Technische Universität Darmstadt

Department of Computer Science

Master's Thesis

Lazy POR for the Verification of Distributed Programs

by

Patrick Metzler

October 31, 2014

Examiner: Prof. Neeraj Suri, Ph.D.
Advisor: Habib Saissi, M.Sc.

Erklärung zur Abschlussarbeit

gemäß §22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Patrick Metzler, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 31. Oktober 2014 _____

Abstract

Model checking-based verification of concurrent systems is challenged by state explosion, the exponential growth of a system's state space. Partial order reduction (POR) mitigates this problem by restricting the verification to a reduced state space while preserving the soundness for properties such as the absence of deadlocks. During the last ten years, advances in the verification of concurrent programs have been made by developing algorithms which reduce a program's state space based on its dynamic behavior, known as dynamic partial order reduction (DPOR).

In this thesis, we present *LazyPOR*, a DPOR algorithm with the novel ability to plan in advance and explore complete program executions. In contrast to LazyPOR, most previously presented DPOR algorithms explore only single program steps at a time, which we believe produces an unnecessary running time overhead. Our experiments on a prototype implementation of LazyPOR indicate that the running time overhead per explored program execution is considerably reduced in comparison to the well-established DPOR algorithm by Flanagan and Godefroid.

As a foundation to our algorithm, we introduce a new characterization of system executions, *trace constraint systems*, and prove their sound applicability for DPOR. Based on trace constraint systems, we provide a correctness result for our algorithm with respect to Mazurkiewicz's trace theory. Finally, we empirically evaluate LazyPOR on challenging example programs and suggest optimizations for particular classes of concurrent programs.

Contents

List of Notations

$u \simeq u'$	Mazurkiewicz equivalence, page 2
$\mathit{enabled}(s) \subseteq T$	W, page 5
$s_0 \in S$	Initial state, page 5
$\mathit{range}(l)$	Range of a list, page 5
${}_{l_1}l_2$	List with removed prefix, page 6
$s \in S$	State, page 5
$TS = (S, s_0, T)$	Transition system, page 5
$t(s)$	Resulting state of a transition, page 5
$t \in T$	Transition, page 5
$l_1 \cdot l_2$	Concatenation of lists, page 5
$l_1 \sqsubseteq l_2$	Prefix relation on lists, page 5
$\mathit{list}(A, O)$	List induced by a totally ordered set, page 5
$D(u, s_1) \subseteq \{1, \ldots, n\}^2 \times S$	Dynamic dependency relation of a sequence at a state, page 6
$s_1 \xrightarrow{t_1 \cdots t_n} s_{n+1}$	Feasible transition sequence at a state, page 6
$u(s_1)$	Resulting state of a transition sequence, page 6
$u, v, w \in T^*$	Transition sequence, page 6
$\nparallel_u$	Dynamic dependency relation of a sequence, page 7
$\mathit{max\text{-}seq}(s)$	Set of maximal transition sequences at a state, page 6
$\mathit{dep}(w, s)$	Dependency list of a transition sequence at a state, page 7
$u_{\lvert t_1, t_2}$	Dependency list of u projected to t_1, t_2, page 7
$\mathit{swapped\text{-}dep}(u, v)$	swapped dependency, page 7
$[u]_\simeq$	Trace of a transition sequence, page 7

CS	Trace constraint system, page 8
$CS(w, s)$	Trace constraint system of a transition sequence at a state, page 9
$CS_\varnothing$	Empty trace constraint system, page 8
$CS' \sqsubseteq CS$	Prefix relation on trace constraint systems, page 9
$SOL(CS)$	Set of solution sequences of a trace constraint system, page 9
$\mathit{variations}(u, s)$	Variations of a transition sequence at a state, page 12
$\mathit{swapped\text{-}set}(u, v)$	set of swapped dependencies of u and v, page 16
$CS' \sqsubseteq_s CS$	Feasible prefix of CS at s, page 16
$<_\alpha$	Called-before relation, page 16
$\mathit{max\text{-}feasible}_{CS}(s)$	Maximal feasible prefix of CS at s, page 16

Chapter 1

Introduction

Safety and security critical information technology systems require a thorough analysis in order to be trustworthy. Often, it is advantageous to be able to verify that a system satisfies certain properties, for example that the system always responds correctly, that the system never enters a dangerous state, or that secret information is never leaked to a public channel. A number of different system verification approaches exist, which provide a wide range of trade-offs between a high expressiveness of supported system properties, a high precision of the analysis result, and small resource requirements such as for time and space.

Model checking describes a family of system verification techniques with a high expressiveness of supported system properties and a high precision of the analysis result, compared to other system analysis approaches such as static program analysis. For any supported system property, model checking either proves that the system satisfies the property or provides a counterexample, i.e., a proof that the system does not satisfy the property. In order to provide such a precise analysis result, model checking algorithms are usually restricted to systems that can be modeled by a finite model, for example a finite state automaton. In this case, it is possible to model-check a system by visiting every reachable state of its model. However, especially for large systems, the exploration of a model's complete state space can be prohibitively expensive, which constitutes one of the main disadvantages of model checking compared to other system verification techniques. One approach to limit the resource requirements of a model checking algorithm consists of reducing the state space of a system's model. Whenever such a reduction is sound, it is sufficient for a model checking algorithm to only visit the states of the reduced state space.

Concurrent systems constitute a class of software for which the analysis by a model checking algorithm may be extremely costly. Concurrent systems allow actions to be executed in parallel. Often, their results depend on the order in which parallel actions take effect. However, this order may usually vary with every system execution. Therefore, the state space of corresponding models grows exponentially with the number of parallel processes, which is known as *state explosion* [Val96]. Because of state space explosion, naive depth-first search model checking is unfeasible for large concurrent systems.

Partial order reduction (POR) is intended to reduce the state space of models for concurrent systems by distinguishing between dependent and independent actions. The order in which independent actions take effect does not effect

the overall behavior of the system. Therefore, the exploration of certain states can be avoided while preserving the correctness of the model checking results for properties describing the overall state of a system. The properties suitable for the use with POR include interesting properties such as the absence of deadlocks and properties expressed in the linear-time temporal logic without the *next* operator [BK08, CGP01].

In this thesis, we present a new POR technique called LazyPOR, which uses a new characterization of system executions. It comprises a POR algorithm which directly works on representations of system executions and, unlike most other POR algorithms, explores complete system executions at once. In particular, we make the following contributions.

- a new POR algorithm *LazyPOR*
- a correctness proof and a detailed formalization of concepts
- a new formalization of system executions as constraint systems with *dependency lists* and a correspondence proof ensuring soundness
- a prototype implementation of LazyPOR showing its practical applicability
- an empirical evaluation of our algorithm with suggestions optimizations
- an experimental comparison of LazyPOR and the POR algorithm by Flanagan and Godefroid

We provide a correctness result for our algorithm which guarantees a correct analysis for all properties expressed in the linear-time temporal logic without the *next* operator, including the absence of deadlocks. Our experimental results show that a prototype implementation of LazyPOR is considerably faster than our implementation of an existing POR algorithm in the case of two well-known benchmarks from the literature. Additionally, we suggest improvements to our technique which increase its performance and potentially avoid any unnecessary exploration.

We proceed by detailing the motivation for POR, reviewing main concepts of POR, namely *Mazurkiewicz equivalence* and *persistent sets*, and state the motivation for LazyPOR. We discuss related POR techniques and continue with basic definitions for our method. Subsequently, we describe our characterization of system executions, present our POR algorithm and prove its correctness. In order to evaluate its performance in challenging situations, we empirically apply our algorithm to example programs and show experimental results which compare our algorithm with an existing POR technique by Flanagan and Godefroid [FG05].

1.1 Background and Motivation

Throughout this thesis, we use *transition systems* in order to model concurrent systems. Intuitively, a transition modifies the state of a system and constitutes an atomic action, i.e., a single step of the system. POR mitigates the state explosion of a concurrent system by identifying transitions whose execution order does not effect the overall behavior of a system. Transitions with this

```
1 Thread 1 {        4 Thread 2 {                    7 Thread n {
2   write x[1]      5   write x[2]      ...        8   write x[n]
3 }                 6 }                            9 }
```

Listing 1.1: A program describing a concurrent system.

property are called *independent*. For example, let S be a concurrent system described by the program in listing 1.1. The system consists of a variable number of processes and the same number of memory locations. Each process writes to a distinct memory location, respectively, and each write operation corresponds to one transition of the system. Because no conflicting write operations to a single memory location occur, all transitions of S are independent. Hence, after all processes have been executed, the system is in a final state which is independent from the order in which individual transitions have been executed. As every memory location has two states (before and after the write operation of the corresponding process), the total number of states is $2^n + 1$, where n is the number of processes. This state explosion, i.e., an exponential growth in the number of states, is typical for concurrent systems.

Once the transitions of S are identified as independent, POR can be performed to reduce the state space to $n+1$ states. These states correspond to the initial, intermediate, and final states visited when executing all transitions of S in an arbitrary order. When model-checking for a system property compatible with POR, it is sufficient to explore only these $n+1$ states instead of all 2^n+1 states of S, which constitutes a considerable saving.

The formal foundation of POR is trace theory, introduced by Mazurkiewicz [Maz86]. Intuitively, trace theory describes a particular (partial) execution of a concurrent system by a *trace*, which contains the following information.

- the set of transitions which are executed and
- the order in which dependent transitions are executed

The order in which independent transitions have been executed is left unspecified. By the definition of dependence between transitions, the information contained in a trace is sufficient to describe an execution of a concurrent system, as the order of independent transitions is considered to be irrelevant in trace semantics.

A trace can formally be specified by a partial order on the executed transitions, where a pair of transitions (t_1, t_2) is related by the partial order if t_1 and t_2 are dependent and if t_1 has been executed before t_2. Traces are defined as equivalence classes on sequences of transitions with the following equivalence relation, known as *Mazurkiewicz equivalence*. Two transition sequences u, v are equivalent, written $u \simeq v$, if v can be obtained from u by consecutively swapping independent, adjacent transitions. If u is a transition sequence of a concurrent system, we write $[u]_\simeq$ for the trace of u, i.e., the set of all transition sequences equivalent to u. This set corresponds to the set of all linear extensions of the partial order describing $[u]_\simeq$.

Different approaches for detecting dependencies between transitions exist. For example, in a system described by a concurrent program, any two transitions

```
1 Thread 1 {
2   t1: x := 1
3 }
4 Thread 2 {
5   t2: y[x] := 1
6 }
7 Thread 3 {
8   t3: y[0] := 2
9 }
```

Listing 1.2: A program with dynamically changing dependencies. All memory locations are initialized to 0.

which access the same memory location and where at least one of the two transitions is a write operation may be considered dependent. If dependencies are detected statically, the memory location accessed by a transition may have to be conservatively approximated. In the program of listing 1.1, all referenced memory locations are statically known. However, if the memory locations were calculated only at run time, it could be difficult or even undecidable to determine the exact memory location accessed by a transition. Therefore, conservative approximations may be necessary which consider transitions as dependent even if they are dependent only at certain states and independent at others.

In contrast to a static detection of dependencies between transitions, *dynamic partial order reduction (DPOR)* techniques calculate dependencies dynamically, during the exploration of the state space. In most computational models, more precise information about dependency is available at exploration time than without knowledge of concrete or symbolic states. For example, consider a concurrent program where the memory location of a write operation W depends on the state it is executed in. DPOR is able to adapt the exploration of the program's state graph to the fact whether another read or write operation accesses the same memory location in the same execution path. POR with static dependency information has to conservatively overapproximate the set of memory locations accessed by the write operation W, because all states where this operation can be executed have to be considered. This overapproximation results in a coarser dependency relation and may lead to a smaller reduction of the program's state graph.

The state search algorithm presented in this thesis, LazyPOR, is a DPOR algorithm, i.e., is capable of dynamically detecting dependencies. We sketch its approach on the following example depicted in listing 1.2. This program corresponds to a transition system with three processes and one transition per process, labeled t_1, t_2, t_3. Process 1 writes to the global variable x. Process 2 writes to a global array y at a position defined by x. Process 3 writes to y at the static location 0. Transition t_1 is dependent with t_2 in every state and never dependent with t_3. Transition t_2 is dependent with t_3 only if t_1 has not been executed before t_2.

LazyPOR begins by exploring a complete, arbitrary execution of the program; for example, it explores the transition sequence $u = t_1t_2t_3$. The memory locations accessed by the explored transitions are tracked and the dependency between t_1 and t_2 is detected. Our algorithm constructs a transition sequence v which corresponds to u except that the dependent pair (t_1, t_2) is swapped. For example, $v = t_2t_1t_3$. Subsequently, it explores the constructed transition sequence v and again tracks the accessed memory locations. In v, the new dependency (t_2, t_3) is detected. Again, our algorithm constructs a new transition sequence w from v where this race is reversed, for example $w = t_3t_2t_1$. The new

sequence w is explored and checked for dependencies. The dependency (t_3, t_2) is detected but discarded as t_3 and t_2 have already been executed in this order. As no additional dependencies remain, the state space has been sufficiently explored and the algorithm terminates.

Each of the explored transition sequences $u = t_1t_2t_3$, $v = t_2t_1t_3$, and $w = t_3t_2t_1$ corresponds to a trace of the transition system corresponding to listing 1.2 and all remaining transition sequences which contain t_1, t_2, and t_3 are equivalent to either u, v, or w. Moreover, the transition sequences u, v, or w correspond to complete program executions. Our algorithm explores them without interrupting the program, hence the name LazyPOR. We conjecture that the exploration of multiple transitions at once, or even complete program executions at once, improves the running time performance of state search algorithms in practice.

In contrast to our algorithm, most existing DPOR algorithms explore single transitions at a time. The DPOR algorithm by Flanagan and Godefroid [FG05], for example, explores single transitions and subsequently performs a recursive call. At each state in the state graph of a system, it calculates a *persistent set*, which is a subset of all enabled transitions in that state. All transitions in the current persistent set are explored and the algorithm is called recursively at the resulting states.

POR and DPOR techniques can be combined with additional model checking optimizations, for example *stateful* exploration. A stateful state search maintains a table of visited states and only continues its exploration at a certain state if this state has not been visited before. This optimization may improve the running time performance but may considerable increase the memory consumption as well. Algorithms without a stateful exploration are called *stateless*.

1.2 Related work

A survey of model checking and POR in particular can be found in the books by Baier and Katoen [BK08] and Clarke [CGP01]. Trace theory has been reviewed, without a special focus on model checking, by Mazurkiewicz [Maz86] and Aalbersberg [AR88]. An algorithm efficiently generating all linear extensions of a poset, i.e., all total orders which are a superset of the given poset, has been presented by Pruesse and Ruskey [AR88]. Although we do not generate multiple linear extensions of a poset, we generate one linear extension for each trace of a system in order to find a representative of the trace.

DPOR has been first introduced by Flanagan and Godefroid [FG05]. It has been presented as a stateless, persistent set-based model checking algorithm. Intuitively, their model checking algorithm begins with exploring an arbitrary maximal path of a state graph. At each state where an alternative scheduling decision is necessary for a sound analysis, a set of processes is maintained which need to be scheduled additionally. Similar to the DPOR algorithm by Flanagan and Godefroid, our algorithm determines dependencies after a maximal path is executed. However, our algorithm requires at most one recursive call per maximal path, whereas Flanagan and Godefroid's algorithm performes a recursive call after each transition of a particular path in the state graph of a system.

DPOR can be combined with the *sleep set* technique [God90, God95]. This technique exploits information about the current path during the exploration of

a state space and associates a sleep set with each state. Transitions in the sleep set of a state will not be explored from that state. A path in a state graph where all enabled transitions of its last state in the corresponding sleep set is called *sleep set-blocked*. Sleep set-blocked paths do not need to be explored in order to explore all Marzurkievicz traces. However, it is possible for a path to become sleep set-blocked only after a number of transitions, rendering the exploration of these transition redundant even when using the sleep set technique. Rather than reducing the number of states which are explored (as it is the case with the persistent set technique), the sleep set technique reduces only the number of transitions.

One of the first attempts to improve the approach of Flanagan and Godefroid for DPOR in terms of optimality has been made by Gueta, Flanagan, Yahav, and Sagiv [GFYS07]. Known as cartesian partial order reduction (CPOR), their approach attempts to minimize context switches during the exploration of a state graph. This reduction of context switches is implemented by exploring multiple consecutive transitions by a process until a transition which is dependent with another process's transition has been explored. Thereby, optimality is obtained in the sense that no more than one representative of each Marzurkievicz trace is explored. Like our approach, CPOR allows to explore multiple transitions of a system at once. However, while our approach always explores paths completely until a deadlock is reached, CPOR interrupts the exploration after the first dependency is detected.

