Automated Systematic Testing Methods for Multithreaded Programs

Kari Kähkönen
Automated Systematic Testing Methods for Multithreaded Programs

Kari Kähkönen

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall TU2 of the school on 2 February 2015 at 12.

Aalto University
School of Science
Department of Computer Science and Engineering
Abstract

This thesis considers the problem of how the testing of multithreaded programs can be automated. One of the reasons multithreaded programs are difficult to test is that their behavior does not only depend on input values but also on how the executions of threads are interleaved. Typically this results in a large number of possible executions through a multithreaded program that is so large that covering them all, even with the help of a computer, is infeasible. Fortunately it is often not necessary to test all combinations of input values and interleavings as many of them cause the program to behave in a similar way. This opens up the research question of how to automatically cover interesting properties of the program under test while trying to keep the number of test executions small.

This work studies how two different approaches, dynamic symbolic execution and net unfoldings, can be combined to automatically test multithreaded programs. Dynamic symbolic execution is a popular approach to automatically generate tests for sequential programs. Net unfoldings, on the other hand, can be used in verification of concurrent systems. The main contributions of this thesis fall under three categories. First, two net unfolding based approaches to systematically cover the local reachable states of threads are presented. Second, the thesis describes how global reachability properties, such as deadlock freedom, can be checked from an unfolding of a program. Third, a lightweight approach to capture abstract states of a multithreaded programs is presented. The method can be used to stop a test execution if it reaches an already covered abstract state.

To evaluate the new approaches, they are compared against an existing partial order reduction based testing algorithm. This is done by applying the algorithms to a number of multithreaded programs. Based on the results, the new algorithms are competitive against more traditional partial order reduction based methods and can in some cases outperform existing approaches by a considerable margin.

Keywords
dynamic symbolic execution, unfoldings, automated testing, partial order reduction

ISBN (pdf) 978-952-60-6039-2
ISSN-L 1799-4934
ISSN (printed) 1799-4934
ISSN (pdf) 1799-4942
Location of publisher Helsinki
Location of printing Helsinki
Year 2015
Pages 161
Säikeistettyjen ohjelmien testaus on usein haastavaa. Yksi syy tähän on se, että kyseisten ohjelmien käyttäytyminen ei riipu ainoastaan ohjelmalle annettavista syötteistä vaan myös siitä, miten ohjelman säikeiden suoritukset lomittuvat toisiinsa nähden. Tyypillisesti tämä tarkoittaa, että säikestetyn ohjelman suoritukset voivat poiketa toisistaan niin monilla eri tavoin, että kaikkien suoritusmahdollisuuksien testaaminen ei ole mahdollista kohtuullisessa ajassa edes tietokoneen avustuksella. Onneksi useissa tapauksissa kaikkia syötearvojen ja säikeiden suoritusjärjestyksen yhdistelmiä ei tarvitse erikseen testata. Tämä johtuu siitä, että tarkasteltava ohjelma toimii samankaltaisesti useilla kyseisillä yhdistelmillä. Tämä herättää tutkimuskysymyksen siitä, miten kiinnostavia ominaisuuksia säikestetystä ohjelmasta voidaan automaattisesti testata mahdollisin pienellä testimäärällä.


Avainsanat dynaaminen symbolinen suoritus, verkkojen aukikerintä, automaattinen testaus, osittaisjärjestysreduktio

| ISSN-L | 1799-4934 | ISSN (painettu) | 1799-4934 | ISSN (pdf) | 1799-4942 |
| Julkaisupaikka | Helsinki | Painopaikka | Helsinki | Vuosi | 2015 |
Many years ago as an undergraduate student I took a course on parallel and distributed systems. Back then that course was organized by my current thesis supervisor, Assoc. Prof. Keijo Heljanko, and it was my first introduction to the topic of computer aided verification. The course piqued my interest and I decided that I wanted to learn more. Little did I know that that decision would set me on a course that eventually led me to writing this thesis.

The work presented in this thesis is the result of the research I have done as a postgraduate student first at the Department of Information and Computer Science and later at the Department of Computer Science and Engineering at Aalto University. The research presented in this work has been funded by the ARTEMIS-JU and Academy of Finland (projects 128050 and 139402), Metso Automation and the LIME project joint with Tekes (Finnish Funding Agency for Technology and Innovation) and industrial partners Nokia, Conformiq, Space Systems Finland and Elektrobit.

I would like to thank Assoc. Prof. Keijo Heljanko and Prof. Ilkka Niemelä for giving me the opportunity to continue the research on automated testing that I started for my Master’s Thesis. The many discussions I have had with Assoc. Prof. Heljanko and his continuous support have been invaluable in completing this work. I am also grateful to everyone who participated in the research projects on automated testing at Aalto University. In particular, I would like to thank Olli Saarikivi for his collaboration on developing testing tools and for many useful discussions.

I also want to thank my pre-examiners, Prof. Bengt Jonsson (Uppsala University, Sweden) and Dr. Stefan Haar (INRIA, France), for their expert evaluation of this thesis and Prof. Ganesh Gopalakrishnan (University of Utah, USA) for agreeing to act as an opponent in the defense of this
thesis.

Finally, I want to thank Tuulia and my family for all the support they have given me during my postgraduate studies.

Espoo, December 16, 2014,

Kari Kähkönen
Author’s Contribution

This thesis unifies and extends the work presented in the following publications.


The author is responsible for the algorithms described in the papers above as well as the proofs, experiments and writing of the papers. The new algorithms in the papers are compared against an improved dynamic partial order reduction algorithm that is due to Olli Saarikivi. Assoc. Prof. 1

Heljanko provided comments to the manuscripts, guidance and supervision.

Additionally the experiments in this thesis were performed on a tool done by the author for the following publications. All the algorithms used in the experiments in this thesis have been implemented by the current author.


The author is responsible for the writing and the experiments in the papers above. The tool described in the paper is joined work with the current author, Olli Saarikivi, Janne Kauttio and Tuomas Launiainen. The current author is the main developer of the sequential version of the LCT tool and Olli Saarikivi is responsible for the multithreaded extension. For the experiments in this thesis the current author implemented an independent version of the support for multithreading. Assoc. Prof. Heljanko and Prof. Niemelä provided comments to the manuscripts, guidance and supervision.

In addition to the work presented in the publications above, this thesis includes an SMT translation based approach for global state reachability checking that has not been published prior to the writing of the thesis. The initial idea of the approach is due to Assoc. Prof. Heljanko but the details of the translation are by the current author. The deadlock and other property checking translations are also by the current author. Furthermore, the thesis includes correctness proofs of the testing algorithms described in the thesis. The author is responsible of these proofs.
The current author has also co-authored the following related publications that are not discussed in this thesis to keep the scope of the thesis focused.


