ed. by Grzegorz Rozenberg. Vol. 483. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 491–515. ISBN: 978-3-540-53863-9. DOI: 10.1007/3-540-53863-1_36.