Recently, Abdulla, Aronis, Jonsson, and Sagonas have proposed a model checking algorithm based on DPOR with a similar optimality result [AAJS14]. In contrast to CPOR, their algorithm is stateless. Similar to the original DPOR algorithm by Flanagan and Godefroid, it reduces the state graph of a system by computing subsets of enabled transitions, but instead of using persistent sets, their algorithm uses *source sets*. In some cases, the source set of a state is smaller than the smallest possible persistent set of this state, which improves the state graph reduction. Whenever a dependency between two transitions is detected, backtracking is performed as in most DPOR algorithms. However, sleep set-blocked paths are avoided by storing path prefixes in *wake-up trees*. These path prefixes lead to a state where the two dependent transitions can be executed in reversed order. This avoids unnecessary explorations of states where the two transitions which have to be explored in reversed order are not enabled, for example. Again, one of the main differences to our algorithm is that only single transitions are explored, while our algorithm always explores a maximal path. Potentially, a similar technique to wake-up trees can be used to optimize our algorithm.

Most POR instances require the input system to be specified in a specific computation model or to satisfy certain provisos such as acyclicity of its state graph. For example, the soundness of the algorithm of [AAJS14] requires that transitions of the input system do not disable transitions from other processes. As these provisos often contain global properties of state graphs, such as acyclicity, it may be difficult to verify that all requirements are met if this guaranty is not provided through the use of a specific computation model. An approach to overcome the limitations of such requirements and provisos has been made by Bokor, Kinder, Serafini, and Suri [BKSS11]. Their approach, *local partial order reduction (LPOR)*, provides a POR technique which can be used for a large variety of transition systems as long as three relations which describe depen-

dencies between transitions are specified. In order to verify that these relations are specified correctly, it suffices to assert that they satisfy local conditions, in contrast to other POR techniques. It remains to be investigated how our approach can be transferred to general transition systems, for example transition systems with cyclic state graphs; in its current form, our approach is designed for transition systems which model terminating, concurrent programs. Unlike LPOR, our algorithm uses the information obtained from exploring maximal paths in order to calculate dependencies automatically.

Similar to LPOR, *peephole partial order reduction (PPOR)* by Wang, Yang, Kahlon, and Gupta [WYKG08] uses local constraints about paths in a state graph in order to describe dependency between transitions. As an instance of SMT/SAT-based bounded model checking [CBRZ01], it iterates over paths of increasing length k, which are described by a formula $\Phi(k)$. This formula is satisfiable if and only if there exists a path of length k which violates the property against which the system is checked. Redundant explorations are avoided by encoding scheduler choices in Φ which admit only one ordering for independent transitions. Due to the local description of dependence, PPOR adds only constraints about paths of a fixed length to Φ. The reduction obtained through PPOR is optimal for systems with no more than two processes, in the sense that exactly one representative of each Mazurkiewicz equivalence class is explored. However, PPOR cannot avoid redundant explorations for systems with more than two processes, in general.

Monotonic Partial Order Reduction (MPOR) by Kahlon, Wang, and Gupta [KWG09] improves PPOR by providing optimality even for systems with more than two processes. Similar to PPOR, MPOR is an SMT/SAT-based bounded model checking technique. Optimality is obtained by exploring exactly one *quasi-monotonic* interleaving for each Mazurkiewicz equivalence class (for paths of a given length) of a system. Similar to the execution semantics used by the CPOR approach, quasi-monotonic interleavings reduce the number of context switches and thereby prevent exploring different orderings of independent transitions. As MPOR, our approach encodes constraints about the ordering of dependent transitions for future exploration. However, our algorithm does not prohibit context switches as in quasi-monotonic interleavings, but allows the exploration of an arbitrary sequence which satisfies all constraints about dependency.

Chapter 2

Preliminaries

In this section, we introduce the basic definitions and concepts for LazyPOR. Our algorithm takes as input a transition system, where we require that input systems for our technique satisfy certain properties which intuitively express that the system models a terminating, distributed program. Executions of a transition system correspond to paths in its state graph, which we model as *transition sequences* started at a unique initial state. As we use dynamic dependencies between transitions, the fact whether two transitions are dependent may vary between different states of the transition system. We associate a dependency relation with every transition sequence which corresponds to a path in the state graph of a transition system. In the following, we give the exact definitions of transition systems, transition sequences, dependency, and transition systems for distributed programs.

2.1 Transition Systems

Our notion of transition systems is borrowed from Saissi et al. [SBM+13] which in turn is based on Attiya and Welch [AW04] and Bokor, Serafini, and Suri [BSS10]. The main characteristics of this notion are that transitions are deterministic and that there exists a unique initial state for every transition system. Both characteristics simplify the presentation of our technique. As our algorithm is stateless, we additionally require all transition systems to be acyclic.

Definition 1 (Transition system)**.** *A* transition system *is a triple* $TS = (S, s_0, T)$ *where* S *is a finite set of states,* $s_0 \in S$ *is the initial state of the system, and* T *is a finite set of transitions such that for all* $t \in T$*,* $t : S \rightharpoonup S$*, i.e., transitions are partial functions from* S *to* S*, and for all* $s_1, \ldots, s_{n+1} \in S$ *and any finite sequence* $t_1 \ldots t_n \in T$ *such that* $t_i(s_i) = s_{i+1}$*,* $s_1 \neq s_{n+1}$*.*

For a transition system (S, s_0, T), $t \in S$, and $s, s' \in S$. , whenever $t(s) = s'$ is defined, s' is called the *resulting state* of t at s and we write $s \xrightarrow{t_1} s'$. A transition $t \in T$ is *enabled* at a state $s \in S$ if there exists a resulting state of t at s. A state for which no transitions are enabled is called a *deadlock*. e write *enabled*(s) for the set of enabled transitions of s, i.e., *enabled*$(s) := \{t \in T : \exists s' \in S.s' = t(s)\}$.

2.2 Lists and Transition Sequences

We use (finite) lists to encode sequences of transitions, collections of dependencies between transitions, and collections of constraints. An element may occur multiple times in a list. Implicitly, a list is associated with a total order on its elements, and it is possible to access the element of a list at a specific position. We define lists defined by enumeration, constructed from a set and a total order, and element access as follows.

Definition 2 (Lists)**.**

Lists *A list is a totally ordered, finite collection of elements. We write* $[a_1, \ldots, a_n]$ *for the list of length* n *which contains the elements (in this order)* $a_1, \ldots, a_n$*. We write* ε *for the empty list.*

Lists from sets *Let* A *be a set with a subset* $A' \subseteq A$ *of cardinality* n *and let* $O \subseteq A \times A$ *be a total order on* A*. We write* $\mathit{list}(A', O)$ *for the list of length* n *which contains all members of* A' *in the order of* O *restricted to* A'*:*

$$\mathit{list}(A', O) := [a'_1, \ldots, a'_n],$$

where $A' = \{a'_1, \ldots, a'_n\} \wedge \forall i, j \in \{1, \ldots, n\}. i < j \Rightarrow (a'_i, a'_j) \in O$*.*

Element access *Let* $l := [a_1, \ldots, a_n]$ *be a list of length* n *and* $i \in 1, \ldots, n$*. We write* $l[i]$ *for the element of* l *at position* i*, i.e.,* $l[i] := a_i$*.*

The *range* of a list is the set of its valid positions, or indices, i.e., the positions where an element access is defined. For a list l of length n, we define

$$\mathit{range}(l) := \{1, \ldots, n\}.$$

Concatenation of lists and the prefix relation on lists are defined analogous to the corresponding operation and relation on words of a finite alphabet. Given two lists $l_1 = [a_1, \ldots, a_n]$, $l_2 = [a'_1, \ldots, a'_n]$, we write $l_1 \cdot l_2$ for the concatenation of l_1 and l_2, i.e.,

$$l_1 \cdot l_2 := [a_1, \ldots, a_n, a'_1, \ldots, a'_n].$$

For two lists l_1, l_2, we write

$$l_1 \sqsubseteq l_2 \text{ if } l_2 = l_1 \cdot l \text{ for some list } l.$$

Given a list and a prefix of this list, we introduce a notation which allows us to easily refer to the remaining part of the list which is not included in the prefix. Let l_1, l_2, l_3 be lists such that $l_3 = l_1 \cdot l_2$. We define ${}_{l_1}l_3$ as the part of l_3 which is not contained in l_1, i.e.,

$${}_{l_1}l_3 := l_2.$$

A transition sequence is defined as a list of transitions of a transition system.

Definition 3 (Transition sequence)**.** *Let* $TS := (S, s_0, T)$ *be a transition system. A* transition sequence (of TS) *is a list of transitions, i.e.,* $[t_1, \ldots, t_n]$ *for some* $t_1, \ldots, t_n \in T$*. The set* T^* *describes the set of all transition sequences of* TS*.*

For brevity, we also write $t_1 \cdots t_n$ for the list $[t_1, \ldots, t_n]$. Not all transition sequences u of a transition system correspond to paths in its state graph. The following notions allow us to refer to transition sequences which correspond to a path in a system's state graph, i.e., which intuitively describe (partial) executions of a system. Let (S, s_0, T) be a transition system with $s_1 \in S$ and $t_1, \ldots, t_n \in T$. We introduce the notation $s_1 \xrightarrow{t_1 \cdots t_n} s_{n+1}$ to describe the fact that there exist states $s_2, \ldots, s_{n+1} \in S$ such that $s_i \xrightarrow{t_i} s_{i+1}$ for all $1 \leq i \leq n$. Furthermore, if $s_1 \xrightarrow{u} s_2$ for some states s_1, s_2 and a transition sequence u, we write $u(s_1)$ to denote the state s_2, which is called the *resulting state of u at s_1*.

For a transition system (S, s_0, T) with some $s \in S$, the resulting state $u(s)$ of a transition sequence $u \in T^*$ is defined exactly for those transition sequences which correspond to paths in the system's state graph which begin at s. We call those transition sequences *feasible at s*.

Finally, we characterize those transition sequences which are feasible at a certain state of a transition system and lead to a deadlock.

Definition 4 (Maximal transition sequences at a state)**.** *Let s be a state. We define the set of* maximal transition sequences at s *as*

$$\mathit{max\text{-}seq}(s) := \{u : \exists s'.s \xrightarrow{u} s' \wedge \mathit{enabled}(s') = \varnothing\}.$$

A transition sequence $u \in \mathit{max\text{-}seq}(s)$ is called maximal at s.

Intuitively, maximal transition sequences at the initial state of a transition system which models a terminating, distributed program correspond to executions of the program from the beginning until either the program terminates or enters a state where no process can execute its next step.

2.3 Dependency and Traces

We borrow the definition of (static) dependency from [God95]. This notion serves as a basis for our definition of dynamic dependency.

Definition 5 (Dependency relation)**.** *Let (S, s_0, T) be a transition system. A binary, reflexive, and symmetric relation $D \subseteq T \times T$ is a* dependency relation *if for all t_1, $t_2 \in T$, $(t_1, t_2) \notin D$ implies that for all $s \in S$:*

- *if $s \xrightarrow{t_1} s'$ for some $s' \in S$, then t_2 is enabled in s if and only if t_2 is enabled in s' and*
- *if t_1 and t_2 are enabled in s, then there exists a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.*

Like any DPOR technique, our technique uses dynamic dependencies, which depend on the current state of a transition system. Therefore, we define *dynamic dependency relations* in order to distinguish cases where a pair of transitions is dependent at some state and independent at another state. We associate dependency of transitions with a transition sequence and state at which this transition sequence is started, rather than states alone. We use the information contained in the corresponding transition sequence to obtain a total order on dependency pairs of the sequence.

Definition 6 (Dynamic dependency relation)**.** *Let (S, s_0, T) be a transition system, let $u \in T^*$ be a transition sequence of length n, and $s_1 \in S$ be a state. A* dynamic dependency relation of w at s_1 *is a relation $D(u, s_1) \subseteq \{1, \ldots, n\}^2 \times S$ if $D(u, s_1)$ is a dependency relation on u, i.e., if for all $i_1, i_2 \in \{1, \ldots, n\}$, $t_1, t_2 \in T$, $t_1 = u[i_1]$ and $t_2 = u[i_2]$ and $(i_1, i_2) \notin D(u)$ implies that for all $s \in \{s_1, \ldots, s_{n+1}\} \subseteq S$ with $s_1 \xrightarrow{u[1]} s_2 \ldots s_n \xrightarrow{u[n]} s_{n+1}$:*

- *if $s \xrightarrow{t_1} s'$ for some $s' \in S$, then t_2 is enabled in s if and only if t_2 is enabled in s' and*
- *if t_1 and t_2 are enabled in s, then there exists a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.*

Dynamic dependency relations are only defined on transition sequences which are feasible at the given state. We omit the state where it is the initial state of a transition system or is otherwise clear from the context. In the rest of this thesis, we assume that a dynamic dependency relation for each transition sequence of a system is given. For each transition sequence u of a system, we denote this dynamic dependency relation by $\nparallel_u$. We abbreviate $(i_1, i_2) \in \nparallel_u$ by the corresponding infix notation $i_1 \nparallel_u i_2$ and write $i_1 \parallel_u i_2$ for $(i_1, i_2) \notin \nparallel_u$ if $i_1, i_2 \in \mathit{range}(u)$.

As described in chapter 1.1, traces are equivalence classes induced by a dependency relation. The partial order of a trace without transitivity corresponds to the dependency relation of any of its representatives. We use the following notation to denote the trace of a transition sequence.

Definition 7 (Trace of a transition sequence)**.** *Let $TS = (S, s_0, T)$ be a transition system and let $u \in T^*$. The* trace of u*, written $[u]_\simeq$, consists of all transition sequences of TS which are Mazurkiewicz equivalent to u, i.e.,*

$$[u]_\simeq := \{v \in T^* : u \simeq v\},$$

where $\simeq$ denotes Mazurkiewicz equivalence.

Our algorithm uses a reduced representation of a transition sequence's dependency relation in order to characterize traces. A *dependency list* of a transition sequence contains pairs of positions in that sequence where dependent transition are located. For each pair of dependent transitions in a transition sequence, a pair of positions is contained in the corresponding dependency list. This pair of transitions describes the order in which the two transitions occur in the sequence. Hence, a dependency list uniquely corresponds to a trace. In the following, $<_{\mathbb{N}\times\mathbb{N}}$ denotes the following linear order on pairs of natural numbers.

$$<_{\mathbb{N}\times\mathbb{N}} = \{((i_1, i_2), (j_1, j_2)) : i_1 < j_1 \vee i_1 = j_1 \wedge i_2 < j_2\}$$

Definition 8 (Dependency list of a transition sequence)**.** *Let w be a feasible transition sequence at some state s. We write $\mathit{dep}(w, s)$ for the* dependency list of w at s*, defined as*

$$\mathit{dep}(w, s) := \mathit{list}(\{(i_1, i_2) \in \mathit{range}(w)^2 : i_1 < i_2 \wedge i_1 \nparallel_w i_2\}, <_{\mathbb{N}\times\mathbb{N}}).$$

We omit the state of a dependency list and simply write $dep(u)$ for the dependency list of some transition sequence u where the corresponding state is clear from the context.

While dependency relations and dynamic dependency relations are reflexive and symmetric, the relation described by a dependency list is antisymmetric and irreflexive. This corresponds to the intention that a pair of positions in a dependency list should represent the order in which two transitions occur in the corresponding transition sequence.

Whenever the order of two dependent transitions in a transition sequence is reversed, a new transition sequence of a new trace is obtained. For example, for $u = t_1 \cdots t_{i_1} \cdots t_{i_2} \cdots t_n$ with $(i_1, i_2) \in dep(u)$, reversing $u[i_1]$ and $u[i_2]$ yields a new transition sequence $u' = t_1 \cdots t_{i_2} \cdots t_{i_1} \cdots t_n$ such that $[u]_{\simeq} \neq [u']_{\simeq}$.

In order to facilitate the presentation, we introduce a notation for the projection of a dependency list to those index pairs which correspond to a pair of transitions. Let u be a sequence and t_1, t_2 be transitions in u. The *dependency list of u projected to t_1, t_2* is defined as

$$u_{|t_1,t_2} := \{(i_1, i_2) \in dep(u) : (u[i_1] = t_1 \wedge u[i_2] = t_2) \vee (u[i_1] = t_2 \wedge u[i_2] = t_1)\}$$

Formally, we define the reversed occurrence of two dependent transitions as follows.

Definition 9 (Swapped dependency)**.** *Two transition sequences u, v have a* swapped dependency*, written swapped-dep(u, v), if*

$$\exists t_1, t_2 \in u.\, \exists i \in range(u_{|t_1,t_2}).\, \exists (i_1, i_2) \in range(u)^2.\, \exists (j_1, j_2) \in range(v)^2.\\ u_{|t_1,t_2}[i] = (i_1, i_2) \wedge v_{|t_1,t_2}[i] = (j_1, j_2) \wedge u[i_1] = v[j_2]$$

Swapped dependencies are used in our algorithm to schedule new paths for exploration. The converse of the above mentioned observation states that whenever two transition sequences are not equivalent, they have a swapped dependency. This statement holds for the class of transition systems which we consider for our algorithm and define in the following section. The relationship between swapped dependencies and Mazurkiewicz equivalence is an important part of our correctness proof and described in more detail in section 3.2.

2.4 Transition Systems for Distributed Programs

Our algorithm is specifically designed for transition systems which model distributed programs and correspondingly uses assumptions about transition systems in order to optimize the exploration of the state graph. We require that transitions cannot disable other transitions and call transition systems which satisfy this requirement *separated*. When modeling a distributed program, this requirement can be satisfied in the following manner. Each statement of the program is modeled by a single transition sequence. A transition t which corresponds to some statement *stmt* is enabled at a state s if and only if the transition sequence which corresponds to the preceding statement of *stmt* has s as a resulting state, or if t models an initial statement of the program and s is the initial state of the transition system.