The dynamic partial order reduction approach presented in [SKH12] is used in the experimental section of this thesis. The algorithm is due to Olli Saarikivi and the current author only provided comments to the paper.
Author's Contribution
Preface

Contents

1. Introduction
   1.1 Related Approaches ........................................... 9
   1.2 Contributions .................................................. 13

2. Background
   2.1 Describing Multithreaded Programs ............................ 15
   2.2 Dynamic Symbolic Execution .................................. 17
   2.3 Testing Multithreaded Programs with DSE .................... 23
   2.4 Using Partial Order Reductions with DSE .................... 24

3. Unfolding Based Testing
   3.1 Motivation .................................................... 35
   3.2 Petri nets and Unfoldings .................................... 36
   3.3 Modeling Multithreaded Programs with Petri Nets ............ 40
   3.4 Systematic Exploration of Execution Paths ................... 46
   3.5 Integrating the Algorithm with Dynamic Symbolic Execution 51
   3.6 Computing Corresponding Events .............................. 53
   3.7 Limitations .................................................... 54
   3.8 Correctness .................................................... 55
   3.9 Discussion ..................................................... 61

4. Efficient Possible Extensions Computation ........................... 63
   4.1 Basic Idea ..................................................... 63
   4.2 Structural Connection Between Read And Write Events .......... 65
   4.3 Structural Connection Between Lock Events ................... 68
   4.4 Possible Extensions from Thread Conditions ................... 71
## 4.5 Possible Extensions from Shared Variable Conditions

- Page 73

## 4.6 Possible Extensions from Lock Conditions

- Page 76

## 4.7 Checking For Concurrency

- Page 78

### 5. Contextual Unfolding Based Testing

- **5.1 Motivation**
  - Page 79

- **5.2 Contextual Nets**
  - Page 80

- **5.3 Modeling Test Executions with Contextual Unfoldings**
  - Page 82

- **5.4 Systematic Testing with Contextual Unfoldings**
  - Page 84

- **5.5 Computing Input Values and Schedules for Test Executions**
  - Page 86

- **5.6 Computing Possible Extensions**
  - Page 88

- **5.7 Handling Dynamic Thread Creation**
  - Page 92

- **5.8 Correctness**
  - Page 93

### 6. Testing Global State Reachability Properties

- **6.1 Motivating Example**
  - Page 97

- **6.2 SMT Translation Based Reachability Checking**
  - Page 99

- **6.3 Checking Deadlocks Directly from Unfoldings**
  - Page 105

- **6.4 Correctness**
  - Page 109

### 7. Lightweight Stateful Approach For Testing Multithreaded Programs

- **7.1 Finite Prefixes of Unfoldings**
  - Page 114

- **7.2 Capturing Program States with Petri Nets**
  - Page 118

- **7.3 Naive Stateful Testing Algorithm**
  - Page 120

- **7.4 Stateful Unfolding Based Testing Algorithm**
  - Page 123

- **7.5 Discussion**
  - Page 126

### 8. Experiments

- **8.1 Implementation**
  - Page 129

- **8.2 Evaluation Subjects**
  - Page 130

- **8.3 Experimental Evaluation**
  - Page 131
    - **8.3.1 Stateless Algorithms**
      - Page 131
    - **8.3.2 SMT Translation Based Checking of Global Properties**
      - Page 135
    - **8.3.3 Stateful Algorithms**
      - Page 136

### 9. Conclusions

- **9.1 Summary**
  - Page 141

- **9.2 Future Work**
  - Page 143

### Bibliography

- Page 145
1. Introduction

Multithreading is an important part of software development as devices ranging from mobile phones to desktop computers have processors with multiple cores. Developing reliable multithreaded programs, however, is typically more challenging than developing sequential ones. This is because it is easy to overlook some subtle and unintended interaction between concurrently executing threads that causes a program, for example, to crash. Manual inspection of a complex sequential program, let alone a multithreaded one, requires a considerable amount of work and it is easy to miss some possible execution scenarios. Because of this, there is a need for tools that can automatically analyze a given program for errors. Such tools have not only the potential to reduce the number of errors in software but also to reduce the cost of software development as defects can be detected and fixed earlier.

In this work we develop methods to automatically and systematically test different behaviors of multithreaded programs where threads communicate with each other through shared memory. In principle it is easy to systematically cover all reachable states of such programs with test executions. One just needs to consider all possible combinations of input values the program can read and for each of these combinations explore all different ways to interleave the executions of the threads. In practice, however, such an approach is infeasible as just two 32-bit integer inputs generate \((2^{32})^2\) possible input value combinations and \(n\) independent program statements that can be executed concurrently can be interleaved in \(n!\) different ways. This is a so called state explosion problem [Val96] that testing and other verification approaches have to face.

Fortunately it is often not necessary to explore all such combinations to find specific types of errors. The input space of the program can be partitioned into equivalence classes where each test execution with in-
puts from the same equivalence class follows the same control flow path through the program (with regard to some interleaving of threads). One way to achieve such partitioning is to use symbolic execution [Kin76] that expresses the equivalence classes through constraints. For example, symbolic execution could determine that in a program reading a single input value, all inputs that are between one and five cause the program to take the true branch at the first conditional statement encountered and false branch at the second one.

The interleavings of threads can also be partitioned into equivalence classes. This is because threads can perform operations that are independent of each other and regardless of the order in which they are executed, the program ends up in the same state. For example, two threads can read the same shared variable without affecting each other. One possible partitioning is obtained by having two interleavings belong to the same class if they can be obtained from one another by performing a number of swaps between adjacent independent operations. Such equivalence classes are often called Mazurkiewicz traces [Maz86, Die95]. Covering one interleaving from each Mazurkiewicz trace preserves properties such as assertion violations and deadlocks. The partitioning to equivalence classes creates a partial order over all possible interleavings and approaches taking advantage of them are typically called partial order reduction methods. Different partial order reductions can preserve different properties of the given program [Val91, Val97, Pel93, God96].

Different interleavings of a multithreaded program can be expressed as a computation tree. Partial order reduction algorithms can then be used to determine which parts of the computation tree can be left untested. In other words, test executions can be used to cover a reduced version of the computation tree. There is also a different approach to limit the number of interleavings that need to be tested. Instead of covering a computation tree where some of the subtrees have been removed, it is possible to construct a representation of all interleavings that is more succinct than the full computation tree. The test executions can then be limited to cover this succinct representation. One approach to obtain such a representation is to use net unfoldings which were first introduced in the context of verification algorithms by McMillan [McM92]. Unfolding is a method originally developed for Petri nets but it can be applied for other models of concurrency as well. An unfolding of a Petri net is another net that represents the reachable markings of the original net with a simpler acyclic
Introduction

The main topic of this thesis is to investigate how a more recent version of symbolic execution, called dynamic symbolic execution (DSE) [GKS05], can be combined with unfoldings in order to test multithreaded programs. In particular, we show how such a combination can be used to cover the local reachable states of threads. This allows errors such as assertion violations and uncaught exceptions to be detected during the automated testing process. As an unfolding symbolically represents all interleavings of threads, we also show that it is possible to predict violations of global properties from an unfolding generated during testing.

The main testing approaches developed in this work are stateless. This means that they will explore the same state space multiple times if, for example, two non-equivalent interleavings lead to the same state. To alleviate this problem, we develop a lightweight approach to capture partial state information by modeling program behavior as Petri nets. Based on the results obtained in this thesis, unfolding based approaches are competitive against more traditional partial order reduction methods in the context of testing multithreaded programs.

1.1 Related Approaches

Testing Multithreaded Programs Perhaps the simplest approach to automate testing is to use random testing [BM83, FM00]. This approach, however, can easily miss errors that can be observed only with specific input values or specific interleavings of threads [OH96]. Random testing can also lead to redundant test executions that explore the same program behavior multiple times.

The closest work to the one presented in this thesis is the dynamic partial order reduction (DPOR) algorithm by Flanagan and Godefroid [FG05] and the race detection and flipping algorithm by Sen and Agha [SA06b]. Both of these algorithms generate at least one execution for each Mazurkiewicz trace in the program and they have been combined with DSE [SKH12, Sen06]. Recently an optimal version of the DPOR algorithm has been introduced in [AAJS14] that performs additional computation to guarantee that any Mazurkiewicz trace is not explored more than once. The authors of [AAJS14] also describe in the paper a near optimal DPOR algorithm, called Source-DPOR, that is not computationally more expensive than the original DPOR algorithm [FG05] but can still lead to a con-
siderably smaller number of test executions. Both DPOR and race detection and flipping will be discussed in more detail in Chapter 2. DPOR has also been applied for other kinds of programs in addition to multithreaded ones.