Formally, we define separated transition systems as follows.

Definition 10 (Separated transition system)**.** *A transition system* (S, s_0, T) *is a* separated transition system *if*

$$\forall t, t' \in T. \forall s, s' \in S. s \xrightarrow{t} s' \wedge t' \in \mathit{enabled}(s) \wedge t' \notin \mathit{enabled}(s') \Rightarrow t = t'.$$

A similar requirement has been made by Abdulla, Aronis, Jonsson, and Sagonas [AAJS14] by assuming that a process cannot disable another process.

Additionally, input transition systems may assign a process to each transition. Such an assignment helps to reduce the computations necessary for the choice of paths to be explored. However, it is not necessary to assign processes to transitions in order to apply our algorithm to a transition system. Therefore, we simplify the presentation of our algorithm and refrain from using processes. Nevertheless, we give a detailed description of how to optimize our algorithm by the use of processes in section 4.1.

2.5 Using Constraint Systems for POR

In this section, we introduce *trace constraint systems* as a characterization of traces. By alternating the constraints of a trace constraint system, it is possible to generate a new trace constraint system representing another trace than the original trace constraint system. Trace constraint systems correspond to dependency graphs as defined by Mazurkiewicz [Maz86]. We provide a correspondence proof between trace constraint systems and dependency graphs, which we use to justify that trace constraint systems are an accurate representation of traces. As observed by Godefroid [God95], it suffices to explore one representative of each trace in order to detect all deadlocks of a transition system, which constitutes the basis for the correctness of our algorithm.

2.5.1 A Constraint System Representation of Traces

A trace constraint system consists of a set of variables, a labeling function assigning a transition to each variable, and a list of constraints, which are pairs of variables. A variable α represents the transition it is labeled with. A constraint (α_1, α_2) expresses that the transition represented by α_1 appears before the transition represented by α_2 in all representatives of the trace described by the trace constraint system. Formally, we define trace constraint systems as follows.

Definition 11 (Trace constraint system)**.** *Given a transition system* (S, s_0, T)*, a* trace constraint system *is a triple* $CS = (A, C, l)$ *where*

- A *is a finite set*
- $l : A \to T$ *is a function which labels the elements of* A *with transitions*
- C *is a list of pairs* $(\alpha_1, \alpha_2) \in A \times A$

We write $CS_\varnothing$ *for the empty constraint system* $(\varnothing, \varnothing, \varnothing)$*.*

For every trace constraint system (A, C, l) where C represents an acyclic relation, (A, C) is a directed acyclic graph and can be extended to a unique strict partial order (by taking its transitive closure). We use this property to identify

partial orders of traces. Our definition of trace constraint systems corresponds to the definition of dependency graphs by Mazurkiewicz [Maz86]; however, a few differences exist. First, dependency graphs are defined for static dependencies while we use dynamic dependencies in trace constraint systems. Second, we allow trace constraint systems with cyclic constraints (which are unsatisfiable and do not correspond to a trace) whereas dependency graphs are acyclic by definition. Third, a trace constraint system potentially represents a trace which does not contain any feasible sequence for our definition of trace constraint systems, while no notion of enabled or disabled transitions is associated with dependency graphs. Fourth, trace constraint systems contain a totally ordered representation of constraints in a sequence, in contrast to dependency graphs, which do not contain an order on nodes or edges.

The additional information contained in trace constraint systems in comparison to dependency graphs is used by our algorithm to generate new trace constraint systems by alternating constraints. The basis for these alternations is a trace constraint system generated from a given transition sequence. Those trace constraint systems respect the dynamic dependency relation of the given transition sequence and correspond to dependency graphs constructed from a transition sequence.

Definition 12 (Trace constraint system of a transition sequence)**.** *Given a transition system* (S, s_0, T) *and a transition sequence* $w = t_1 \dots t_n \in T^n$ *feasible at some* $s \in S$*, the* trace constraint system of w at s *is defined as* $CS(w, s) = (A, C, l)$ *where*

- $A = \{\alpha_1, \dots, \alpha_n\}$
- $l(\alpha_i) = t_i$ *for all* $1 \leq i \leq n$
- $C = list(\{(\alpha_i, \alpha_j) \in A \times A : (i, j) \in dep(w, s)\}, <_{\mathbb{N}\times\mathbb{N}})$

We denote by $CS(TS)$ *the set* $\{CS(w, s) : w \in T^* \wedge s \in S\}$ *of trace constraint systems constructed out of all transition sequences of* TS*.*

We omit the state of a trace constraint system where it is clear from the context. Clearly, the construction described in definition 12 yields a trace constraint system (according to definition 11).

A trace constraint system CS' constructed from another trace constraint system CS by removing constraints is called a *prefix* of CS. Removing constraints from the end of the list of constraints of CS may be useful if CS is unsatisfiable, but a prefix of CS is satisfiable. Formally, $CS' = (A', C', l')$ is a prefix of $CS = (A, C, l)$, written $CS' \sqsubseteq CS$, if

$$A' \subseteq A \wedge C' \sqsubseteq C \wedge \forall \alpha \in A'.l'(\alpha) = l(\alpha).$$

In order to obtain a representative of the trace represented by a trace constraint system, it suffices to generate a solution, i.e., find a total ordering on the variables of the corresponding trace constraint system such that all constraints are met. The transition sequence corresponding to the labels of the ordered variables is a representative of the trace. We use functions from variables to natural numbers in order to describe such a total order on variables.

Definition 13 (Solution)**.** *Given a trace constraint system $CS = (A, C, l)$, a* solution *to CS is an injective (1-to-1) function $\sigma : A \to \{1, \ldots, |A|\}$ such that for all $(\alpha_1, \alpha_2) \in C : \sigma(\alpha_1) < \sigma(\alpha_2)$.*

Any solution to a trace constraint system is necessarily bijective as the cardinalities of its domain and codomain are equal. A solution to a trace constraint system corresponds to a transition sequence in the following way. Let σ be a solution of a trace constraint system $CS = (\{\alpha_1, \ldots, \alpha_n\}, C, l)$. Then σ corresponds to the transition sequence $u = l(\sigma^{-1}(1)) \cdots l(\sigma^{-1}(n))$ where σ^{-1} is the inverse of σ, which exists since σ is necessarily a bijection. We call u a *solution sequence of CS* and write $SOL(CS)$ for the set of all solution sequences of CS.

2.5.2 Selective Search

In the following, we establish a correspondence between the trace constraint system of a transition sequence and the trace of a transition sequences. Based on this correspondence, we argue that trace constraint systems provide a valid representation of traces for a sound selective search. As trace constraint systems are similar to dependency graphs, we first establish a correspondence between the trace constraint system of a transition sequence and the dependency graph of a transition sequence. In a second step, we use Mazurkiewicz's result that the algebra of dependency graphs is isomorphic to the algebra of traces. This result can be translated to trace constraint systems and validates our representation of traces. Finally, by using an observation by Godefroid, we argue that trace constraint systems are suitable for sound deadlock detection.

We begin by defining isomorphism between trace constraint systems and dependency graphs.

Definition 14 (Isomorphism between trace constraint systems and dependency graphs)**.** *Let $CS = (A, C, l)$ be a constraint system, and $G = (V, R, \varphi)$ be a dependency graph. CS and G are isomorphic, written $CS \cong_{CS\text{-}G} G$, if and only if there exists an injective (1-to-1) function $f : A \to V$ such that*

1. *for all $\alpha \in A$, $l(\alpha) = \varphi(f(\alpha))$ and*
2. *for all $\alpha, \alpha' \in A$, $(\alpha, \alpha') \in C \Leftrightarrow (f(\alpha), f(\alpha')) \in R$*

Based on the definition of isomorphism from definition 14, the following lemma establishes a correspondence between trace constraint systems of transition sequences and dependency graphs of transition sequences.

Lemma 2.1 (Isomorphism between trace constraint systems and dependency graphs)**.** *For all transition sequences w, $CS(w) \cong_{CS\text{-}G} \langle w \rangle$.*

Proof. By induction on the length of w.

- case $w \in T^0$, i.e., w is the empty word

  We have $CS(w) = CS_\varnothing$ and $\langle w \rangle = (\varnothing, \varnothing, \varnothing)$. Clearly, $CS(w)$ and $\langle w \rangle$ are isomorphic.

- case $w = u \cdot t$, where $u \in T^n$, $n \geq 0$, and $t \in T$

  Consider $CS(u) = (A_u, C_u, l_u)$ for some A_u, C_u, l_u. Then $A_u = \{\alpha_1, \ldots, \alpha_n\}$ according to definition 12. Similarly, consider

$\langle u \rangle = (V_u, R_u, \varphi_u)$ for some V_u, R_u, φ_u. By the induction hypothesis, $CS(u) \simeq \langle u \rangle$, i.e., there exists an injective function $f_u : A_u \to V_u$ such that for all $\alpha \in A_u$, $l(\alpha) = \varphi(f(\alpha))$ and for all $\alpha, \alpha' \in A_u$, $(\alpha, \alpha') \in C_u \Leftrightarrow (f(\alpha), f(\alpha')) \in R_u$.

By definition, $\langle u \cdot t \rangle = (V_u \cup \{v_{n+1}\}, R_u \cup \{(v, v_{n+1}) : v \in V_u \text{ and } (\varphi(v), \varphi(v_{n+1})) \in D\}, \varphi_u \cup \{v_{n+1} \mapsto t\})$ for some node v_{n+1}.

It is easy to see that $CS(u \cdot t)$ analogously includes (in a component-wise manner) $CS(u)$; we have $CS(u \cdot t) = (A_u \cup \{\alpha_{n+1}\}, C_u \cup \{(\alpha_i, \alpha_{n+1}) : i < n+1 \text{ and } (i, n+1) \in D(u \cdot t)\}, l_u \cup \{\alpha_{n+1} \mapsto t\})$.

We extend f_u to an injective function $f : A \to V$ in order to obtain $CS(u \cdot t) \simeq \langle u \cdot t \rangle$. Let $f = f_u \cup \{\alpha_{n+1} \mapsto v_{n+1}\}$. It remains to show that f is an injective function and that conditions 1 and 2 of definition 14 are satisfied. As f_u is injective and $f_u : A_u \to V_u$ (i.e., v_{n+1} is not in the image of f_u), f is injective as well.

We now prove that for all $\alpha \in A$, $l(\alpha) = \varphi(f(\alpha))$ and for all $\alpha_i, \alpha_j \in A$, $(\alpha_i, \alpha_j) \in C \Leftrightarrow (f(\alpha_i), f(\alpha_j)) \in R$. Let $\alpha, \alpha_i, \alpha_j \in A$.

- case $\alpha_i, \alpha_j \in A_u$
  We have $l(\alpha) = \varphi(f(\alpha))$ and $(\alpha_i, \alpha_j) \in C \Leftrightarrow (f(\alpha_i), f(\alpha_j)) \in R$ by the induction hypothesis.
- case $\alpha_j \notin A_u$
  We have $j = n+1$ and hence $i < j$. Furthermore, $\alpha_i \in A_u$. As $\alpha_i \in A_u$, $l(\alpha_i) = \varphi(f(\alpha))$ by the induction hypothesis. By definition of f and by the constructions of $CS(u \cdot t)$ and $\langle u \cdot t \rangle$, $l(\alpha_{n+1}) = \varphi(f(\alpha_{n+1}))$.
  - case $(\alpha_i, \alpha_j) \in C$
    By the construction of $CS(u \cdot t)$, $(i, n+1) \in D$. Hence, $l(\alpha_i), l(\alpha_{n+1})$ are dependent in $u \cdot t$ and by the construction of $\langle u \cdot t \rangle$, $(f(\alpha_i), f(\alpha_{n+1})) \in R$.
  - case $(\alpha_i, \alpha_j) \notin C$
    By the construction of $CS(u \cdot t)$, $(i, n+1) \notin D$. Hence, $l(\alpha_i), l(\alpha_{n+1})$ are independent in $u \cdot t$ and by the construction of $\langle u \cdot t \rangle$, $(f(\alpha_i), f(\alpha_{n+1})) \notin R$.
- case $\alpha_i \notin A_u$
  We have $\alpha_i = \alpha_{n+1}$. Due to the condition $i < j$ in the definition of C, $(\alpha_i, \alpha_j) \notin C$. As $f(\alpha_{n+1}) = v_{n+1} \notin V_u$, $(f(\alpha_i), f(\alpha_j)) \notin R$ by the construction of $CS(u \cdot t)$. □

Mazurkiewicz provides a correspondence between dependency graphs and traces [Maz86, theorems 1 to 3]. Theorem 1 by Mazurkiewicz states that two transition sequences w_1 and w_2 are equivalent if and only their dependency graphs are isomorphic.

$$[w_1] = [w_2] \Leftrightarrow \langle w_1 \rangle \cong_G \langle w_2 \rangle,$$

where $\cong_G$ is the isomorphism on dependency graphs. Using lemma 2.1, we can translate this result to a correspondence between trace constraint systems and

traces.

$$[w_1] = [w_2] \Leftrightarrow CS(w_1) \cong_{CS} CS(w_2),$$

where $\cong_{CS}$ is the isomorphism on trace constraint systems. By theorem 2 by Mazurkiewicz, each linear extension of the partial order induced by a dependency graph is a representative of the corresponding trace. Similarly, a solution of a trace constraint system describes a representative of the corresponding trace. Finally, theorem 3 establishes that dependency graphs preserve the semantics of traces, which we reproduce slightly abbreviated. Again, theorem 3 can be translated for trace constraints systems.

Theorem 2.1 ([Maz86, theorem 3])**.** *Given a transition system, the algebra of dependency graphs is isomorphic to the algebra of traces.*

The following corollary states that the set of solutions of a trace constraint system is equal to the corresponding trace. It enables us to use trace constraint systems in a state search in order to obtain representatives of traces.

Corollary 2.2. *For all transition sequences w, $[w]_{\simeq} = SOL(CS(w))$.*

Proof. Follows from theorem 2 by Mazurkiewicz [Maz86]. □

Godefroid has shown that it is sufficient for deadlock detection to explore only one representative of each trace [God95]. Fundamental to this observation is the following theorem.

Theorem 2.2 ([God95, theorem 3.10])**.** *Let s be a state of a transition system. If $s \xrightarrow{w_1} s_1$ and $s \xrightarrow{w_2} s_2$ and $[w_1]_{\simeq} = [w_2]_{\simeq}$, then $s_1 = s_2$.*

Furthermore, since only maximal transition sequences lead to a deadlock (by definition), it is sufficient to only explore one representative of each trace corresponding to a maximal transition sequence in order to reach all deadlocks of a system. By corollary 2.2, it is sound to explore one solution sequence of each trace constraint system $CS \in CS(TS)$ for some transition system TS, instead of one representative of each trace of TS, in order to detect all deadlocks of TS. Furthermore, a state search exploring each trace of a transition system is sound for any linear time logic property without the *next* operator [BK08]. The correctness result for our algorithm is based on this observation.

Chapter 3

POR Using Constraints

In this section, we present our DPOR algorithm LazyPOR for state search in separated transition systems. The algorithm makes use of trace constraint systems to generate transition sequences with alternated dependencies. We start by presenting the algorithm and describing its functionality, before we establish and prove a corresponding correctness condition.

3.1 LazyPOR

The main procedure of the algorithm is explore, which uses two subroutines execute and solve. A transition system $TS = (S, s_0, T)$ can be explored by invoking the algorithm with two empty transition sequences as arguments, i.e., by executing explore$(\varepsilon, \varepsilon)$. This call explores a representative of every trace which starts at s_0 and ends at a deadlock.

During the exploration of the state graph, all recursive calls to the main procedure explore are performed with two transition sequences u_1 and u_2 as arguments such that $s_0 \xrightarrow{u_1} s'$ for some s' and u_2 is maximal at s'. Hence, $u_1 \cdot u_2$ is a maximal transition sequence at the initial state s_0. Intuitively, u_2 is a transition sequence which has been explored and which contains dependencies that need to be swapped in order to explore alternative traces. The first argument u_1 describes the path in the state graph to the state s' at which u_2 is maximal. All dependencies in u_1 have already been swapped and the corresponding transition sequences are already covered. Therefore, the main procedure explore creates a trace constraint system for each combination of swapped and original dependencies in the dependency list $dep(u_2)$, but not for dependencies of u_1. Each of these trace constraint systems is passed to execute, which creates a solution (if possible) via solve and executes the solution.

Before describing the algorithm in more detail, we introduce some auxiliary notations which occur in the pseudo code of the algorithm. For some transition sequence u, we denote by $variations(u, s)$ the set of trace constraint systems

which can be obtained by swapping one or more constraints in $CS(u, s)$, i.e.,

$$\begin{aligned} \mathit{variations}(u, s) := \{c : \mathit{CS}(u, s) = (A, C_u, l) \wedge c = (A, C, l) \\ \wedge \mathit{range}(C) = \mathit{range}(C_u) \\ \wedge (\forall i \in \mathit{range}(C).C[i] = C_u[i] \\ \vee (\exists \alpha_1, \alpha_2 \in A.\, C[i] = (\alpha_2, \alpha_1) \wedge C_u[i] = (\alpha_1, \alpha_2)))\}. \end{aligned}$$

In our algorithm, *variations*() is always used at the state $u_1(s_0)$, where u_1 is the first argument of explore, and we write *variations*(u) to denote *variations*$(u, u_1(s_0))$.