Farzan et al. [FHRV13] have present an alternative approach to use DSE to test concurrent programs. Their approach aims to obtain maximal code coverage by systematically exploring so called interference scenarios, where interference means that a some thread reads a value that has been generated by another thread. Another related tool is Symbolic Java PathFinder [PMB+08, PVB+13] that uses symbolic execution and has a support for multithreaded programs. The tool performs partial order reduction using its own algorithms.

The approaches above generate tests by analyzing a concrete program. Another way to generate tests is to analyze a model of the system under test. Such an approach is called model based testing. The models are usually constructed in a modeling language and tests are then generated from the model to check whether the system under test conforms to the model. Related to our work, Petri nets and their unfoldings can be used to specify systems and to generate test suites as presented in [dLHL13]. DSE has also been used in model based testing of distributed and concurrent systems in a semiautomatic way [GAJS09].

As exhaustive testing of all possible behaviors of programs is rarely feasible, different heuristics to guide testing and to limit the space of input values and interleavings have been suggested. For example, the Korat tool [MMMK07] uses predicates provided by the user to construct complex test inputs that satisfy the predicates. CHESS [MB+08], on the other hand, uses preemption bounding and prioritizes interleavings with fewer preemptions. In [XTdHS09] an approach to guide DSE is presented that uses fitness functions to guide testing towards a test target. Such heuristics are orthogonal to our work and could be combined with the approaches presented in this thesis.

**Formal Verification.** There also exists an extensive amount of work on methods that can guarantee that specific types of errors are absent in the program. One such approach is model checking [CGP99, BK08] that automatically and exhaustively checks whether a formal model of a system satisfies its specification. Such specifications can, for example, be given as a temporal logic formula. Model checking can be applied to a model of a program but there are also approaches that directly check programs writ-
ten in full programming languages [God05, YCGK07, PMB+08, PVB+13].

Model checking approaches often suffer from a so called state explosion problem as for large systems the number of states is often too large to be explored exhaustively. One way to limit the size of the state space is to use bounded model checking (BMC) [BCCZ99]. In the context of software it is possible, for example, to check programs where the loops have been unwound up to a given bound. This is similar to the approaches developed in this thesis where we bound the length of test executions. Concurrency in these approaches can be handled, for example, by using partial orders as in CBMC [CKL04], explicitly exploring the schedules as in ESBMC [CF11] or by translating the program into a sequential one [QW04, LR09] that captures a bounded number of context switches like in Lazy-CSeq [ITF+14]. In a sense our unfolding based algorithms can be seen as approaches that on-the-fly construct and explore a model of a multithreaded program by unwinding loops up to a given bound.

As proving properties directly from a concurrent system can be difficult and does not scale well, verification approaches such as [CNR11, GPR11] use abstractions of the system being verified. The use of abstractions, however, can sometimes result in false warnings of errors. That is, an error state can be reachable in the abstraction but not in the real system. One popular approach to automatically improve a model is counterexample guided abstraction refinement [CGJ+00, BMMR01, BPR01]. The approaches in this thesis are based on executing the concrete program under test and therefore any detected errors are guaranteed to be real ones.

There are also approaches that combine guarantee based verification with dynamic approaches that analyze executions. For example, it is possible to guide abstraction refinement based on information obtained by test executions and vice versa as shown in [GHK+06, BNRS08, GNRT10].

**State Capturing.** There exists a large number of verification algorithms that are based on state space exploration. Remembering which states have been covered can significantly speed up such approaches but it also requires a considerable amount of memory. In approaches like model checking where the real system can be abstracted, storing the visited states is feasible. In systematic testing of real-world multithreaded programs, however, storing the explored states can quickly exhaust all available memory. Even though methods to alleviate this problem have been developed (e.g, compression and hashing [Hol97] and selective caching [BLP03]), many testing tools rely on stateless exploration. The problem with state-
less testing, even when combined with partial order reductions, is that part of the state space may be explored multiple times. The lightweight state capturing approach developed in this work balances between complete state capturing and stateless search.

Yang et al. [YCGK08] propose a related lightweight approach to capture states at runtime that is based on tracking changes between successive local states without storing full state information. In their approach the captured local states are abstract but they capture the shared portion of the state concretely. Unlike our approach, their approach cannot directly be combined with DSE. They also describe a stateful DPOR algorithm based on their state capture approach. To guarantee the soundness of their algorithm, additional computation needs to be performed to make sure that any subset of the state space is not missed. In our work we use standard unfolding techniques to guarantee soundness.

It is also possible to capture and match symbolic states (e.g., resulting from symbolic execution). Anand et al. [APV06] propose a combination of subsumption checking and abstractions for data structures such as lists and arrays. Such approaches are considerably more heavyweight compared to the approaches presented in this thesis. This allows them to match states that our approach cannot. An alternative way to reduce the number of states that need to be explored when using symbolic execution is to use state merging [KKBC12]. In this approach multiple execution paths are expressed symbolically instead of exploring them separately. This, however, makes the symbolic constraints more complex and therefore more demanding to solve.

Checking Programs for Property Violations In our work we can check different properties of multithreaded programs from an unfolding constructed during testing. An approach related to this is predictive analysis [SRA03, SRA06] that takes an execution trace as input and computes other feasible interleavings of the events in the trace in order to determine if a given property is violated in any of the predicted interleavings.

There exists approaches based on predictive analysis and runtime monitoring that can be used to detect properties such as data races [FF10, HMR14], deadlocks [AS06], atomicity violations [WLGG10] and violations of safety properties [SRA05, WKL+11]. These approaches typically analyze one execution trace at a time and predict other valid executions from the observed trace. These approaches often use a weaker form of the happens before relation than we use in our algorithms. This allows them to
predict some global states from a single execution that our approach cannot. There are also predictive analyses that can be used to detect safety property violations described, for example, with linear temporal logics whereas our work focuses solely on properties based on reachability of global states. However, our approach is able to take advantage of information over multiple test runs and symbolic constraints based on input values to predict additional global states.

1.2 Contributions

This thesis summarizes and extends the work presented in earlier publications by the author. The main contributions of the thesis are the following.

1. In Chapter 3 we present a novel testing algorithm that combines unfolding and dynamic symbolic execution to cover feasible control flow paths of threads and to detect assertion violations. We show that the new approach can in the best case cover the program under test with even an exponentially smaller number of test executions than there are Mazurkiewicz traces of the program. The algorithm has originally been presented in [KSH12] and [KSH14]. In Chapter 3 we also provide a correctness proof for the algorithm which has not been published prior to the writing of this thesis.

2. In Chapter 4 we show that a computationally expensive part of unfolding algorithms, called possible extensions computation, can be done efficiently in our new algorithm. The reason for this is that it is possible to take advantage of the restricted structure of the unfoldings that our testing algorithms generate. The possible extensions algorithm has been presented originally by the author in [KSH12] and [KSH14] but many details are discussed here for the first time.

3. In Chapter 5 we further reduce the number of test executions needed by unfolding based testing approaches by considering contextual nets which are an extension of Petri nets. We show that by using contextual nets, an unfolding of a program can sometimes be considerably more compact when compared to using regular nets. We also extend the unfolding based testing approach to programs that can create threads dynamically which is not supported by the algorithm presented in Chap-
ter 3. The approaches in this chapter were first presented in [KH14b]. In this thesis we additionally prove the correctness of the contextual unfolding based algorithm.

4. In Chapter 6 we show that it is possible to determine if a program contains deadlocks or if some shared state is reachable in the program by analyzing its unfolding. We present two approaches in the chapter: an algorithm that searches deadlocks directly from the unfolding and a SMT translation based approach where SMT solvers are used as the search engine. The direct search algorithm for deadlocks has originally been presented in [KSH14]. The other parts of the chapter have not been published prior to the writing of this thesis.