We call the feasible prefixes of the solution sequences for the trace constraint systems in *variations*(u, s) the *variations of* u *at* s. Intuitively, u corresponds to a path beginning at s in the state graph of a transition system and the variations of u correspond to alternative paths beginning at s, which cover all remaining traces of the transition system in the following sense. Whenever a path beginning at s corresponds to a different trace than u, a variation of u exists which corresponds to a prefix of this path.

During an execution of explore(u_1, u_2) with $u_2 \neq \varepsilon$, all trace constraint systems in *variations*(u_2) are generated and their feasible solutions are explored. In case a trace constraint system CS is unsatisfiable, the algorithm attempts to find a partial solution by generating and solving a prefix of CS. In order to satisfy as many constraints as possible, a *maximal satisfiable prefix of* CS is generated in the sense that no other satisfiable prefix of CS has a longer list of constraints. As the set of solutions is equal for all maximal satisfiable prefixes of CS, we use a function *max-satisfiable*$_{CS}(s)$ to denote an arbitrary maximal satisfiable prefix of CS at state s.

Finally, we use the function *split-at-dependency* which helps to separate dependencies from a dependency list into those dependencies which still need to be swapped for a new trace constraint system and those which are already covered. We provide the definition of *split-at-dependency* below, in the description of the algorithm.

Description of the algorithm As described above, the main procedure explore takes as arguments two transition sequences u_1 and u_2 such that $u_1 \cdot u_2$ is maximal at s_0 and u_1 contains only covered dependencies (which do not need to be swapped), while all variations of u_2 need to be explored. The main procedure consists of a check of the second argument, u_2, (lines 5–10) and a loop over all constraint systems in *variations*(u_2) (lines 11–14). The check of the second argument is only required initially, when explore is called the first time, and for recursive calls which do not trigger a further exploration. In the first case, when explore is called the first time, execute returns in line 6 a pair of transition sequences (u_1', u_2') such that u_1' is empty and u_2' is maximal at s_0. The parameters u_1 and u_2 are overwritten with the return values u_1' and u_2'. If the transition system contains no transition, u_2 is still empty and the algorithm returns in line 8. Otherwise, the algorithm continues with the exploration of all variations of w in the following loop. In the second case, where u_2 is empty during a call of explore which does not trigger a further exploration, u_1 is a maximal transition sequence at s_0. Therefore, execute does not find new dependencies which need to be swapped (since u_1 only contains covered dependencies) and returns u_1

```
1  // initially: u_1 = ε, u_2 = ε
2  // explores a representative of every trace beginning at s_0
3  // u_2 has to be either empty or a maximal transition sequence at u_1(s_0)
4  explore(u_1, u_2) {
5    if (u_2 = ε) {
6      (u_1, u_2) := execute(u_1, CS_∅)
7      if (u_2 = ε) {
8        return
9      }
10   }
11   for (c ∈ variations(u_2, u_1(s_0))) {
12     (u'_1, u'_2) := execute(u_1, c)
13     explore(u'_1, u'_2)
14   }
15 }
16
17 // executes a maximal transition sequence v_1·v_2 beginning at u_1(s_0) such that
       v_1 complies with trace constraint system c as long as possible
18 // returns a pair of transition sequences (u'_1, u'_2) such that all dependencies in u'_1
       are covered and all dependencies of u'_2 need to be swapped.
19 execute(u_1, c) {
20   v_1 := solve(u_1, c)
21   s := u_1(s_0)
22   for (t ∈ v_1) {
23     if (t ∈ enabled(s)) {
24       s := t(s)
25     } else {
26       remove all unexplored transitions from v_1
27       break
28     }
29   }
30   v_2 := ε
31   while (enabled(s) ≠ ∅) {
32     t := enabled(s).pop()
33     v_2 := v_2·t
34     s := t(s)
35   }
36   (u'_1, u'_2) := split-at-dependency(u_1, v_1, v_2, c)
37   return (u'_1, u'_2)
38 }
39
40 // returns a solution sequence of the maximal satisfiable prefix of c
41 solve(u_1, c) {
42   return some solution sequence v of max-satisfiable_c(u_1(s_0))
43 }
```

Listing 3.1: The LazyPOR algorithm.

unchanged along with an empty transition sequence. The condition in line 7 is satisfied and the algorithm terminates.

In line 11, the trace constraint systems of *variations*$(u_2, u_1(s_0))$ are generated. The state $u_1(s_0)$ at which *variations*() is calculated corresponds to the state at which u_2 has previously been executed. For each trace constraint system c in *variations*$(u_2, u_1(s_0))$, execute(u_1, c) executes a maximal transition sequence w at $u_1(s_0)$ which complies with c as long as possible. It returns two transition sequences u_1' and u_2' such that $w = u_1' \cdot u_2'$. Intuitively, u_1' contains only dependencies which do not need to be swapped because every variation of u_1' is already covered. In contrast, all variations of u_2' still need to be explored, which is done by the recursive call in line 13. We continue by describing execute in more detail.

The subroutine execute takes as argument a transition sequence u_1 and a trace constraint system c. The transition sequence u_1 corresponds to the first argument of explore. It describes its resulting state (the state s' such that $s_0 \xrightarrow{u_1} s'$) from which a new path will be explored. The trace constraint system c indicates a new path beginning at s', which is an alternative path to u_2, the second argument of explore whose execution calls execute. Beginning at s', execute executes a maximal transition sequence $v_1 \cdot v_2$ divided into two transition sequences v_1 and v_2. Here, v_1 is the longest feasible prefix of a solution sequence of the maximal satisfiable prefix of c. Let s'' be the resulting state of v_1, i.e., $s' \xrightarrow{v_1} s''$. Then v_2 is a maximal transition sequence at v_2. Thereby, execute always reaches a deadlock. Additionally to indicating s', the transition sequence u_1 describes the transitions which lead to s'. This information is used to detect new dependencies between u_1 and $v_1 \cdot v_2$ which arose from swapping dependencies of u_2.

In line 20, execute begins by calling solve, which returns a solution sequence of the maximal satisfiable prefix of c at s'. Line 21 fetches s' from u_1. The loop in lines 22–29 executes the longest feasible prefix of v_1. If v_1 is not feasible at s', the first disabled transition of v_1 and all following transitions are removed from v_1 in line 26. Hence, after the loop of lines 22–29, the variable v_1 holds a feasible transition sequence which leads to the state s'', i.e., $s' \xrightarrow{v_1} s''$. Lines 30–35 execute, beginning at s'', an arbitrary maximal transition sequence, which is stored as v_2. Finally, the transition sequence $u_1 \cdot v_1 \cdot v_2$ is split in two transition sequences u_1' and u_2' by *split-at-dependency*. The pair (u_1', u_2') is returned and used by explore as the argument of a recursive call.

The function *split-at-dependency* is used to separate the transition sequence $u_1 \cdot v_1 \cdot v_2$ into to transition sequences u_1' and u_2' such that

- u_1' contains only covered dependencies,
- all dependencies in u_2' have still to be swapped, and
- no dependencies between u_1' and u_2' exist, in the sense that for all $(i_1, i_2) \in dep(u_1' \cdot u_2')$, i_1 and i_2 are either both contained in u_1' or in u_2'.

Satisfying these conditions, the return values u_1' and u_2' of *split-at-dependency* can be used as arguments to explore.

In order to specify *split-at-dependency*, we define the following auxiliary function, which divides the transition sequence $u_1 \cdot v_1 \cdot v_2$ into its three components u_1, v_1, and v_2. We call these components section 1, 2, and 3. For all

$i \in \mathit{range}(u_1 \cdot v_1 \cdot v_2)$, we define $\mathit{section}(i)$ as follows.

$$\mathit{section}(i) := \begin{cases} 1 & \text{if } i \in \{1, \ldots, \mathit{length}(u_1)\} \\ 2 & \text{if } i \in \{\mathit{length}(u_1) + 1, \ldots, \mathit{length}(u_1 \cdot v_1)\} \\ 3 & \text{if } i \in \{\mathit{length}(u_1 \cdot v_1) + 1, \ldots, \mathit{length}(u_1 \cdot v_1 \cdot v_2)\} \end{cases}$$

The function *split-at-dependency* can be specified as follows. It takes as arguments the three transition sequences u_1, v_1, and v_2 as described above and the constraint system which lead to v_1. It returns a pair of transition sequences,

$$\mathit{split\text{-}at\text{-}dependency}(u_1, v_1, v_2, (A, C, l)) := (u_1', u_2'),$$

such that the following holds.

$$(u_1' \sqsubseteq u_1 \cdot v_1) \wedge (u_1' \cdot u_2' = u_1 \cdot v_1 \cdot v_2) \tag{3.1}$$
$$\wedge \forall (i_1, i_2) \in \mathit{dep}(u_1 \cdot v_1 \cdot v_2). \tag{3.2}$$
$$((\mathit{section}(i_1) \neq \mathit{section}(i_2) \tag{3.3}$$
$$\vee \mathit{section}(i_2) = 3) \tag{3.4}$$
$$\Rightarrow i_1 \notin \mathit{range}(u_1')) \tag{3.5}$$
$$\wedge \forall (j_1, j_2) \in \mathit{dep}(v_1). ((j_1, j_2) \notin C \Rightarrow j_1 \notin \mathit{range}(u_1')) \tag{3.6}$$

The conditions in line 3.1 ensure that

- u_1' and u_2' partition the transition sequence $u_1 \cdot v_1 \cdot v_2$, but do not otherwise change it, and
- u_1' cannot contain transitions from v_2, whose dependencies do not have been swept yet.

Additionally, whenever a dependency pair $(i_1, i_2) \in \mathit{dep}(u_1 \cdot v_1 \cdot v_2)$ extends to multiple sections (line 3.3), it has not yet been covered and therefore must not be included in $\mathit{dep}(u_1')$ (line 3.5). Likewise, any dependency pair which has at least one index from section 3 (line 3.4), must not be included in $\mathit{dep}(u_1')$. Finally, all dependency pairs $(j_1, j_2) \in \mathit{dep}(v_1)$ which do not occur as constraints in the constraint system leading to v_1 (line 3.6) must equally not be included in $\mathit{dep}(u_1')$.

This concludes the description of our algorithm. We continue by showing its correctness in the following section.

3.2 Correctness

Based on the correspondence result of section 2.5.2, which allows us to uniquely represent traces by trace constraint systems, we prove the correctness of our algorithm. The correctness result states that whenever a state search by our algorithm has terminated, every transition sequence which is maximal at the initial state is equivalent to a transition sequence which has been explored.

We begin by formalizing the observation from section 2.3 about the relationship between swapped dependencies and Mazurkiewicz equivalence. In a separated transition system, no transition can disable another transition. Therefore, every two nonequivalent transition sequences have a swapped dependency. This

property of separated transition systems enables us to generate a representative of every trace of a separated transition system by starting with an arbitrary maximal transition sequence and gradually swapping dependencies, as it is done in our algorithm when variations are calculated.

Lemma 3.1 (Existence of swapped dependency)**.** *Let u, v be maximal transition sequences at some state s of a separated transition system. Then u and v are Mazurkiewicz equivalent if and only if they contain a swapped dependency, i.e., $[u]_\simeq \neq [v]_\simeq \Leftrightarrow$ swapped-dep(u, v).*

Proof. From right to left: *swapped-dep*(u, v) implies that the dynamic dependencies of u and v differ. Hence, u and v are not equivalent.

From left to right: Let s be a state of a separated transition system TS and let u, v be transition sequences in *max-seq*(s) such that $[u]_\simeq \neq [v]_\simeq$. Induction over the length of $u \cdot v$.

- base case $u = v = \varepsilon$

  We have $[u]_\simeq = [v]_\simeq$, contradiction.

- induction step

  Choose the longest prefixes $u_1 \sqsubseteq u, v_1 \sqsubseteq v$ such that $[u_1]_\simeq = [v_1]_\simeq$. Then there exists a state s' such that $s \xrightarrow{u_1} s' \wedge s \xrightarrow{v_1} s'$.

  1. case $u_1 \neq \varepsilon \vee v_1 \neq \varepsilon$
     We have ${}_{u_1}u \in$ *max-seq*(s'), ${}_{v_1}v \in$ *max-seq*(s'), and $[{}_{u_1}u_1]_\simeq \neq [{}_{v_1}v_1]_\simeq$. By the induction hypothesis, *swapped-dep*$({}_{u_1}u, {}_{v_1}v)$. Because $[u_1]_\simeq = [v_1]_\simeq$, u_1 and v_1 do not have a swapped dependency. Hence, we have *swapped-dep*(u, v).
  2. case $u_1 = \varepsilon \wedge v_1 = \varepsilon$
     Both u and v cannot be empty as the length of $u \cdot v$ is greater than zero and $u, v \in$ *max-seq*(s). Let t_u be the first transition of u and let t_v be the first transition of v. As both u and v are maximal in s and TS is separated, $t_v \in u$ and $t_u \in v$.
     - case $t_u \not\parallel_u t_v \wedge t_v \not\parallel_v t_u$
       As t_u is the first transition of u and $t_u \not\parallel_u t_v$, we have $u_{|t_u,t_v}[0] = (i_1, i_2)$ such that $u[i_1] = t_u$ and $u[i_2] = t_v$. Analogously, we have $v_{|t_u,t_v}[0] = (j_1, j_2)$ such that $v[j_1] = t_v$ and $v[j_2] = t_u$. Hence, *swapped-dep*(u, v).
     - otherwise
       Without loss of generality, we assume $t_u \parallel_u t_v$. We construct u' by swapping t_u and t_v in u. Then $[u']_\simeq = [u]_\simeq$ and the first transition of u' and v is equal. Hence, there exists a state s'' such that $s \xrightarrow{t_v} s''$ and ${}_{t_v}u', {}_{t_v}v \in$ *max-seq*(s''). Analogous to case 1, the goal holds for ${}_{t_v}u'$ and ${}_{t_v}v$ by the induction hypothesis.
       As we constructed u'' by swapping only independent transitions, $[u']_\simeq = [u]_\simeq$. Therefore the goal also holds for u and v. □

We develop our correctness condition as follows. In order to provide correctness, this condition should guarantee that representatives of all traces of a separated transition system have been explored if the algorithm is called with

two empty transition sequences as arguments. However, in order to facilitate the proof of the correctness theorem, we adapt the correctness condition to the following property of our algorithm. For all algorithm arguments u_1 and u_2 such that $u_1 \cdot u_2$ is maximal at the initial state of a separated transition system, our algorithm explores representatives of all traces $[v]_\simeq$ such that all swapped dependencies of $u_1 \cdot u_2$ and v do not occur in u_1. In order to formalize this statement, we formally define swapped dependencies as triples (i, t_1, t_2) where t_1, t_2 are dependent transitions in two transition sequences u, v and i is a position in $u_{|t_1,t_2}$ and $v_{|t_1,t_2}$ such that $u_{|t_1,t_2}[i]$ and $v_{|t_1,t_2}[i]$ indicate a reversed occurrence of t_1, t_2 in v relative to u. Let u, v be transition sequences. The *set of swapped dependencies of u and v* is defined as follows.

$$\begin{aligned} \mathit{swapped\text{-}set}(u, v) := \{(i, t_1, t_2) : i \in (\mathit{range}(u_{|t_1,t_2}) \cap \mathit{range}(v_{|t_1,t_2})) \\ \wedge \exists i_1, i_2.\, \exists j_1, j_2.\, u_{|t_1,t_2}[i] = (i_1, i_2) \wedge v_{|t_1,t_2}[i] = (j_1, j_2) \\ \wedge u[i_1] = v[j_2] \wedge u[i_2] = v[j_1]\} \end{aligned}$$

The above definition allows us to reformulate the property relevant for correctness more precisely. A call to $\mathsf{explore}(u_1, u_2)$ such that $u_1 \cdot u_2$ is maximal explores equivalent transition sequences to all v where for all $(i, t_1, t_2) \in \mathit{swapped\text{-}set}(u_1 \cdot u_2, v)$, $(u_1 \cdot u_2)_{|t_1,t_2}[i] = (i_1, i_2)$ implies that the dependency (i_1, i_2) is contained in u_2, i.e., $i_1 \notin \mathit{range}(u_1)$. Built on this property, we formulate our correctness condition.

Theorem 3.1 (Correctness)**.** *Let s_0 be the initial state of a separated transition system and let u, w be maximal transition sequences at s_0. Let $u =: u_1 \cdot u_2$ for some transitions sequences u_1 and u_2 such that for all $(i, t_1, t_2) \in \mathit{swapped\text{-}set}(u, w)$, $u_{|t_1,t_2}[i] = (i_1, i_2)$ implies $i_1 \notin \mathit{range}(u_1)$. If $\mathsf{explore}(u_1, u_2)$ has terminated, there exists a representative $v \in [w]_\simeq$ such that $\mathsf{explore}(u_1, u_2)$ has explored v.*

Theorem 3.1 guarantees that if $\mathsf{explore}(\varepsilon, u_2)$ has terminated, where u_2 is a maximal transition sequence at the initial state s_0 of a separated transition system, an equivalent transition sequence to every maximal transition sequence at s_0 has been explored. Because our algorithm automatically generates an arbitrary maximal condition sequence at the initial state if the second argument is empty, executing $\mathsf{explore}(\varepsilon, \varepsilon)$ explores representatives for the same set of traces. Therefore, theorem 3.1 provides correctness for our algorithm.

Before we proof the correctness theorem, we introduce the notion of a *maximal feasible prefix of a trace constraint system*. In our algorithm, maximal satisfiable prefixes of trace constraint systems are generated and solved. Solutions to those maximal satisfiable prefixes always exist; however, the corresponding solution sequences may not be feasible because of disabled transitions. Our algorithm handles unfeasible solution sequences by continuing with an arbitrary maximal transition sequence. Intuitively, the prefix of a solution sequence until the first disabled transition is a solution sequence of the maximal feasible prefix of the corresponding trace constraint system.

Formally, we define feasible prefixes and maximal feasible prefixes of trace constraint systems as follows. Let s be a state of a transition system. A prefix $CS' = (A', C', l')$ of a trace constraint system $CS = (A, C, l)$ is a *feasible prefix of CS at s*, written $CS' \sqsubseteq_s CS$, if

$$\exists u \in \mathit{SOL}(CS').\, \exists s'.\, s \xrightarrow{u} s'$$

The maximal feasible prefix of a trace constraint system is its longest feasible prefix. For each constraint system CS and each state s, we define the *maximal feasible prefix of CS at s* as the unique feasible prefix $\mathit{max\text{-}feasible}_{CS}(s)$ of CS such that

$$\begin{aligned}&\mathit{max\text{-}feasible}_{CS}(s) \sqsubseteq_s CS\\ \wedge\, &\forall CS' \sqsubseteq_s CS.(\mathit{max\text{-}feasible}_{CS}(s) \sqsubseteq CS'\\ &\Rightarrow CS' = \mathit{max\text{-}feasible}_{CS}(s))\end{aligned}$$

As the correctness theorem assumes that an execution of explore has terminated, we are able to use information about the completed execution of explore in our proof of theorem 3.1. We define a total order on individual calls to explore in an execution of our algorithm and prove the correctness theorem by an induction on this order. This total order is called *called-before relation* and orders calls to explore with respect to the point in time of their completion, similar to the proof technique used by Abdulla, Aronis, Jonsson, and Sagonas [AAJS14].

Let I be the domain of inputs for explore, i.e., $I = T^* \times T^*$ for some separated transition system (S, s_0, T). Let an execution of explore have terminated. We define the *called-before relation* $<_\alpha \subseteq (I \times I)$ as the smallest total order such that $\iota' <_\alpha \iota$ if and only if $\mathsf{explore}(\iota)$ called (possibly indirectly) $\mathsf{explore}(\iota')$ and $\mathsf{explore}(\iota')$ terminated before $\mathsf{explore}(\iota)$.

We restate Theorem 3.1 more concisely in order to facilitate formulations in the correctness proof, which follows afterwards.

$$\begin{aligned}\forall u, w \in \mathit{max\text{-}seq}(s_0).\, \forall u_1, u_2.\, u = u_1u_2 \wedge\, &\text{“}\mathsf{explore}(u_1, u_2) \text{ has terminated”}\\ \wedge(\forall (i, t_1, t_2) \in \mathit{swapped\text{-}set}(u, w).\, &\forall (i_1, i_2) \in \mathit{range}(u)^2.\\ &u_{|t_1,t_2} = (i_1, i_2) \Rightarrow i_1 \notin \mathit{range}(u_1))\\ \Rightarrow \exists v \in [w]_\simeq.\text{“}&\mathsf{explore}(u_1, u_2) \text{ has explored } v\text{”}\end{aligned}$$

Proof of Theorem 3.1. Let s_0 be the initial state of a separated transition system and let w, u be maximal transition sequences at s_0 such that $u =: u_1u_2$ for some transition sequences u_1 and u_2 and $\forall (i, t_1, t_2) \in \mathit{swapped\text{-}set}(u, w).\, \forall (i_1, i_2) \in \mathit{range}(u)^2.\, u_{|t_1,t_2} = (i_1, i_2) \Rightarrow i_1 \notin \mathit{range}(u_1)$. Let $\mathsf{explore}(u_1, u_2)$ have terminated. Induction over $<_\alpha$.

- base case: (u_1, u_2) is minimal with respect to $<_\alpha$.

  Since (u_1, u_2) is minimal with respect to $<_\alpha$, u_1, u_2 are the arguments to first run of explore that terminates. This run of explore cannot have issued a recursive call. Case distinction.

  - case $\mathsf{execute}(u_1, CS_\varnothing)$ was called and u_2 is empty after the call of $\mathsf{execute}(u_1, CS_\varnothing)$ in line 6

    Here, no transitions are enabled in s_0, i.e., s_0 is a deadlock. Both w and u_1 are maximal at s_0. By assumption, $\neg \mathit{swapped\text{-}dep}(u_1, w)$. Hence, by lemma 3.1, w and u_1 are equivalent.
  - case $\mathit{variations}(u_2) = \varnothing$

    Here, u_2 is a maximal transition sequence at $u_1(s_0)$ without dependencies, i.e., $\mathit{dep}(u_2) = \varepsilon$. Hence, $\mathit{dep}(u_2)$ cannot have a swapped dependency with the corresponding part of w, and we have $\neg \mathit{swapped\text{-}dep}(u, w)$. By lemma 3.1, $[u]_\simeq = [w]_\simeq$.

- induction hypothesis

$$\begin{aligned}
&\forall u', w' \in \textit{max-seq}(s_0).\forall u'_1, u'_2.u' = u'_1u'_2 \\
\wedge&(u'_1, u'_2) <_\alpha (u_1, u_2) \wedge \text{“}\mathsf{explore}(u'_1, u'_2) \text{ has terminated”} \\
\wedge&(\forall (i', t'_1, t'_2) \in \textit{swapped-set}(u', w').\, \forall (i'_1, i'_2) \in \textit{range}(u')^2. \\
&u'_{|t'_1, t'_2} = (i'_1, i'_2) \Rightarrow i'_1 \notin \textit{range}(u'_1)) \\
\Rightarrow\ &\exists v' \in [w']_\simeq.\text{“}\mathsf{explore}(u'_1, u'_2) \text{ has explored } v'\text{”}
\end{aligned}$$

- induction step: (u_1, u_2) is not minimal w.r.t. $<_\alpha$.

  If $u_2 = \varnothing$, an arbitrary maximal transition sequence v beginning at $u_1(s_0)$ is executed. If $u_1 \cdot v$ is a representative of $[w]_\simeq$, we are done. Otherwise, the algorithm proceeds as if $u_2 \neq \varnothing$ from the beginning. Hence, these two cases are exactly analogous. (The executed transition sequence v cannot be empty as (u_1, u_2) is not minimal w.r.t. $<_\alpha$.)

  Beginning in line 11, every constraint system from $\textit{variations}(u_2)$ is generated. If there are no variations of u_2, i.e., $\textit{variations}(u_2) = \varnothing$, then $[w]_\simeq = [u_1u_2]_\simeq$ by lemma 3.1.

  Otherwise, let $c \in \textit{variations}(u_2)$ be a constraint system such that u_1 and c do not share a swapped dependency with w in the following sense. Let $c = (A, C, l)$. For all $i \in \textit{range}(\textit{dep}(u_1) \cdot C)$, if $(\textit{dep}(u_1) \cdot C)[i] = (i_1, i_2)$ and $\textit{dep}(w)[i] = (j_1, j_2)$ then one of the following holds.

$$\begin{cases} u_1[i_1] \neq w[j_2] & \text{if } i_1 \in \textit{range}(u_1) \\ C[i_1] \neq w[j_2] & \text{otherwise} \end{cases} \text{ or } \begin{cases} u_1[i_2] \neq w[j_1] & \text{if } i_2 \in \textit{range}(u_1) \\ C[i_2] \neq w[j_1] & \text{otherwise} \end{cases}.$$

  As every combination of swapped and unchanged dependencies is covered by the variations of u_2, such a constraint system exists.

  In line 12, $\mathsf{execute}(u_1, c)$ is called. By definition, execute executes a solution sequence v of the maximal feasible prefix of c. This solution sequence is a representative of the trace with the dependencies of c by corollary 2.2. If $v \in [w]_\simeq$, we are done.

  Otherwise, let (u'_1, u'_2) be the return value of the last call to execute. In line 13, a recursive call with u'_1, u'_2 as arguments is performed. By the induction hypothesis, equivalent transition sequences to all transition sequences in the following set have been explored:

$$\begin{aligned}
E := \{&v' \in \textit{max-seq}(s_0) : \\
&\forall (i, t_1, t_2) \in \textit{swapped-set}(u'_1 \cdot u'_2, v').\, \forall (i_1, i_2) \in \textit{range}(u'_1 \cdot u'_2)^2. \\
&(u'_1 \cdot u'_2)_{|t_1, t_2} = (i_1, i_2) \Rightarrow i_1 \notin \textit{range}(u'_1)\}
\end{aligned}$$

  If a representative of the trace of w is in E, we are done. Thus, it remains to show that $\forall (i, t_1, t_2) \in \textit{swapped-set}(u'_1 \cdot u'_2, v').\, \forall (i_1, i_2) \in \textit{range}(u'_1 \cdot u'_2)^2.\, (u'_1 \cdot u'_2)_{|t_1, t_2} = (i_1, i_2) \Rightarrow i_1 \notin \textit{range}(u'_1)$.

We have $(u_1', u_2') = \textit{split-at-dependency}(u_1, v_1, v_2,)$ for some v_2 and $v_1 \in \mathit{SOL}(\textit{max-feasible}_c(u_1(s_0)))$. By the definition of *split-at-dependency*, any dependency which is not already contained in u_1 or c is not contained in u_1' but in u_2'.

By the assumption that swapped dependencies of u and w are not contained in u_1 and by the choice of c, $\forall(i, t_1, t_2) \in \textit{swapped-set}(u_1' \cdot u_2', v').\, \forall(i_1, i_2) \in \textit{range}(u_1' \cdot u_2')^2.\, (u_1' \cdot u_2')_{|t_1,t_2} = (i_1, i_2) \Rightarrow i_1 \notin \textit{range}(u_1')$. Thus, a representative of w is contained in E.

□

By this correctness proof, every analysis that has been carried out by our algorithm is sound. An accompanying proof that our algorithm terminates for all separated transition systems is desirable. However, our algorithm converges, in its simple form of this chapter, for some separated transition systems. We discuss this issue in more detail and suggestions for optimizations in the following chapter. We refrain from extending our algorithm and providing a proof that the new variant always terminates in order to perform empirical and experimental evaluations based on several example systems.

This concludes our notion of correctness. It entails a precise formulation of a correctness condition, detailed notions and definitions which facilitate the formalization of concepts used in our algorithm and separated constraint systems, and a clearly structured correctness proof. We expect that correctness proofs for variants of our algorithm can be easily obtained based on our definitions and by extending our correctness condition and proof.

Chapter 4

Empirical Evaluation and Potential Optimizations

This chapter presents interesting example programs which expose inefficiencies of our algorithm in its simple form of chapter 3. Sections 4.1 to 4.3 describe optimizations related to the constraint solving used by our algorithm. The following two sections 4.5 and 4.4 discuss how to apply the general optimizations of stateful exploration and parallelization.

4.1 Using a Program Order

A characteristic of any multi-threaded program is that it can be modeled as a separated transition system, by defining one transition for each statement in the program. Intuitively, such a separated transition system can be thought of as consisting of one or more processes, which in turn have one or more transitions. In any state of the system, exactly one transition of each process is enabled, and executing a transition of any process does not disable the currently enabled transitions of any other process. The fact that a process has no more than one enabled transition at each state corresponds to the concept of a *program order* [AG96]. In this section, we present a similar notion specifically for trace constraint systems.

As an example, let us consider the *program order example*, a separated transition system which models the program in listing 4.1. It consists of two processes which write to two shared variables. Process 1 consists of transitions t_{11} and t_{12} and process 2 consists of transitions t_{21} and t_{22}. Each transition models one statement and accordingly is, intuitively, enabled if and only if the imaginary

```
1 Thread 1 {
2   t11: x := 1
3   t12: y := 1
4 }
```

```
5 Thread 2 {
6   t21: x := 2
7   t22: y := 2
8 }
```

Listing 4.1: The program order example.

program counter of its process is at the matching position. In other words, t_{11} is enabled if and only if t_{12} has not been executed before and t_{12} is enabled if and only if t_{11} has been executed before. The transitions of process 2 are enabled analogously.

Because the execution of t_{11} enables t_{12} and the execution of t_{21} enables t_{22}, there exist dependencies

$$(t_{11}, t_{12}) \text{ and } (t_{21}, t_{22}) \tag{4.1}$$

according to definition 5 (dependency relations). These dependencies differ from the remaining dependencies

$$(t_{11}, t_{21}) \text{ and } (t_{12}, t_{22})$$

in that the former relate transitions of the same process. Consequently, the dependencies of (4.1) do not occur in reversed order in any transition sequence of the program order example.

If we run the algorithm from chapter 3 on this example, it explores a maximal transition sequence at the initial state and calculates variations for all four dependencies, i.e., 15 variations. However, from these 15 variations, only those three are feasible where the dependencies from (4.1) are not swapped. Therefore, we introduce the *program successor relation*, which contains dependencies which can not be swapped due to program order constraints. The corresponding optimization of our algorithm modifies the generation of trace constraint systems in *variations*(). Whenever variations are calculated, dependencies included in the program successor relation are not swapped and used unchanged as dependencies. Thereby, the generation of unfeasible constraint systems is avoided.

We sketch the construction of a program successor relation at the following example. Let us consider the program `if b then x:=1 else x:=2` which assigns `1` to `x` if the condition `b` is satisfied and assigns `2` to `x` otherwise. Assume that the branching is modeled by a transition t_1 and the statements `x:=1` and `x:=2` are modeled by transitions t_2 and t_3, respectively. In order to model this program as input to our algorithm, we may specify the program successors of t_1 as $\{t_2, t_3\}$ because there exists a state where t_1 is followed by t_2 and a state where t_1 is followed by t_3.

In general, a program successor relation is characterized by the following property, which expresses that whenever a transition enables another transition, these transitions have to be related by the program successor relation.

$$\forall t, t' \in T. \forall s, s' \in S. (s \xrightarrow{t} s' \wedge t' \notin \mathit{enabled}(s) \wedge t' \in \mathit{enabled}(s') \Rightarrow t \xrightarrow{ps} t')$$

The program successor relation of a transition system can be used in our algorithm in order to generate, given a transition sequence u feasible at some state s, a transition sequence v which is not Mazurkiewicz equivalent to u but still feasible at s. If, during the process of generating v, two transitions of u from the same process are swapped, v is likely not feasible at s, as otherwise two transitions of the same process would be enabled at the same state. The program successor relation helps to avoid switching the order of transitions which are from the same process.

Given a program successor relation of a transition system, the program successor relation of a specific transition sequence u at a state s can be generated

```
1  Thread 1 {                          7  Thread 2 {
2    t11: if x == 0 {                  8    t21: if y == 0 {
3       t12: y := 1                    9       t22: z := 1
4    }                                 10   }
5    t13: read x                       11   t23: read y
6  }                                   12 }
```

Listing 4.2: The global branchings example.

as $ps(u,s) := \{(i_1,i_2) \in range(u) : i_1 < i_2 \wedge u[i_1] \xrightarrow{ps} u[i_2]$
$\wedge \forall i_2'.\, i_1 < i_2' < i_2 \Rightarrow \neg u[i_1] \xrightarrow{ps} u[i_2']\}$. By introducing separate lists for program successor constraints and constraints for remaining dependencies in trace constraint systems, variations can be easily generated by swapping only dependencies in the latter list. Specifically, we could change the definition of trace constraint systems of transition sequences by replacing the list of constraints by the following two lists (where $ps^*(u,s)$ denotes the transitive closure of $ps(u,s)$).

- $C_{dep} = list(\{(\alpha_i,\alpha_j) \in A \times A : (i,j) \in dep(w) \wedge (i,j) \notin ps^*(w)\}, <_{\mathbb{N}\times\mathbb{N}})$
- $C_{ps} = list(\{(\alpha_i,\alpha_j) \in A \times A : (i,j) \in ps(w)\}, <_{\mathbb{N}\times\mathbb{N}})$

We use program successor relations in the prototype implementation of our algorithm. Chapter 5 describes our implementation and discusses the algorithm we use to solve trace constraint systems with program successor relations.

4.2 Programs with Global Branchings

As an example, let us consider the program of listing 4.2 and a corresponding transition system. Both threads read a global variable and write to a global variable only if the read operation returns 0. Therefore, the transitions modeling the write operations are disabled depending on an operation on a global variable (the initial read operation). We denote such a control flow branching as a *global branching*. In general, a control flow branching is a global branching if both the evaluation of the branching condition and the body (or region) of the branching contain distinct operations on global memory.