5. In Chapter 7 we present an approach to capture partial state information by modeling programs as Petri nets. We then describe how such a Petri net model can be used to further improve the algorithm presented in Chapter 3 such that it in some cases needs to perform substantially less test executions. This is possible because the new testing algorithm is able to execute tests on the model and is able to use cut-off events in the generated unfolding. The approaches presented in this chapter have originally been published in [KH14a].

6. In Chapter 8 we provide an experimental evaluation of the testing approaches developed in this thesis and compare them with an existing testing algorithm that generates a test for every Mazurkiewicz trace. Based on the experiments, the unfolding based testing approaches are viable and practical algorithms that can offer competitive performance to more traditional partial order reduction approaches. Furthermore, in some cases they can be significantly faster than approaches aiming to cover all Mazurkiewicz traces while still being able to detect all assertion violations and deadlocks.
2. Background

In this chapter we introduce a simple multithreaded programming language that we will use to describe the programs under test throughout this thesis. We then describe how such programs can be tested with dynamic symbolic execution that is an approach that will be used as the basis for the new testing algorithms developed in this work. We also discuss existing partial order reduction approaches that have been combined with dynamic symbolic execution.

2.1 Describing Multithreaded Programs

To simplify the descriptions of the testing algorithms presented in this work, we introduce a simple multithreaded language with integer-valued variables. The syntax of this language is shown in Table 2.1 and can be seen as a subset of imperative programming languages such as C or Java. We assume that programs consists of a finite number of threads that communicate with each other through shared memory. There are two types of variables in the language: variables local to a thread and shared variables. To differentiate the variable types, we write local variables with lowercase letters and shared variables with capital letters. We further assume that the number of threads and shared variables is fixed when the program starts. Testing programs that create threads and shared variables dynamically is discussed separately later in this thesis.

We assume that a thread can use a shared value only by assigning it first to a local variable. Similarly we assume that a thread can update a shared value only by assigning to it either a constant value or a value from a local variable. Programs written with proper programming languages can be automatically modified to satisfy these assumptions. For example, an if-statement that depends on a value of a shared variable can
Table 2.1. Simplified language syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ::= Stmt*</td>
<td>(thread)</td>
</tr>
<tr>
<td>Stmt ::= lv := e</td>
<td>SV ::= lv</td>
</tr>
<tr>
<td>while (b) {Stmt}</td>
<td>if (b) else {Stmt}</td>
</tr>
<tr>
<td>e ::= lv</td>
<td>SV</td>
</tr>
<tr>
<td>b ::= true</td>
<td>false</td>
</tr>
<tr>
<td>lv = lv</td>
<td>lv ≠ lv</td>
</tr>
<tr>
<td>lv &lt; lv</td>
<td>lv &gt; lv</td>
</tr>
<tr>
<td>lv ≤ lv</td>
<td></td>
</tr>
<tr>
<td>op ∈ {+, -, *, /, mod, ...},</td>
<td></td>
</tr>
<tr>
<td>lv is a local variable,</td>
<td></td>
</tr>
<tr>
<td>SV is a shared variable,</td>
<td></td>
</tr>
<tr>
<td>lc is a lock identifier and,</td>
<td></td>
</tr>
<tr>
<td>c is a constant</td>
<td></td>
</tr>
</tbody>
</table>

be replaced with statements that read the value of the shared variable to a temporary local variable and then branch the execution based on the value of the temporary variable. Although we limit our discussion to this simple language, the algorithms described in this work can handle language features such as function calls, recursion and goto statements as well.

Semantics A (global) state of a multithreaded program consists of a local state of each thread, shared state and program counters of each thread. A local state is a map from local variables to values. The shared state maps shared variables to values and lock identifiers to boolean values representing whether the lock has been acquired or not. A program counter maps a thread identifier to a program location (i.e., to a line of code).

A multithreaded program starts its execution with an unique initial state and can change its state by executing statements. We call the execution of a statement a local operation if it accesses only the local state of the executing thread and a global operation if it accesses the shared state. The global operations considered in this work are reading and writing of shared variables and acquiring and releasing locks. An execution of a multithreaded program is a (possibly infinite) sequence of operations.
performed by the threads in the program. A control flow path of a thread corresponds to a sequence of program locations that the thread visits during an execution.

We define an execution step to be a part of an execution of a single thread that starts with a global operation and is followed by all the subsequent local operations in the execution until the next global operation is reached. An initial execution step of a thread corresponds to executing a sequence of local operations from the thread’s initial local state until the first global operation is reached. By considering only sequences of execution steps, a testing algorithm can avoid exploring interleavings of local operations which is unnecessary when checking for assertion violations or deadlocks. This approach is used in many tools and algorithms such as Verisoft [God97] and DPOR [FG05].

The execution of a thread is blocked if it cannot execute the statement specified by its current program counter value. In the language presented in Fig 2.1 the only statement type that can block is the lock statement. A program contains a deadlock if there exists an execution after which at least one thread is blocked and any thread that is not blocked has reached its final state by executing an end statement. The semantics of individual operations are the obvious ones and not defined formally here.

### 2.2 Dynamic Symbolic Execution

A sequential program can be considered to be a program described in our simplified language such that it consists of only one thread. In such programs the only sources of nondeterminism are the input values from the environment. A naive way to systematically test a sequential program is to perform test executions with all possible input values. As discussed in Chapter 1, this is rarely feasible. Furthermore, typically many input values cause the program to follow the same control flow path which can lead to redundant test executions.

It is possible to represent all control flow paths of a sequential program as a control flow graph. As an example, Fig. 2.1 shows a simple program and its control flow graph where each node is labeled with the corresponding program location. One way to systematically explore all control flow paths is to unwind the control flow graph into a computation tree and perform a test execution for each path in the tree. However, a control flow graph as presented in Fig. 2.1 is an overapproximation of feasible control
flow paths unless data values are taken into account. This means that some paths through the control flow graph cannot be followed no matter what input values are used. In our example control flow graph, the path corresponding to executing lines 1 and 2 and then jumping to the end is infeasible because the value of variable \( i \) is less than 2 when the while-statement is executed.

One approach to systematically explore the feasible control flow paths of a sequential program is to use dynamic symbolic execution (DSE) [GKS05], which is also known as dynamic test generation, directed testing and concolic testing. DSE, or a variation of it, has been used in many tools such as PEX [TdH08], SAGE [GLM12], CUTE [Sen06], KLEE [CDE08] and GKLEE [LLS+12]. The basic idea in DSE is to perform test executions where the program under test is executed both concretely and symbolically at the same time. The information obtained from symbolic execution is used to group executions together that follow the same control flow path. DSE then systematically explores one execution from each group. Typically it is not feasible to cover all control flow paths as there can be a large or even an infinite number of them. However, compared to random testing, DSE often achieves better coverage as it can generate inputs for cases requiring specific input values that are unlikely to be generated randomly.

Symbolic execution is performed by using symbolic counterparts of the local and shared states where variables are mapped to symbolic value expressions instead of concrete values. Each evaluation of the input statement returns a fresh symbolic input value. The assignment operations update the symbolic local and shared states by evaluating the right hand
side of the assignment symbolically. For example, if an input value is assigned to a variable and then the value of this variable is incremented by one, the resulting symbolic value expression can be $input_{i+1} + 1$, where $input_{i}$ is a fresh symbolic input value returned by the input statement.

At conditional statements DSE follows the true or false branch taken by the concrete execution and constructs a symbolic constraint that describes the input values that cause the program to take the same branch. For example, if a conditional statement $\text{if } (x > 0)$ is executed such that the true branch is followed and $x$ has the symbolic value expression $input_{i+1} + 1$, then the generated symbolic constraint is $input_{i+1} + 1 > 0$. If the result of the conditional statement does not depend on any input values (i.e., it is executed on purely concrete values), no symbolic constraint is generated.