When a concurrent program is modeled as a transition system, the transition modeling a global branching has the characteristics of a branching as well. Formally, a transition t is a branching transition if it enables different program successor transitions (cf. section 4.1) depending on the current state, i.e., if $\exists s, s'.\, t \in enabled(s) \wedge t \in enabled(s') \Rightarrow (enabled(t(s)) \setminus enabled(s)) \neq (enabled(t(s')) \setminus enabled(s'))$. In contrast, a control flow branching with only local operations inside the body of the branching is not visible as a branching transition in the transition system if local operations are combined into one transition with a following or preceding global operation. Such an optimization is made, for example, in the Java Pathfinder tool[1]. Similarly, if a global operation may disable a following global operation depending on its outcome but both operations are executed atomically all at once, no branching transition

[1] `http://babelfish.arc.nasa.gov/trac/jpf`

```
1 Thread 1 {
2   t11: write x
3 }
```

```
4 Thread 2 {
5   t21: read x
6   t22: read x
7 }
```

Listing 4.3: The unsatisfiable constraints example.

appears in the transition system. The latter is the case, for example, when a compare-and-swap operation is modeled as a single transition.

Let us investigate in more detail the *global branchings* example from listing 4.2. It consists of two processes and three memory locations x, y, and z. Process 1 reads x (transition t_{11}) and writes to y (transition t_{12}) if x evaluates to 0. Subsequently, it reads x again (transition t_{13}). Process 2 analogously reads y (transition t_{21}), writes to z (transition t_{22}) if y evaluates to 0 and finally reads y again (transition t_{23}).

When running our algorithm on the program order example, it first explores an arbitrary maximal transition sequence at the initial state, for example

$$u = t_{11}t_{21}t_{12}t_{13}t_{22}t_{23}.$$

Both dependencies

$$(t_{21}, t_{12}) \text{ and } (t_{12}, t_{23})$$

(modulo program order dependencies) are detected. Hence, three constraint systems are generated, one for each variation of u. Let us investigate the constraint system c where only the first dependency is swapped. The call to execute with c as argument executes the following transition sequence, for example.

$$v = t_{11}t_{12}t_{21}t_{13}t_{23}$$

As the write to x by t_{12} occurs before the read from x by t_{21}, transition t_{22} is disabled. Therefore, the execution of the solution sequence is canceled after t_{13}. An arbitrary maximal transition sequence at the corresponding state is executed, in our example t_{23}. Hence, t_{23} belongs to section 3 of v (cf. section 3.1). The dependency (t_{12}, t_{23}) is already covered but not recognized as such because t_{12} and t_{23} belong to different sections of v. Consequently, the recursive call explore$(t_{11}, t_{12}t_{21}t_{13}t_{23})$ is performed. The arguments of explore correspond to the transition sequence u which has been executed during the first call to explore. Accordingly, the same two dependencies (t_{21}, t_{12}) and (t_{12}, t_{23}) are detected, and the constraint system where only the first dependency is swapped will eventually lead again to the recursive call explore$(t_{11}, t_{12}t_{21}t_{13}t_{23})$. Thus, the algorithm diverges in this case.

The issue occurring for the above example exploration is the unawareness of our algorithm that t_{22} is disabled if t_{12} occurs before t_{21}. A potential solution is to change our algorithm so that it explores transition sequence only up to the next branching transition instead of up to a deadlock.

4.3 Omission of Empty Solution Sequences

In order to cover all executions of a transition system, our algorithm solves the maximal satisfiable prefix of a trace constraint system. In some cases, this leads

to a redundant exploration of a single trace. We illustrate this issue at the example from listing 4.3. Here, process 1 writes to a shared variable from which process 2 reads twice.

For example, our algorithm begins by exploring the transition sequence $t_{11}t_{21}t_{22}$. There are two dependencies, (t_{11}, t_{21}), (t_{11}, t_{22}), and accordingly, our algorithm creates three trace constraint systems. Two of these trace constraint systems are solved and the corresponding solution sequences $t_{21}t_{11}t_{22}$ and $t_{21}t_{22}t_{11}$ are explored without issuing a recursive call to explore. However, the third constraint system c with constraints (t_{11}, t_{21}) and (t_{22}, t_{11}) is unsatisfiable due to the program order constraint (t_{21}, t_{22}) (cf. section 4.1).

In its original form of chapter 3, our algorithm does not terminate because the call to execute generates the empty solution sequence of the maximal satisfiable prefix of c and consecutively executes an arbitrary maximal transition sequence at the initial state, where both dependencies are not recognized as already covered. A solution to this issue is to omit maximal satisfiable prefixes which are empty. We implement this optimization in our prototype implementation described in chapter 5.

4.4 Stateful Exploration

Stateful exploration is a technique which is generally applicable to many state search algorithms and is not limited to POR algorithms. We propose a variant of stateful exploration which does not provide the benefit of being able to detect cycles in the state graph of a transition system but also reduces the additional memory required to store states for a conventional stateful exploration.

Specifically, we propose to store for each call to explore(u_1, u_2) the resulting state of u_1, i.e., $u_1(s_0)$. Whenever a state has already been stored, the current call to explore terminates without further exploration. We implement this optimization in our prototype described in chapter 5. It suits as an intermediate mitigation of the issue described in 4.2. As soon as the optimization proposed in section 4.2 is implemented, recording states can be abandoned.

4.5 Parallelization

As every loop iteration in the main procedure explore of our algorithm is independent, the loop can easily be parallelized. For example by using a thread pool, each trace constraint system of the current set of variations can be solved and the corresponding solution sequence be explored on a separate processor.

However, the optimization for programs with global branchings, proposed in section 4.2 may change our algorithm so that additional synchronization between several loop iterations is necessary. In this case, the performance gain obtained by parallelization is presumably reduced.

Chapter 5

Prototype Implementation

We have implemented our algorithm and the DPOR algorithm by Flanagan and Godefroid as Python programs in order to compare their performance experimentally. Additionally, we have implemented a model checking tool which takes as input a distributed program, compiles it into a transition system representation, and applies selected model checking algorithms on the transition system. We have implemented our algorithm with three optimizations,

- usage of a program order (cf. section 4.1),
- omission of empty solution sequences (cf. section 4.3), and
- state tracking (cf. section 4.4),

and otherwise as presented in section 3.1. In this section, we give a brief overview over the implementations of our algorithm, the algorithm by Flanagan and Godefroid, and our model checking tool. Details about the architecture and the usage of these implementations can be found in appendix A.

Our prototype implementation shows the feasibility of our algorithm on three benchmarks for model checking algorithms. Specifically, we evaluate the running time, number of explored transitions, and number of explored maximal sequences of our algorithm in comparison to the algorithm by Flanagan and Godefroid. The corresponding experiments are presented in chapter 6.

We have chosen Python as the programming language for our implementation because it has enabled us to write more abstract program code than it would have been possible with Java or C++, for example. In effect, the implementations of the two model checking algorithms are more similar to their pseudo code variants. Disadvantages of Python, in comparison to Java or C++, include missing performance optimization of program code. However, the absolute performance is only of minor importance to us, as we intend to evaluate the running time of both algorithms only relatively.

Alternatively to our new model checking tool, we could have implemented our algorithm in an existing model checking tool like Java PathFinder [VHB+03] or Concuerror [GCS11], for example. Due to the limited time constraints of this thesis, and as the focus of this thesis is the theoretical development of a new DPOR algorithm, we have chosen to develop a new model checking tool. Additionally, this choice facilitates the implementation of features such as support for a specific benchmark or a specific calculation of dependencies.

For our model checking tool, we have developed a new input language dlang along with a compiler which translates terminating programs written in dlang to Python encodings of transition systems. Our input language allows to write distributed programs in the style of an imperative, general purpose programming language. The supported features include conditionals, loops, constants, global and local variables, the usage of a program block inside multiple threads, array access and initialization of the program memory. The corresponding compiler takes as input a program in dlang and produces a Python program which encodes a transition system corresponding to the input program. Included in the compiler is a static program analysis which records static dependencies between transitions, based on a static approximation of accessed memory locations. The implementation of our compiler is based on the ANTLR parser generator.[1] We have specified the grammar and the domain specific source code for our compiler, for example static dependency calculation and code generation, inside an ANTLR grammar file. Using this grammar file, ANTLR generates a Java program that implements our compiler. A more detailed description and example programs written in dlang are included in appendix A.

The transition systems generated by our compiler contain exactly one transition for each statement of the input program. A transition may access multiple memory locations. If an input program provided to our compiler terminates, a separated transition system is generated. Otherwise, the generated code may not comply with definition 1 of transition systems because of cycles in the corresponding state graph.

Our model checking tool calculates the dependency between transitions both statically and dynamically. As mentioned above, the compiler statically approximates for each transition the accessed memory locations. The resulting static dependency relation is available to any model checking algorithm run by the tool. During the run of a model checking algorithm, our tool calculates the exact memory locations a transition reads from and writes to. For our algorithm, we consider two transitions dynamically dependent if and only if the following conditions are met.

- The transitions belong to different processes (i.e., they are not related by the transitive closure of the program successor relation, cf. section 4.1).
- One of the transitions writes to a location from which the other transition either reads from or writes to.

Implementation of LazyPOR The implementation of our algorithm is mostly straightforward, with the exception of the generation and solving of trace constraint systems. We have implemented trace constraint systems in two variants; the first variant uses the SMT solver Z3,[2] while the second variant directly generates a linear extension of the partial order associated with a trace constraint system. As preliminary experiments have shown that the process of solving trace constraint systems using Z3 consumed a high amount of time, we have focused on the faster, second variant. This variant generates a trace constraint system $CS(u)$ from a given transition sequence u in the following manner.

[1] `http://www.antlr.org/`
[2] `http://z3.codeplex.com/`

First, the program successor relation $ps(u)$ and the dependency list $dep(u)$ are calculated for u. The program successor relation $ps(u)$ induces a linear order on each process's indices of u; the relation induced by $dep(u)$ relates indices of u which are labeled with transitions from different processes. Second, a graph is constructed with a node for every index of u and an edge for every member of both relations. As a result, the graph contains one weakly connected component for each process with edges between these components for each dependency. If the graph is cycle-free, it represents a partial order and a solution to the corresponding trace constraint system can be generated by constructing a linear extension.

Listing 5.1 depicts the algorithm used by the second implementation variant to generate such linear extensions. As required by our algorithm from section 3.1, it generates solutions for the maximal feasible prefix of the trace constraint system which corresponds to the input graph. For a given input graph, the algorithm of listing 5.1 attempts to iteratively find and remove a node of the graph which is minimal with respect to the induced partial order, i.e., which does not have any predecessor with respect to both the program successor relation and the dependency list. The nodes thereby removed constitute the solution.

In more detail, the algorithm of listing 5.1 continuously iterates over all processes. For the current process pid, it fetches in line 6 the node which corresponds to a statement of pid without a program predecessor. A pointer to the minimal node of each process *proc* (with respect to the program successor relation) can be easily maintained by storing all nodes of *proc* in a list, ordered with respect to the program successor relation. If the minimal node of pid has additionally no predecessor with respect to the dependency list, it is removed and added to the solution. Otherwise, the algorithm registers that pid does currently not have a minimal node. If no process has a minimal element, the graph contains a cycle and the condition in line 14 is satisfied. The algorithm exits the loop and returns the current solution.

Implementation of Flanagan and Godefroid's algorithm Our implementation of Flanagan and Godefroid's DPOR algorithm follows the pseudo code of the variant without stack traversals [FG05, Figure 5].

```
solve(graph) {
  solution := []
  loop_detector := []
  pid := 0
  while true {
    next := graph.get_minimal(pid)
    if next has no dependency predecessor {
      remove all outgoing edges from next
      remove next from graph
      solution.append(next)
      loop_detector := {}
    } else {
      loop_detector.append(next)
      if loop_detector.length = number_of_processes {
        break
      }
    }
    if graph is empty {
      break
    }
    pid := (pid + 1) % number_of_processes
  }
  return solution
}
```

Listing 5.1: Algorithm that generates a linear extension as a solution to a trace constraint system.

Chapter 6

Experimental Evaluation

In order to compare the practical fitness of our algorithm with the well-established DPOR algorithm by Flanagan and Godefroid [FG05], we conduct four experiments on example programs. In this chapter, we refer to our algorithm as *ZPOR* and to Flanagan and Godefroid's algorithm as *DPOR*. The experiments consist of two example programs from the literature, indexer [FG05] and readers−writers [CHP71], along with our example program branching and a modification of the indexer example. In particular, we measure the running time and the number of explored *executions* for *ZPOR* and for *DPOR*, where an execution in this context corresponds to a path from the initial state to a deadlock of a transition system. Detailed measurement results are depicted in appendix B. We continue by describing the setup of our experiments, stating our hypotheses, and reporting the results.

6.1 Experiment Setup

For all experiments reported in this thesis, we use two machines with an identical setup. Each machine has an Intel Core i7−4790 CPU with eight cores at 3.60 GHz and 16 GB main memory. The operating system is Debian GNU/Linux version 3.13.5−1 x86_64 with kernel 3.13−1−amd64 #1 SMP. The Python version is 3.4.2. Both machines are headless and do not have a running instance of Xorg or other graphical environments. We control both machines over SSH connections.

Using our model checking tool, we apply our prototype implementations of *ZPOR* and *DPOR* to the example programs. Both the implementation of the model checking tool and the implementations of the algorithms are described in chapter 5. In order to measure the number of explored executions, we extend our implementations of the algorithms as follows. In the implementation of our algorithm (*ZPOR*), we increment a variable for each call to execute. In the implementation of Flanagan and Godefroid's algorithm, we increment a variable each time the current state is a deadlock. Thereby, the number of explored paths from the initial state to a deadlock of the respective transition system, or, equivalently, the number of explored maximal transition sequences at the transition system's initial state is measured.

The running time is measured by the help of the Python package cProfile. In

particular, the running time of an algorithm in our experiments is the time in seconds reported by cProfile as the *cumulative time* consumed for the execution of the explore procedure of the respective algorithm. The cumulative time of a function execution is the time spent in the function itself and all functions called by the function during its execution. In order to facilitate their presentation, all time measurements below 0.001 seconds are rounded to 0.001 seconds.

Each of the example programs used in our experiments has a parametric number of threads. An experiment consists of multiple runs of both algorithms on the respective example program with an increasing number of threads. In particular, we conduct each experiment by invoking our model checking tool twice. Each invocation runs one of the two algorithms on the selected example program consecutively with an increasing number of threads. The model checking tool and the implementations of both algorithms use no more than one CPU core. No additional applications are executed on the two machines during an experiment. Hence, the remaining seven CPU cores are used only for system processes and the SSH connection we use to control the machines. Whenever a single run of *ZPOR* or *DPOR* takes considerably longer than two days, we stop the model checking tool and omit all remaining runs of the respective algorithm in the current experiment.

6.2 Possible Outcomes and Hypotheses

This section describes how we evaluate measurement outcomes and which outcomes we expect for a specific class of example programs. We denote the measurements for a single run of an algorithm on an example program with a specific number of threads by

$\boldsymbol{ZPOR}_{time}$ for the running time of *ZPOR*

$\boldsymbol{ZPOR}_{exec}$ for the number of executions explored by *ZPOR*

$\boldsymbol{DPOR}_{time}$ for the running time of *DPOR*

$\boldsymbol{DPOR}_{exec}$ for the number of executions explored by *DPOR*

We categorize the possible outcomes of our measurements into four categories. These categories characterize the performance of our algorithm, taking *DPOR* as a reference point.

1. $ZPOR_{time} \leq DPOR_{time}$ and $ZPOR_{exec} \leq DPOR_{exec}$
2. $ZPOR_{time} \leq DPOR_{time}$ and $ZPOR_{exec} > DPOR_{exec}$
3. $ZPOR_{time} > DPOR_{time}$ and $ZPOR_{exec} \leq DPOR_{exec}$
4. $ZPOR_{time} > DPOR_{time}$ and $ZPOR_{exec} > DPOR_{exec}$

As our algorithm explores multiple transitions at once, we expect that *ZPOR* shows a lower overhead than *DPOR*, in the sense that the quotient of running time and number of explored executions for *ZPOR* is less than for *DPOR*. Our corresponding hypothesis is that no measurement outcome of our experiments lies in category 3, i.e., whenever $ZPOR_{exec} \leq DPOR_{exec}$, $ZPOR_{time} \leq DPOR_{exec}$.

```
1 Thread Writer {
2   write x
3 }
4
5 Thread(tid) Reader {
6   read x
7 }
```

Listing 6.1: Readers-writers example (readers−writers).

As described in section 4.2, the state graph reduction performed by our algorithm may be non-optimal in the presence of global branchings (disabled transitions due to control flow branchings on global variables). Accordingly, we expect that *ZPOR* explores more executions than *DPOR* in these cases. However, if an example program contains no global branchings, we expect that *ZPOR* explores less than or equally many executions as *DPOR*. Hence, our second hypothesis is that for programs without global branchings, the measurement outcomes of our experiments lie in category 1 and for programs with global branchings, *ZPOR* explores more executions than *DPOR*, i.e., the measurement outcomes of our experiments lie in category 2 or 4.

6.3 Experiments

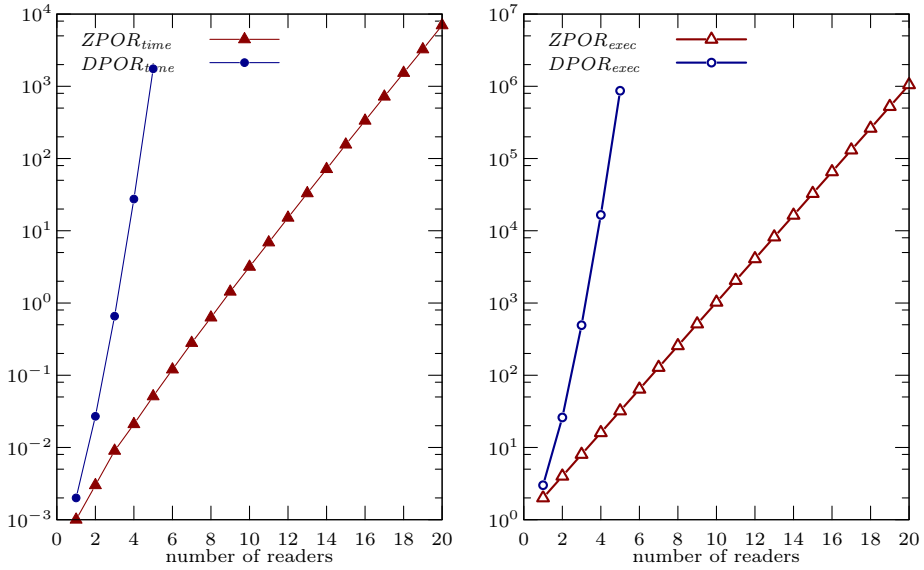
In this section, we shortly describe the example program of each experiment and present our measurement results. We evaluate our hypotheses and assess the performance of our algorithm.

6.3.1 Readers-Writers

Our first example program is the well-known readers-writers problem [CHP71]. We use a simplified variant, shown in listing 6.1, where there exists one thread which writes to a shared variable x and one or more additional threads which read from x. The notation Thread(tid) indicates that the following code block is executed by one or more threads. The number of these threads increases during an experiment and in the n-th thread, tid evaluates to $n-1$. In this experiment, we run *ZPOR* and *DPOR* for one to 20 readers, i.e., the total number of threads (including one writer thread) ranges from two to 21.

This example program contains no control flow branching. Furthermore, the transition corresponding to the write statement is dependent with all other transitions, while two read transitions are never dependent. All dependencies in this program can be accurately predicted by a static dependency analysis, i.e., all dependencies hold globally in all states of the transition system.

Figure 6.1 shows the measurement outcomes for the readers−writers example. On the left, in figure 6.1a, the running time in seconds (ordinate) for the number of readers (abscissa) is depicted. On the right, figure 6.1b shows the number of explored executions (ordinate) for the number of readers (abscissa). Both ordinates have a logarithmic scale with base 10.



(a) Running time in seconds

(b) Number of explored executions

Figure 6.1: Results for the readers−writers example.

For all measurements of this experiment, the outcome lies in category 1. Both the running time and the number of explored executions is considerably smaller for *ZPOR* than for *DPOR*. Moreover, the growth of $DPOR_{time}$ and $DPOR_{exec}$ is considerable larger than $ZPOR_{time}$ and $ZPOR_{exec}$, respectively. This rapid growth induces that *DPOR* shows a running time of more than three days for six readers. Therefore, we omit all measurements for *DPOR* with six or more threads for this experiment.

Both our first and second hypothesis are confirmed by this experiment, as all measurement outcomes lie in category 1.

6.3.2 Indexer

The indexer example [FG05] is a program where all transitions are statically but not dynamically dependent. It has served as a benchmark for several DPOR algorithms [FG05, GFYS07, AAJS14]. The pseudo code illustrating this program is depicted in listing 6.2. Each thread calculates four messages and writes them in a shared hash table table. Collisions are avoided by the use of a compare-and-swap statement (cas), which writes its third argument at the memory location of its first argument, if its first argument evaluates to the same value as the second argument. The message values (calculated by getmsg) depend on the thread ID tid, which ranges from 0 for the first thread to 18 for the 19th thread. As the cas statement is the only statement accessing global memory, each thread is modeled by a single transition. Therefore, there exists no global branching.

Typically, a static dependency analysis cannot distinguish cases where two threads access the same location inside the shared hash table and cases where two threads both write to the same hash table but at distinct positions. There-

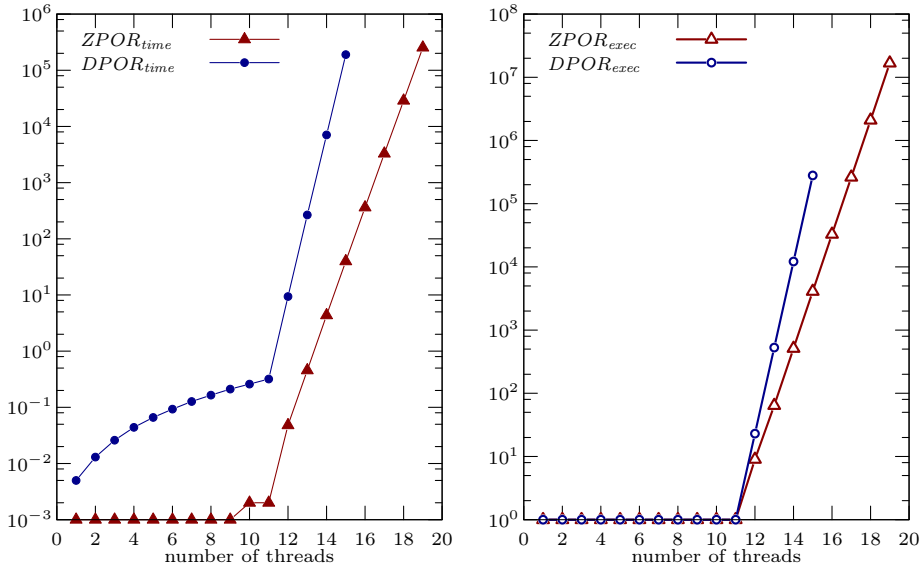
```
1  const int size = 128
2  const int max = 4
3  int[size] table
4
5  Thread(tid) Indexer {
6    int m = 0
7    int w
8    int h
9    while (true) {
10     w := getmsg()
11     h := hash(w)
12     while (cas(table[h], 0, w) == false) {
13       h := (h + 1) % size
14     }
15   }
16 }
17
18 int getmsg(tid) {
19   if (m < max) {
20     return (++m) * 11 + tid
21   } else {
22     exit()
23   }
24 }
25
26 int hash(int w) {
27   return (w * 7) % size
28 }
```

Listing 6.2: Indexer example by Flanagan and Godefroid [FG05] (presentation slightly changed).

fore, all transitions in this example have to be considered as mutually statically dependent. However, dynamic dependencies occur only occasionally and only with twelve threads or more, due to the definition of getmsg and hash.

The measurement outcomes for indexer are depicted in figures 6.2a (running time) and 6.2b (number of explored executions). All measurement outcomes of this experiment lie in category 1. For all runs with one to eleven threads, no dynamic dependencies occur and both *ZPOR* and *DPOR* explore exactly one execution. For these runs, the running time of *DPOR* constantly increases, while the running time of *ZPOR* only shows a slight increase of 0.001 seconds at ten threads and is otherwise constant. For all runs with twelve threads or more, both the running time and the number of explored executions for *ZPOR* is less than for *DPOR*. Additionally, both $DPOR_{time}$ and $DPOR_{exec}$ raise more quickly than $ZPOR_{time}$ and $ZPOR_{exec}$, respectively, and $ZPOR_{time}$ is below three days for up to 19 threads, while $DPOR_{time}$ raises above three days for already 14 threads.

As for the previous experiment, both our first and second hypothesis are



(a) Running time in seconds

(b) Number of explored executions

Figure 6.2: Results for the indexer benchmark.

confirmed by the indexer experiment, as all measurement outcomes lie in category 1.

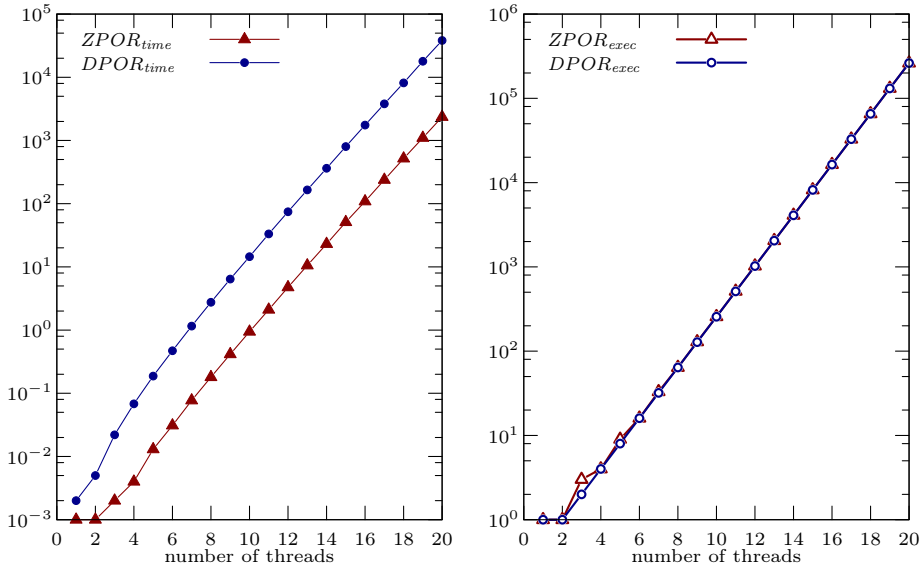
6.3.3 Small Indexer

We have tested several variants of the indexer example where we changed the number of loop iteration (constant max) and the number of threads at which dynamic dependencies occur (in the definition of getmsg). In this experiment, we use the variant with two loop iterations and dynamic dependencies with three threads or more. We refer to this variant as indexer−small. It corresponds exactly to the indexer depicted in listing 6.2 with the following two changes. Line 2 is changed to **const int** max = 2 and line 20 to **return** (++m)∗ 2 + tid.

We choose the indexer−small variant because both algorithms finish in less than one day and explore nearly the same number of executions for all runs. In particular, the indexer−small variant allows us to easily compare the running time overhead of both algorithms, i.e., the quotient of running time and number of explored executions.

Figure 6.3 depicts the measurement outcomes for the indexer−small example, with the running time in figure 6.3a and the number of explored executions in figure 6.3b. The running time of *ZPOR* is always below the running time of *DPOR* and both measures increase at a similar rate. The number of explored executions is very similar for both algorithms. For *DPOR*, the number of explored executions is 2^n, where n is the number of threads. The number of executions explored by *ZPOR* is either equal to $DPOR_{exec}$ or one to two executions greater than $DPOR_{exec}$. Thus, the measurement outcomes lie in the categories 1 and 2.

As no measurement outcome lies in category 3, our first hypothesis is con-



(a) Running time in seconds

(b) Number of explored executions

Figure 6.3: Results for the indexer−small example.

firmed in this experiment. Moreover, the running time overhead of *ZPOR* is considerable smaller than the running time of *DPOR*; with nearly the same number of executed executions, the difference in the running time of *DPOR* and *ZPOR* increases exponentially from 0.001 seconds for one thread to approximately ten hours for 20 threads.

Our second hypothesis is not confirmed by all runs in this experiment, as $ZPOR_{exec}$ lies one or two executions above $DPOR_{exec}$ for some runs. This observation indicates that even for programs without global branchings, our algorithm in the form of our implementation is not optimal, i.e., explores two representatives of the same trace in some cases.

6.3.4 Branching

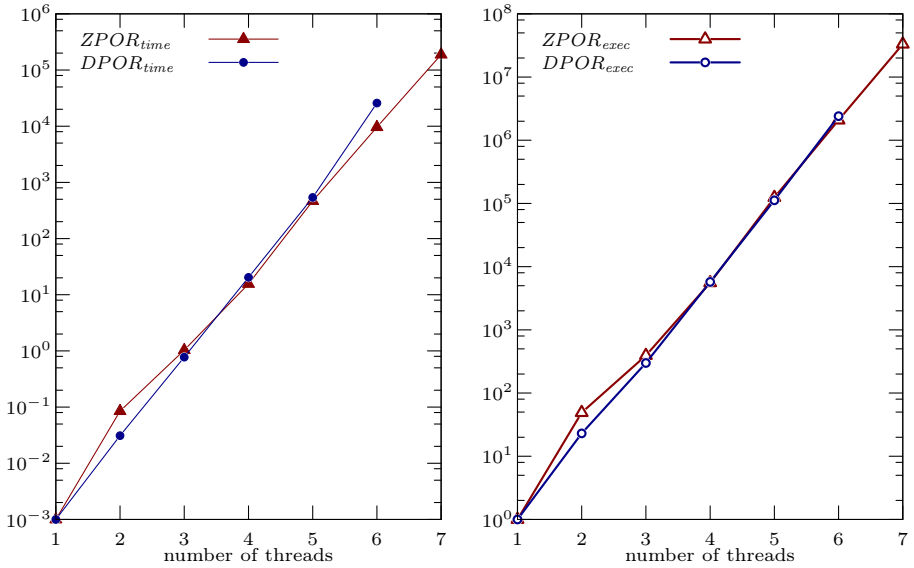
In order to evaluate our algorithm on a program with global branchings, we introduce the branching example, which is depicted in listing 6.3. Each thread of the program reads a shared array at the position indicated by its thread ID and writes to the following position if the read yields zero. Subsequently, the same operations are repeated once.

Preliminary experiment results with the branching example have shown that *ZPOR* accumulates a deep recursion stack even for a small number of threads. Presumably, our algorithm diverges for these inputs. As a consequence, it has been practically unfeasible to run *ZPOR* on the branching example. Therefore, we extend *ZPOR* so that it maintains a set of already visited states, as described in section 4.4 and chapter 5.

Figures 6.4a and 6.4b show the measurement outcomes in the branching example for the running time and the number of explored executions, respectively.

```
1  const int length
2  int[] x[length]
3
4  Thread(tid) Branching {
5    if x[tid] == 0 {
6      x[tid + 1 % length] := 1
7    }
8    if x[tid] == 0 {
9      x[tid + 1 % length] := 1
10   }
11 }
```

Listing 6.3: Branching example



(a) Running time in seconds

(b) Number of explored executions

Figure 6.4: Results for the branching example.

The running time of *ZPOR* is equal to the running time of *DPOR* for one thread, greater for two and three threads, and smaller for four to seven threads. The running times of *DPOR* for seven threads and of *ZPOR* for eight threads lie above three days and we canceled the corresponding runs. For one thread, both *ZPOR* and *DPOR* explore one execution. For two, three, and five threads, $ZPOR_{exec}$ lies above $DPOR_{exec}$, for four and six threads, $ZPOR_{exec}$ lies below $DPOR_{exec}$. Both measures $ZPOR_{exec}$ and $DPOR_{exec}$ show a similar increase. The measurement outcomes are spread among the categories 1, 2, and 4.

Our first hypothesis is confirmed in this experiment, as no measurement outcome lies in category 3. Our second hypothesis, in this case about programs with global branchings, is confirmed in the runs with two, three, and five threads. In the remaining runs, $ZPOR_{exec}$ is either equal to or slightly smaller than $DPOR_{exec}$, despite the presence of global branchings. Although $ZPOR_{exec}$ and $DPOR_{exec}$ show similar values, the running time of *ZPOR* raises less quickly than the running time of *DPOR*, with an increasing number of threads. As a consequence, *ZPOR* still finishes within approximately two days for seven threads, while *DPOR* does not terminate within three days for seven threads.

6.4 Summary of Experiment Results

The experiments show that our algorithm is applicable to the investigated example programs and often completes the state search in less time than our implementation of Flanagan and Godefroid's algorithm. In a number of cases, the state search by *ZPOR* was feasible within three days for a higher number of threads than the state search by *DPOR*.

In all experiments, our first hypothesis that the overhead of *ZPOR* is lower than the overhead of *DPOR* is confirmed. This supports our conjecture than exploring multiple transitions at once requires less time than exploring one transition at a time.

Our second hypothesis is confirmed by most but not all runs in the presented experiments. First, our experiments indicate that *ZPOR* improves over the running time of *DPOR* for programs without global branchings. However, it appears that our algorithm unnecessarily explores redundant executions in some cases. Second, the branching experiment shows that the performance of *ZPOR* decreases when applied to a program with global branchings. Therefore, the performance of our algorithm can potentially be improved by optimizing it for global branchings, for example by the optimization proposed in section 4.2.

Chapter 7

Conclusion

Our new DPOR algorithm, LazyPOR, implements a new approach of exploring the state space of a system by exploring complete system executions at once. We introduce a new representation of Mazurkiewicz traces, trace constraint systems, which allow to plan alternative system executions before exploring them. The soundness of representing traces by trace constraint systems is established by our correspondence proof for dependency graphs. Based on this preliminary result, we provide a correctness condition and a correctness proof for a basic variant of our algorithm.