Each control flow path can be represented by a path constraint which is a conjunction of all the symbolic constraints resulting from executing the path symbolically. A path constraint describes all the input values for which the program follows the corresponding control flow path. DSE explores all the control flow paths of a program systematically by typically performing a random test execution first. The input values for a subsequent test execution can be obtained by negating one symbolic constraint on the path constraint of the current test execution and removing any symbolic constraints after the negated constraint. The resulting path constraint corresponds to a control flow path where the previously explored path is followed up to some conditional statement after which a different branch at that statement is taken. If the path constraint is unsatisfiable, no input values exists that cause the program to follow the represented path. In traditional DSE the symbolic constraint to be negated is the last one in the path constraint such that its negation has not already been explored. This corresponds to exploring the control flow paths of a program in depth first order.

Example 1. To illustrate how DSE works, let us consider the program shown in Fig. 2.1. First a test execution with random input values is performed to explore an initial execution path. Let us assume that the randomly generated input values are $x = 5$ for the first iteration of the while loop and $x = 10$ for the second. Note that the condition of the while-statement never depends on input values and therefore executing the statement symbolically does not result in a symbolic constraint. The condition of the if-statement guarding the error-statement, however, depends on input values. In the first iteration of the while-loop, $x$ gets a
Background

fresh symbolic input value \( \text{input}_1 \) at line 3. Executing line 4 updates the symbolic value of \( x \) to \( \text{input}_1 + 5 \). The concrete execution takes the false branch at the if-statement as \( 5 + 5 \neq 0 \). Symbolic execution of the statements results in a symbolic constraint \( \text{input}_1 + 5 \neq 0 \). After executing line 7, the program executes the body of the while-loop again and similarly as before, a new symbolic constraint is generated. The resulting path constraint for the first test execution is \( \text{input}_1 + 5 \neq 0 \land \text{input}_2 + 5 \neq 1 \).

DSE then negates the last constraint to obtain a new path constraint: \( \text{input}_1 + 5 \neq 0 \land \text{input}_2 + 5 = 1 \). One satisfying assignment for the new path constrain is \( (\text{input}_1 = 0, \text{input}_2 = -4) \). Executing the program with these values causes it to follow the same control flow path as the initial execution except that when the body of the while-loop is executed for the second time, the true branch at the if-statement is followed. This execution reaches the error statement. The path constraint for the second test execution is in this case the same as the one used for computing the input values. It could also have contained additional constraints if the execution had encountered additional conditional statements. To continue testing, the constraint \( \text{input}_2 + 5 = 1 \) in the current path constraint is discarded as its negation has already been explored. The remaining constraint \( \text{input}_1 + 5 \neq 0 \) is then negated to get \( \text{input}_1 + 5 = 0 \). Any concrete input value satisfying this constraint forces the program to follow the true branch and reach the error at the if-statement on the first iteration of the while-loop.

Exploring the paths in depth first order has the advantage that it is memory efficient. That is, only the information regarding the last test execution needs to be stored. However, when testing large programs, the number of distinct control flow paths is typically too large to be covered exhaustively. In these cases exploring the paths in depth first order may restrict the testing to only some localized part of the program. That is, all test executions follow the same true or false branch at the first conditional statement until all possible control flow paths after the branch are explored. One way to address this issues is to represent all the feasible control flow paths as a symbolic execution tree that contains branches for each of the constructed symbolic constraint and its negation. A test execution then explores one path of this tree and for a subsequent test execution any unexplored branch along the already explored paths can be selected as a test target. As an example, Fig. 2.2 shows an symbolic execution tree of the program in Fig. 2.1. Storing all the unexplored branches
Figure 2.2. Symbolic execution tree of the program in Fig. 2.1

makes it possible to use heuristics to guide the testing process towards interesting executions paths [XTdHS09]. The disadvantage of the approach is the increase in memory usage as additional information of the unexplored branches needs to be stored.

The path constraints generated by DSE are typically solved using SMT-solvers. The constraints can be represented in different theories supported by the solvers. Examples of such theories are linear integer arithmetic and fixed size bit vectors. The choice of the theory affects how accurately the constraints can represent the control flow paths. For example, overflows of integer values can be captured naturally with bit vectors but not with linear integer arithmetic. In this work we assume that fixed size bit vectors are used to represent the integer values of our simple programming language. Bit vectors have also the benefit that non-linear constraints are decidable unlike with linear integer arithmetic.

Example 2. As a second example, let us consider the program in Fig. 2.3. Let us assume that the randomly generated input values for the initial test execution are \( x = 5 \) and \( y = 0 \). The resulting concrete test execution takes the true branch on line 3 and the false branch on line 4 after which the program terminates. The corresponding symbolic execution generates symbolic constraints at both of the if-statements as their results depend on input values. For the first if-statement the generated constraint is \( \text{input}_1 > \text{input}_2 \) and for the second, \( \text{input}_1 \cdot \text{input}_2 \neq 1452 \). For the second test execution DSE can attempt to follow either the false branch at the first if-statement or the true branch at the second one. Let us assume that the latter alternative is selected. The path constraint for the corresponding control flow path is \( \text{input}_1 > \text{input}_2 \land \text{input}_1 \cdot \text{input}_2 = 1452 \) and one possible solution is \( \text{input}_1 = 726 \) and \( \text{input}_2 = 2 \). The concrete execution with these input values takes the true branches at the first two if-statements and the false one at the third. To see if the error statement is reachable, the path constraint \( \text{input}_1 > \text{input}_2 \land \text{input}_1 \cdot \text{input}_2 = 1452 \land \text{input}_1/2 = 5 \)
Background

Figure 2.3. A simple sequential program and its symbolic execution tree

needs to be solved. At a first look, the path constraint seems unsatisfiable. After all, \( \text{input}_1 \div 2 = 5 \) indicates that \( \text{input}_1 = 10 \). Therefore \( \text{input}_2 \) needs to be smaller than 10 and multiplying the values together cannot result in the value 1452. However, if we consider the values to be signed 32-bit integers represented with bit vectors, the path constraint is satisfiable with values \( \text{input}_1 = 10 \) and \( \text{input}_2 = -858993314 \) as the multiplication of them underflows to 1452. Therefore with signed 32-bit integers the error in the program is reachable.

As DSE executes a program both concretely and symbolically, it can be seen as a generalization of static symbolic execution where a program is executed only symbolically. The advantage of the concrete execution is that it provides additional and accurate information about the program under test. In DSE the execution path followed by the concrete execution is trivially known to be satisfiable. In static symbolic execution, on the other hand, path constraints need to be solved for all paths. Additionally, static symbolic execution is helpless if the program calls, for example, system libraries whose source code is not available. In these cases symbolic execution cannot determine what symbolic values such calls would return. As discussed in [GKS05], DSE can in these cases use the concrete values to underapproximate the symbolic values. The concrete execution therefore gives DSE additional information over static symbolic execution that allows it to make progress in some cases where static approaches cannot. Naturally if symbolic values are replaced with concrete ones, some execution paths are likely to be left unexplored.
2.3 Testing Multithreaded Programs with DSE

DSE can be extended to handle multithreaded programs by using a runtime scheduler that controls the execution of threads [Sen06]. The program is executed such that there is always only one thread executing at a time and a new scheduling decision (i.e., selecting an enabled thread) is made after the currently scheduled thread has performed an execution step. The order in which the threads are allowed to execute can then be considered as an additional input to the program. By creating a branch in the symbolic execution tree for each scheduling decision, it is possible to explore systematically all interleavings of execution steps.