The formalization of our new concept of handling dependencies of complete system executions both help in the presentation of our correctness proof and with a prototype implementation of our algorithm. This prototype implementation comprises several optimizations of our algorithm, which we present with an empirical evaluation of our algorithm on challenging example programs. Along with the prototype implementation of our algorithm, we implement the well-established DPOR algorithm by Flanagan and Godefroid and develop a model checking tool for running state search algorithms on concurrent programs.

Our experiments compare LazyPOR with the DPOR algorithm by Flanagan and Godefroid. All our experiments confirm our hypothesis that our algorithm reduces the running time overhead of state searches for concurrent programs. This experimental result strengthens our conjecture that a practical exploration of complete system executions consumes less time than the exploration of single transitions. In most of our experiments, our algorithm improves over the reference DPOR algorithm both in terms of the running time performance and the amount of reduction.

In conjunction with our suggested optimizations, our algorithm enables new approaches for state search such as parallelized exploration and combined exploration of transition sequences. These approaches may help to further improve the performance of verification techniques for concurrent systems.

Bibliography

[AAJS14] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 373–384. ACM, 2014.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AR88] IJsbrand Jan Aalbersberg and Grzegorz Rozenberg. Theory of Traces. *Theor. Comput. Sci.*, 60:1–82, 1988.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKSS11] Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. Supporting domain-specific state space reductions through local partial-order reduction. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 113–122. IEEE, 2011.

[BSS10] Péter Bokor, Marco Serafini, and Neeraj Suri. On Efficient Models for Model Checking Message-Passing Distributed Protocols. In John Hatcliff and Elena Zucca, editors, *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 216–223. Springer, 2010.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[CHP71] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. Concurrent Control with "Readers" and "Writers". *Commun. ACM*, 14(10):667–668, 1971.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[GCS11] Alkis Gotovos, Maria Christakis, and Konstantinos F. Sagonas. Test-driven development of concurrent programs using concuerror. In Kenji Rikitake and Erik Stenman, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*, pages 51–61. ACM, 2011.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian Partial-Order Reduction. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[God90] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* PhD thesis, UniversitÃľ de LiÃÍge, 1995.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009.

[Maz86] Antoni W. Mazurkiewicz. Trace Theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[SBM+13] Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. Efficient Verification of Distributed Protocols Using Stateful Model Checking. In *SRDS*, pages 133–142. IEEE, 2013.

[Val96] Antti Valmari. The State Explosion Problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[WYKG08] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.

Appendix A

Prototype Usage and Implementation

This chapter provides details about our prototype implementation in addition to chapter 5. First, we describe how to install and invoke our model checking tool, followed by a description of our implementation.

A.1 Usage

Our model checking tool consists of a compiler, CDL, for input programs and state search tool, dtool, which invokes specified algorithms on a transition system. The typical procedure for using our tool is as follows.

1. Specifying an input program, written in dlang, in a file <program>.d.
2. Invoking the compiler CDL on <program>.d, which generates a transition system representation in a file <program>.py.
3. Repeating steps 1–2 for each program to be analyzed.
4. Invoking the state search tool dtool on all program files <program>.py and specify the state search algorithms to be run on those systems.

Installation In order to prepare the use of our model checking tool, it is necessary to

- make available the ANTLR run time environment (antlr−4.4−complete.jar),
- adapt the Java class path for CDL,

  $ export CLASSPATH=".:<path␣to␣CDL>:<path␣to␣ANTLR>/antlr−4.4−complete.jar"

- make available an installation of Python 3,
- make available the files dtool.py, machine.py, and the module file of every algorithm to be used for dtool and,
- in case the Z3 variant of *ZPOR* is required, a working installation of Z3.

```
1 <"readers−writers" 21, x 1, l 20>
2 [x := 42]
3 <{20
4   [
5    if l[$] == 0
6    if x == 0
7   ]
8  }>
```

Listing A.1: The readers−writers example written in dlang.

Input language The syntax for our input language dlang is depicted in figures A.1 and A.2. As an example, the input for the readers−writers program with one writer and 20 readers from section 6.3.1 is given in listing A.1.

Line 1 specifies the optional program name, the number of threads (21), and the used variables with their respective lengths. In line 2, the program block for the writer thread is given. Lines 3 to 8 contain a *repetition block*, which specifies the program blocks of all reader threads. In line 3, the number of repetitions is given and in the following block, all occurrences of $ are replaced by the ID of the current repetition, starting at 0. In lines 5 and 6, conditional statements without branches are used to express read operations.

The dlang syntax is easily customizable by changing the file Dlang.g4, cf. section A.2.2.

Compiler invokation The following shell command can be used to compile dlang programs given in files <program∗>.d into transition system representations.

```
$ java mt.CDL <programs∗>.d
```

State search invokation The state search is invoked by executing the python script dtool and specifying on the command line the system files and algorithms to be used. For example, the following shell command invokes dtool on a transition system given in file program.py with *ZPOR* and *DPOR*.

```
$ ./dtool.py −a pp dp −s program.py
```

In general, the usage of dtool is as follows.

```
1 usage: dtool.py [−h] [−s [SYSTEMS [SYSTEMS ...]]]
2                 [−a [{dps,ppsf,fs,dp,pp} [{dps,ppsf,fs,dp,pp} ...]]]
3                 [−l {warning,info,debug,none}]
4
5 optional arguments:
6   −h, −−help show this help message and exit
7   −s [SYSTEMS [SYSTEMS ...]], −−systems [SYSTEMS [SYSTEMS ...]]
8                         run algorithms on these systems
9   −a [{dps,ppsf,fs,dp,pp} [{dps,ppsf,fs,dp,pp} ...]], −−algorithms [{dps,ppsf,fs,dp
        ,pp} [{dps,ppsf,fs,dp,pp} ...]]
10                        run these algorithms
```

```
11   -l {warning,info,debug,none}, --log {warning,info,debug,none}
12                         use this loglevel
```

In the directory where dtool is invoked, a directory logs is created (if it does not already exist). In this directory, a log file for measurement results with the naming scheme <date and time>-res is placed (it receives log messages of level warning or higher). Additionally, if the log level is info or debug, a file with the naming scheme <date and time>-log is placed in directory logs, which receives all log messages of the specified level.

A.2 Implementation

A.2.1 Algorithms

This section describes our implementations of state search algorithms. Commonly used classes are contained in module machine. Specifically, it contains the following classes.

Sequence
: The representation of a transition sequence. It contains an instance variable transitions which holds the transitions of the transition sequence. Algorithms extend this class in order to add functionality such as a dependency list or a program successor relation.

Machine
: This class is responsible for executing transitions. Specifically, it provides run_sequence, which executes a given transition sequence at a given state. The LazyPOR algorithms extend this class in order to add functionality for dependency tracking.

State
: The representation of a transition system's state. It contains the memory which consists of a program counter for each process and data memory for all declared variables. The following methods are provided.

  is_enabled
  : Returns true if and only if the specified transition is enabled at this state.

  get_all_enabled
  : Returns all transitions of the specified processes which are enabled in this state.

  next
  : Returns the next transition of the specified process in this state, or None if no such transition exists.

  apply
  : Applies the given transition to this state if the transition is enabled in this state. Returns whether the transition has been applied and the set of locations the transition has written to and has read from.

AbstractTransition
: This class represents transitions of a transition system. Each transition

⟨*abstract_program*⟩ ::= (.*? ⟨*duplicate_block*⟩)* .*?

⟨*duplicate_block*⟩ ::= `<{` ⟨*num_procs*⟩ .*? `}>`

⟨*num_proc*⟩ ::= ⟨*INT*⟩

⟨*program*⟩ ::= ⟨*declaration_list*⟩ ⟨*process_block*⟩+

⟨*declaration_list*⟩ ::= `<` (⟨*system_name*⟩)? ⟨*proc_count*⟩
`,` ⟨*declaration*⟩)* `,?` `>`

⟨*system_name*⟩ ::= ⟨*STRING*⟩

⟨*proc_count*⟩ ::= ⟨*INT*⟩

⟨*declaration*⟩ ::= ((`local` | `const`))? ⟨*ID*⟩ ⟨*length*⟩ ⟨*init_values*⟩

⟨*length*⟩ ::= ⟨*INT*⟩

⟨*init_values*⟩ ::= ⟨*INT*⟩*

⟨*process_block*⟩ ::= `[` ⟨*block*⟩ `]`

⟨*block*⟩ ::= ⟨*statement*⟩ ⟨*block*⟩ | ⟨*statement*⟩

⟨*statement*⟩ ::= (⟨*atomic_statement*⟩ | ⟨*conditional*⟩ | ⟨*loop*⟩)

⟨*atomic_statement*⟩ ::= (⟨*assignment*⟩ | ⟨*comp_swap*⟩ | ⟨*indexer*⟩)

⟨*assignment*⟩ ::= ⟨*asgn_target*⟩ `:=` ⟨*int_expr*⟩

⟨*indexer*⟩ ::= `indexer` `(` max=⟨*var*⟩ `,` size=⟨*var*⟩ `,` m=⟨*var*⟩ `,` w=⟨*var*⟩ `,`
h=⟨*var*⟩ `,` target=⟨*asgn_target*⟩ `,` nprocs=⟨*INT*⟩ `)`

⟨*conditional*⟩ ::= `if` ⟨*bool_expr*⟩ (`then` `{` ⟨*block*⟩ `}` (`else` `{` ⟨*block*⟩ `}`)?)?

⟨*loop*⟩ ::= `while` ⟨*bool_expr*⟩ `{` ⟨*block*⟩ `}`

⟨*bool_expr*⟩ ::= left=⟨*int_expr*⟩ op=(⟨*LT*⟩|⟨*GT*⟩|⟨*EQ*⟩) right=⟨*int_expr*⟩

⟨*int_expr*⟩ ::= `(` ⟨*int_expr*⟩ `)`
| ⟨*int_expr*⟩ (⟨*ADD*⟩|⟨*SUB*⟩|⟨*MUL*⟩|⟨*DIV*⟩|⟨*MOD*⟩) ⟨*int_expr*⟩
| ⟨*var*⟩ | ⟨*array*⟩ | ⟨*INT*⟩

⟨*asgn_target*⟩ ::= ⟨*var*⟩ | ⟨*array*⟩

⟨*var*⟩ ::= ⟨*ID*⟩

⟨*array*⟩ ::= ⟨*var*⟩ `[` ⟨*int_expr*⟩ `]`

Figure A.1: The dlang syntax (parser rules).

⟨*ID*⟩ ::= (a..z|A..Z)+(a..z|A..Z|0..9|-|_)*

⟨*STRING*⟩ ::= " (..~)* "

⟨*INT*⟩ ::= 0..9+

⟨*WS*⟩ ::= [\t\n\r]+ — channel(HIDDEN)

⟨*COMMENT*⟩ ::= // ~[\n\r]* [\n\r] — skip

⟨*MUL*⟩ ::= *

⟨*DIV*⟩ ::= /

⟨*MOD*⟩ ::= %

⟨*ADD*⟩ ::= +

⟨*SUB*⟩ ::= -

⟨*LT*⟩ ::= <

⟨*GT*⟩ ::= >

⟨*EQ*⟩ ::= ==

Figure A.2: The dlang syntax (lexer rules).

contains a pointer to its process, its position in the program (the program counter value for which it is enabled), a function which changes the memory in order to model the transition's effect, a set of static dependencies, and a list of program successors.

Transition(AbstractTransition)
: The execution of a transition is represented by a Transition. Each Transition (implicitly through inheritance) refers to an AbstractTransition.

  For example, a Sequence contains Transition instances and every Transition instance occurs at most once in a Sequence. In this way, transition sequences which contain transitions multiple times are represented.

Process(tuple)
: The representation of a process, defined by the list of its transitions.

System
: The representation of a transition system. It contains a list of its processes and the initial values for all variables.

The *ZPOR* algorithm is contained in module popor. It defines the following classes.

Trace
: The representation of a trace constraint system. It contains a graph representation of a transition sequence's dependency list. It is used to generate variations, using the algorithm described in chapter 5.

UniqueTransition(AbstractTransition)
: A variant of Transition with an additional counter for debugging.

POPORMachine(Machine)
: The extension of Machine adapted for LazyPOR. The instance variable ordering maps UniqueTransitions to their position in the previously executed transition sequence. The instance variable instances maps AbstractTransitions to the set of their occurrences in the previously executed transition sequence.

  The class provides the methods run_section_0, run_section_1, run_section_2, which correspond to executing the transition sequences u_1, v_1, and v_2 in the pseudo code of listing 3.1.

  The method split_at_dependency directly corresponds to *split-at-dependency*.

POPORSequence(Sequence)
: This class extends Sequence in order to add functionality for generating a trace constraint system, the program successor relation, and the dependency list for the represented transition sequence. Additionally, the method variations generates the variations of the represented transition sequence.

POPOR
: This class contains the methods explore and execute, which directly follow the corresponding pseudo code descriptions of listing 3.1.

The *DPOR* implementation is contained in module dpor. It contains the following class definitions.

ClockVector
: Clock vectors are represented as lists of lists: a list with an entry for each memory location contains a list with an entry for each process. This class provides the method get_empty which returns an empty clock vector.

AccessList
: Access lists correspond to the argument L in the pseudo code of [FG05, Figure 5]. They are represented as lists with an entry for every memory location.

DPORSequence(Sequence)
: This class extends Sequence in order to add functionality for executing transitions and recording resulting states for all transitions in the represented transition sequence.

DPOR
: This class contains the method explore, which directly corresponds to the respective procedure in the pseudo code of [FG05, Figure 5].

A.2.2 Compiler

The CDL implementation consists of the ANTLR grammar file Dlang.g4, the Java source file Compiler.java which holds most of the compiling functionality, the Java source file CDL.java which contains the main method of the compiler, and additional Java source files which are automatically generated by ANTLR.

In the ANTLR grammar file, *semantic predicates* are used to call back to Compiler which generates code during parsing. The compiler performs two passes, during the first pass, *repetition blocks* are substituted with concrete dlang code and during the second pass, the resulting code is compiled into a Python script which represents the corresponding transition system.

The following shell command can be used to recompile CDL after a change (here, mt is the directory holding all files of CDL).

```
$ cd mt && antlr4 -Dlanguage=Java Dlang.g4 && javac *.java
```

Appendix B

Detailed Measurement Results

readers−writers

Threads	$ZPOR_{time}$ (s)	$ZPOR_{exec}$	$DPOR_{time}$ (s)	$DPOR_{exec}$
2	0.001	2	0.002	3
3	0.003	4	0.027	26
4	0.009	8	0.657	493
5	0.021	16	27.463	16559
6	0.051	32	1750.778	866697
7	0.120	64		
8	0.280	128		
9	0.629	256		
10	1.433	512		
11	3.171	1024		
12	6.894	2048		
13	15.153	4096		
14	33.044	8192		
15	71.406	16384		
16	155.865	32768		
17	334.032	65536		
18	718.283	131072		
19	1534.372	262144		
20	3245.686	524288		
21	6982.460	1048576		

indexer

Threads	$ZPOR_{time}$ (s)	$ZPOR_{exec}$	$DPOR_{time}$ (s)	$DPOR_{exec}$
1	0.001	1	0.005	1
2	0.001	1	0.013	1
3	0.001	1	0.026	1
4	0.001	1	0.044	1
5	0.001	1	0.066	1
6	0.001	1	0.093	1
7	0.001	1	0.127	1
8	0.001	1	0.165	1
9	0.001	1	0.211	1
10	0.002	1	0.259	1
11	0.002	1	0.319	1
12	0.048	9	9.388	23
13	0.455	64	266.064	529
14	4.344	513	7069.726	12167
15	39.765	4096	190012.311	279841
16	361.107	32769		
17	3283.134	262144		
18	28764.877	2097152		
19	253524.515	16777216		
20				

indexer−small

Threads	$ZPOR_{time}$ (s)	$ZPOR_{exec}$	$DPOR_{time}$ (s)	$DPOR_{exec}$
1	0.001	1	0.002	1
2	0.001	1	0.005	1
3	0.002	3	0.022	2
4	0.004	4	0.068	4
5	0.013	9	0.187	8
6	0.031	16	0.471	16
7	0.077	33	1.158	32
8	0.180	64	2.758	64
9	0.415	129	6.409	128
10	0.943	256	14.501	256
11	2.105	513	33.284	512
12	4.747	1024	74.498	1024
13	10.535	2049	165.129	2048
14	23.056	4096	362.931	4096
15	51.090	8194	800.262	8192
16	108.559	16384	1751.365	16384
17	238.402	32770	3796.769	32768
18	518.973	65536	8147.626	65536
19	1100.982	131073	17929.869	131072
20	2356.374	262144	38178.370	262144

branching

Threads	$ZPOR_{time}$ (s)	$ZPOR_{exec}$	$DPOR_{time}$ (s)	$DPOR_{exec}$
1	0.001	1	0.001	1
2	0.085	49	0.031	23
3	1.032	393	0.773	299
4	15.594	5555	20.336	5731
5	466.504	124123	539.520	112168
6	9636.756	2085951	25774.401	2407542
7	188331.801	33153882		