Example 3. To illustrate the extension of DSE to multithreaded programs, let us consider the program in Fig. 2.4. First the initial execution steps of threads 1 and 2 are performed. The order of these execution steps does not matter as the threads cannot affect each other before they perform any global operations. Thread 1 executes a conditional statement depending on an input value at line 3 before performing a global operation. This creates a branch in the symbolic execution tree with a symbolic constraint $\text{input}_1 + 1 \neq 5$ and its negation. Let us assume that the initial execution step follows the false branch at line 3. The initial execution step of thread 2 does not execute any conditional statements and therefore always corresponds to executing line 7. After the initial execution steps, both threads want to perform a read operation on the shared variable $X$.

The runtime scheduler selects which thread is allowed to proceed and a branch in the symbolic execution tree is created for both scheduling possibilities. Let us assume that thread 2 is allowed to proceed first. It will then execute the read operation and continue its local computation. The next statement to be executed after the read is a conditional statement and this results in a new branch in the symbolic execution tree. Assuming that the thread follows the false branch, the thread gets terminated and therefore finishes the current execution step. After that thread 1 is the only one available for scheduling and it is allowed to execute its read statement.

Fig. 2.4 shows the full symbolic execution tree of the example program. To better differentiate which branches are due to local conditional operations and which are the scheduling decisions, the parent nodes for the latter cases are shown with a darker color. The thread and the type of the global operation is also shown for each scheduling possibility. There
Global vars: Thread 1: Thread 2:

\[ X = 0; \]

1: \( i_1 = \text{input}() \); 7: \( i_2 = \text{input}() \);

2: \( i_1 = i_1 + 1; \) 8: \( b = X; \)

3: if \( (i_1 == 5) \) 9: if \( (i_2 > 10) \)

4: \( X = 5; \) 10: \( b = 0; \)

5: else

6: \( a = X; \)

Figure 2.4. A simple multithreaded program and its symbolic execution tree

are eight paths in the symbolic execution tree and it takes as many test executions to cover them all.

2.4 Using Partial Order Reductions with DSE

Even when ignoring the interleavings of local operations, the number of remaining interleavings is often too large to explore exhaustively. Fortunately properties such as assertion violations and deadlocks can be checked without necessarily always exploring all interleavings. To be able to describe which interleavings can be safely left unexplored, we use the following definitions. In these definitions we use similar terminology as used in [Sen06, SA06b].

**Definition 1.** An execution is a sequence of operations that can be performed starting from the initial state of a program \( P \). \( EX(P) \) is the set of all possible executions of \( P \) on all possible input values and all possible interleavings of threads.

**Definition 2.** Let \( ex = o_1, o_2, ..., o_n \) be an execution in \( EX(P) \). The operations \( o_i \) and \( o_j \) in \( ex \) are sequentially related, denoted by \( o_i \prec_{ex} o_j \), if and only if \( i \leq j \) and the operations \( o_i \) and \( o_j \) belong to the same thread.
Definition 3. Let \( ex = o_1, o_2, ..., o_n \) be an execution in \( EX(P) \). The operations \( o_i \) and \( o_j \) are shared memory access precedence related, denoted by \( o_i <_{ex}^{mem} o_j \), if and only if \( i < j \) and either (i) both \( o_i \) and \( o_j \) access the same shared variable and at least one of the operations is a write, or (ii) both operations access the same lock.

Definition 4. A happens before relation \( \rightarrow_{ex} \) is the transitive closure of the union of the \( <_{ex}^{seq} \) and \( <_{ex}^{mem} \) relations. If \( o_1 \rightarrow_{ex} o_2 \), it is said that \( o_1 \) happens before \( o_2 \) in an execution \( ex \).

The happens before relation is a partial order relation and it can be used to partition executions into equivalence classes. These equivalence classes are often called Mazurkiewicz traces [Die95]. For example, given executions \( ex_1 = (o_1, ..., o_n, o_{n+1}, ..., o_m) \) and \( ex_2 = (o_1, ..., o_{n+1}, o_n, ..., o_m) \) such that \( o_n \) and \( o_{n+1} \) are operations that belong to different threads and read the same shared variable, the execution \( ex_1 \) and \( ex_2 \) belong to the same Mazurkiewicz trace as both executions are linearizations of the same \( \rightarrow_{ex_1} \) partial order.

We say that two operations that can be concurrently enabled are dependent if executing them in different orders lead to different Mazurkiewicz traces and independent otherwise. In the context of our simple programming language, two operations that access the same shared variable are dependent if at least one of them is a write. Lock operations are dependent if they access the same lock. It is easy to see that given an execution, it is possible to obtain all other executions belonging to the same Mazurkiewicz trace (i.e., linearizations of the same happens before relation) by performing a number of swaps between adjacent independent operations.

It has been shown that restricting testing to Mazurkiewicz traces preserves all deadlocks and reachable local states [FG05]. For example, in Figure 2.4 there is a scheduling point (node 2) where two reads are enabled. Exploring both of the subtrees after the scheduling point is not necessary if we want to preserve the reachability of these properties.

When the execution order of two operations lead to different Mazurkiewicz traces, it can be seen that the operations are in race. Intuitively speaking two operations in an execution are in race if they are co-enabled in some equivalent execution and the operations are dependent.

Definition 5. Let \( ex = o_1, o_2, ..., o_n \) be an execution in \( EX(P) \). Operations \( o_i \) and \( o_j \) in \( ex \) are in race if and only if the operations belong to different
threads, \( o_i \xrightarrow{\text{ex}} o_j \) and in addition either of the following holds:

- \( o_i \) is a read or write event and there does not exist an operation \( o_k \) such that \( i < k < j \) and \( o_i \xrightarrow{\text{ex}} o_k \xrightarrow{\text{ex}} o_j \), or

- \( o_i \) and \( o_j \) are lock operations, \( o_u \) is the corresponding unlock operation of \( o_i \) and there does not exist an operation \( o_k \) such that \( u < k < j \) and \( o_u \xrightarrow{\text{ex}} o_k \xrightarrow{\text{ex}} o_j \).

The definition above describes a general race relation between operations. As a special case we call races between a write and either a read or write operations as **data races**.

In the following we provide an overview of partial order reduction based approaches that have been combined with DSE. The main purpose of this is to get a basic understanding of these methods so that the advantages and disadvantages of the new testing approaches presented in this thesis can be discussed. We will start by describing dynamic partial order reduction [FG05] and race detection and flipping [SA06b] algorithms that are the two most closely related approaches to the algorithms discussed in this thesis. We will then describe sleep sets that can be used in conjunction with the other two approaches.

**Dynamic Partial Order Reduction**

The dynamic partial order reduction (DPOR) algorithm aims to cover all Mazurkiewicz traces of a given program. The general idea behind DPOR is as follows. Initially any test execution (e.g., a random one) is performed. This execution is then analyzed in order to determine if there are operations in the execution that are in race. If DPOR detects such a race, it computes a point in the execution where a different scheduling decision must be made so that the operations in race could be executed in a different order. In the following we call such points as backtrack points.

The algorithm then covers the backtrack points by performing new test executions. These new executions are also analyzed so that additional backtrack points can be obtained. The algorithm in Fig. 2.5 shows a high level description of a DPOR algorithm. Note that the presented algorithm explicitly constructs an execution tree. This is not necessary if the backtrack points are processed in a depth first order like done in the original DPOR algorithm presented in [FG05]. The construction of the execution tree, however, allows the backtrack points to be explored in an arbitrary
**Input:** A program $P$, an execution length bound $k$

1: tree := execution tree with only a root node
2: unvisited := \{ root of tree \}
3: while unvisited \neq \emptyset do
4: remove $target \in$ unvisited
5: $ex := \text{EXECUTE}(P, target, k)$
6: let $o_1, o_2, \ldots, o_n = ex$
7: $N := \text{root of tree}$
8: for $i = 1$ to $n$ do
9: node := CORRESPONDINGNODE($o_i, N$)
10: if node \notin tree then
11: add node to tree as a child node of $N$
12: $s := \text{the state reached in ex after } o_i$
13: for all threads $t$ do
14: $o_t := \text{the operation } t \text{ wants to perform in } s$
15: if $\exists j = \max(\{ j \mid j < i \text{ and } \text{race}(o_j, o_t)\})$ then
16: unvisited := unvisited \cup \text{BACKTRACK}(ex, j, t)$
17: $N := node$

**Figure 2.5.** Dynamic partial order reduction algorithm

order. Furthermore, by explicitly constructing an execution tree, we can describe the algorithm in Fig. 2.5 using a similar structure that we will use for the new unfolding based algorithms described later in this thesis.

The algorithm in Fig. 2.5 maintains a set of nodes of the execution tree that have not been covered by test executions (line 2). These nodes correspond to the backtrack points computed by DPOR. Initially this set contains the root node that can be seen as corresponding to the initial state of the program under test. The algorithm then performs test executions until the set of unvisited nodes becomes empty. For each test execution, the algorithm selects and removes a node from the list of unvisited nodes (line 4). The \text{EXECUTE} subroutine then performs a test execution where the threads perform execution steps in the same order as the global operations occur in the path from the root node to the target node in the execution tree. Such execution therefore covers the target node. After reaching the node, any thread schedule can be followed for the remainder of the execution. This means that to cover the root node, the initial execution can be any random execution. As a result the \text{EXECUTE} subroutine returns a sequence of operations that is then analyzed in order
to find backtrack points. The parameter $k$ given to the \textsc{execute} subroutine denotes a bound on the execution length (e.g., the maximum number of operations in an execution). Any execution whose length exceeds the given bound is terminated in which case \textsc{execute} returns only a prefix of a complete execution. This guarantees that the algorithm terminates also in cases where there are infinite execution paths. Naturally this means that only an underapproximation of the full program will be tested if there are executions whose length exceeds the bound $k$. In such cases the algorithm is not guaranteed to detect all errors.

The algorithm analyses the sequence of operations returned by \textsc{execute} one by one (line 8). For each operation the \textsc{correspondingnode} subroutine is used to get the node in the execution tree that is reached after performing the operation. If such node does not already exist in the tree, the subroutine creates a new node. For any new node, the algorithm computes backtrack points. This is done by considering the state $s$ reached by the operation sequence processed so far and the operation $o_t$ (which can be enabled or blocked in $s$) that a thread $t$ wants to perform in $s$, for all threads $t$. For each such operation it is then checked if there exists another operation in the current execution that is in race with $o_t$. If there are multiple operations in the current execution that are in race with $o_t$, then DPOR only considers the most recently executed of them (line 15). That is, if operations $o_5$ and $o_7$ are in race with $o_t$, then a backtrack point is computed only for the race between $o_7$ and $o_t$ (i.e., $\text{max}(5, 7) = 7$ on line 15). The other race is processed recursively when an execution is analyzed where the operation $o_t$ is performed before $o_7$.

To be able to compute when operations can be in race, DPOR needs to track the happens before relation between operations. Typically vector clocks \cite{Mat89} are used in DPOR algorithms for this purpose. The original DPOR paper also simplifies the computation of when two operations can be in race by considering any operations (even two reads) to be dependent if they access the same shared memory location and by assuming for reads and writes that they can always be co-enabled. In \cite{SKH12} a version of DPOR is described that uses vector clocks to avoid exploring interleavings resulting from reads of the same shared variable. The implementation of DPOR used in this thesis corresponds to the algorithm described in \cite{SKH12}.

If the algorithm detects that there can be a race between two operations, a new test execution needs to be performed where the operations
are executed in a different order. For this reason the BACKTRACK subroutine adds new nodes to the set of unvisited nodes. For adding backtrack points, the original DPOR paper describes the following heuristic. If the thread \( t \) is enabled in the state reached in \( ex \) after performing \( o_{j-1} \), a test execution needs to be performed that follows the prefix of \( ex \) up to \( o_{j-1} \) and after that performs an operation belonging to the thread \( t \). If \( t \) is not enabled after \( o_{j-1} \), then a backtrack point is created for all threads that are enabled after \( o_{j-1} \). In our algorithm this means if a node \( n \) corresponds to the operation \( o_{j-1} \) in the execution tree, then BACKTRACK adds a node for thread \( t \) as a child node of \( n \) in the case \( t \) is enabled after \( o_{j-1} \) and otherwise nodes for all enabled threads as children of \( n \). Note that if there already exists a corresponding child node in the tree, then there is no need to create a new node. In the implementation of the DPOR used in the experiments in this thesis, we use this same heuristic for computing backtrack points.

**Example 4.** To illustrate how DPOR works, let us consider the program shown in Fig. 2.6. To cover the root node of the execution tree, DPOR can perform any execution. Let us assume that the first test execution corresponds to performing an operation sequence \((t_1: \text{read}(X), t_2: \text{read}(X), t_1: \text{write}(Y), t_2: \text{write}(Y))\). This operation sequence is then processed one operation at a time. As there is no such node in the execution tree that corresponds to the operation \( t_1: \text{read}(X) \), a new node is created and added to the tree. This is shown as the node 2 in Fig. 2.6. After adding the node, DPOR considers the operations \( t_1: \text{write}(Y) \) and \( t_2: \text{read}(X) \) that the threads want to perform after the first operation. As neither of these can be in race with the operations processed so far (i.e., the operation \( t_1: \text{read}(X) \)), no backtrack nodes are added. The operation \( t_2: \text{read}(X) \) is processed in the same way. As a result the node 3 is added to the tree but again no backtrack nodes are found. After adding the node 4 for operation \( t_1: \text{write}(Y) \), the algorithm notices that the thread \( t_2 \) wants to perform the operation \( t_2: \text{write}(Y) \) next. As the both writes are dependent and can be co-enabled, the algorithm adds a backtrack node to the execution tree. This is shown as the node drawn with dashed lines. After adding the backtrack node, the algorithm processes the last operation and adds node 5 to the execution tree. The execution tree after the first test execution is shown on the left hand side in Fig. 2.6.

As there is one unvisited node in the tree (i.e., the one marked with dashed lines) after the first execution, the algorithm selects this node as
Global variables:  
Thread 1:  
\[ X = \emptyset; \]  
1: \( a = X; \)  
3: \( b = X; \)  

Thread 2:  
\[ Y = \emptyset; \]  
2: \( Y = 1; \)  
4: \( Y = 2; \)  

Figure 2.6. An example illustrating the DPOR algorithm  

The test target. To cover it, a test execution needs to be performed that has an operation sequence \((t_1: \text{read}(X), t_2: \text{read}(X), t_2: \text{write}(Y))\) as its prefix. After performing and processing the corresponding execution, the final execution tree shown on the right in Fig. 2.6 is obtained. Note that in this execution the operations that write to \(Y\) are also in race. However, the backtrack point for this corresponds to the node 4 and therefore no new backtrack nodes are created.

The presented DPOR algorithm is guaranteed to cover all Mazurkiewicz traces as proven in [FG05]. However, in some cases it can cover some Mazurkiewicz traces more than once. Therefore the algorithm may perform unnecessary test executions. Furthermore, the number of unnecessary test executions can vary depending on the order in which the algorithm explores different interleavings of threads. This is illustrated by the following example.

**Example 5.** Let us again consider the program in Fig. 2.6. This time, however, assume that in the first test execution both operations of thread 1 are performed first and the operations of thread 2 after them. Such execution corresponds to the nodes from 1 to 5 in Fig. 2.7. After adding the node 4 to the execution tree, the algorithm notices that the write operations of both threads are in race. Therefore the algorithm adds a backtrack node before the write of thread 1 to the execution tree. This corresponds to the node 6. To cover the backtrack node, the algorithm can perform, for example, a test execution corresponding to the nodes 1, 2, 6, 7 and 8 in Fig. 2.7. In this execution the write operations are again in race. This causes the algorithm to add the backtrack node 9 and to perform a third test execution. Therefore the DPOR algorithm can perform either
Figure 2.7. The effect of execution order in the number of explored interleavings

two or three test executions on this particular program.

One approach to avoid exploring unnecessary interleavings is to use a heuristic to select in which order they are explored. Several different heuristics were experimented in [LKMA10]. However, based on the results there is no single heuristic that works well for all kinds of programs. Another way to address the problem is to combine DPOR with sleep sets. We will discuss this option later in this section. It has also been recently shown in [AAJS14] that DPOR can be made optimal in the sense that it explores each Mazurkiewicz trace only once by using additional constructs called wakeup trees.

Race Detection and Flipping

Sen and Agha propose an approach called race detection and flipping that is very similar to DPOR. That is, the race detection and flipping algorithm also analyses a sequence of operations observed during a test execution and checks if some operations in it can be in race. The algorithm also uses vector clocks to track the causality between operations in test executions in order to detect races. The main difference to DPOR is that instead of adding backtrack nodes directly when races are detected, the race detection and flipping algorithm uses race flags and postponed sets to handle the backtracking. To be more precise, let $o_1, o_2, \ldots, o_j, \ldots, o_t, \ldots$ be an execution where the algorithm notices that $o_j$ and $o_t$ are in race. If the operation $o_j$ is performed from a node $n$ in the execution tree, then race detection and flipping sets a race flag true for node $n$ and adds the thread performing $o_j$ to a set of postponed threads in node $n$. After the execution, the algorithm notices that there is a race at node $n$ and it performs another execution that has the same prefix up to the node $n$ and after reaching $n$ the execution of threads in the postponed set of $n$ are delayed
as much as possible. This has the effect that the execution order of \( o_j \) and \( o_t \) gets flipped in the new execution, hence the name race detection and flipping.

The race detection and flipping algorithm is guaranteed to cover all Mazurkiewicz traces. However, like DPOR, it can cover some Mazurkiewicz traces multiple times if in the executions there are independent operations between the operations that are in race. Some but not all of these unnecessary test executions can be eliminated if the algorithm is combined with sleep sets.

**Sleep Sets**

One way to avoid some of the redundant test executions made by DPOR and race detection and flipping algorithms is to combine them with sleep sets [God96]. In the context of the algorithms described in this chapter, sleep sets are associated with nodes in the execution tree. A sleep set is a set of operations with the following intuitive meaning. If an execution reaches a state (i.e., a node in the execution tree) that has a non-empty sleep set, then the execution does not need to perform an operation \( o \) in the sleep set until an operation \( o' \) that is dependent with \( o \) has been performed.

For the root of the execution tree the sleep set is an empty set. For any other node \( n \) a sleep set is computed when it is added to the execution tree (either as a node that has been covered or as a backtrack node). This is done as follows. Let \( p \) be the parent node of \( n \). First add all operations in the sleep set of \( p \) to the sleep set of \( n \). After this add all operations corresponding to the child nodes of \( p \) (including non-visited backtrack nodes) to the sleep set of \( n \). Finally remove all operations from the sleep set of \( n \) that are dependent with the operation that was performed to move the execution from \( p \) to \( n \).

**Example 6.** As an example, let us consider the execution tree in Fig. 2.7 and the test executions described in Example 5. For the root node 1 the sleep set is empty. For the node 2 the sleep is also empty as the sleep set of the parent is empty and there are no other operations from node 1. In a similar way the nodes 3, 4 and 5 also have empty sleep sets. (Note that when the sleep set for node 3 is computed during the first test execution, the node 6 does not yet exists in the execution tree.) For node 6 the sleep set is the empty set from the parent node augmented with write(Y) opera-
tion of thread 1 minus any operations that are dependent with the read(X)
operation of thread 2. As the write of thread 1 is not dependent with the
read of thread 2, the sleep set of node 6 therefore is \( \{ t_1 : \text{write}(Y) \} \). This
means that any test execution reaching the node 6 would not perform the
operation \( t_1 : \text{write}(Y) \) before an operation that is dependent with it is
performed. Therefore the execution path corresponding to nodes 6,7 and
8 would never be explored. After node 6 the test execution would there-
fore cover the node 9. For this node the sleep set contains the \( t_1 : \text{write}(Y) \)
operation from the parent node but as the operation \( t_2 : \text{write}(Y) \) is de-
pendent with it, it gets removed. Therefore the sleep set for node 9 is an
empty set and the write of thread 1 can be performed after it.

With sleeps sets the set of test executions that terminate normally (i.e.,
do not end up in a state where all enabled threads are in the sleep set)
contain one execution for each Mazurkiewicz trace [GHP95]. In other
words, sleep sets guarantee that no two complete test executions covering
the same Mazurkiewicz trace are performed. However, a large number of
test execution may end up in a state where all threads belong to a sleep
set. Therefore sleeps sets alone are not enough to eliminate all redundant
test executions when the aim is to cover all Mazurkiewicz traces. As illus-
trated by the example above, when sleep sets are combined with DPOR,
some redundant test executions can be avoided. However, there can still
be cases where DPOR, and also race detection and flipping, perform un-
necessary test executions where all threads end up in a sleep set. There-
fore larger modifications to the algorithms, like described in [AAJS14],
are needed to make the algorithms optimal.

As a final note, one needs to be careful when combining sleep sets with
DPOR algorithms that use heuristics. For example, if DPOR finds that
two operations \( o_j \) and \( o_t \) are in race, such that thread \( t \) performs \( o_t \) and
\( n \) is the node from which \( o_j \) is performed, the DPOR version described in
this chapter uses the heuristic that adds a backtrack node for thread \( t \) as
a child node of \( n \). However, it is possible that the operation of thread \( t \)
is in the sleep set of node \( n \). This means that according to the sleep sets
it is not necessary to cover the backtrack node. This can have the effect
that the testing algorithm misses some Mazurkiewicz traces. A simple
way to avoid this is to ignore the sleep sets for backtrack nodes as done in [SKH12]. This can limit the usefulness of the sleep sets considerably.
Another way is to add backtrack nodes for all enabled threads in \( n \) that
are not in the sleep set of \( n \). This corresponds to the approach that the
original DPOR algorithm in [FG05] uses for cases where the heuristic fails to find a backtrack node. In our implementation of DPOR we use this second approach.
3. **Unfolding Based Testing**

The algorithms discussed in the previous chapter reduce the number of needed test executions by ignoring irrelevant thread interleavings. In this chapter we describe a different kind of DSE based testing algorithm that constructs an acyclic representation of all possible interleavings that is often more compact than the full symbolic execution tree of the program. The reduction obtained by this method is based on the observation that such a succinct representation can be covered with less test executions than covering a full symbolic execution tree. In this chapter we concentrate on the problem of determining whether an *error* statement (see Table. 2.1) is reachable in a multithreaded program by exploring all (symbolic) local states of threads. The approach presented in this chapter can also be extended to detect deadlocks. We consider this problem separately in Chapter 6.

### 3.1 Motivation

All Mazurkiewicz traces of a program can be explored efficiently with an algorithm such as DPOR. However, if the aim is to detect violations of assertions on local states, it is enough to cover the reachable local states of threads. This does not always require all Mazurkiewicz traces to be covered. This is true, for example, for programs that contain parts where the executing threads are highly independent of each other. Let us consider a simple program with \( n \) pairs of threads and \( n \) shared variables such that the first thread in each pair reads a shared variable and the second thread writes to it. Furthermore, each pair operates on a different shared variable. In such a program the number of Mazurkiewicz traces grows exponentially as the number of threads increases. Figure 3.1 shows an instance of the program with \( n = 2 \) and one possible execution tree repre-
Global variables:  Thread 1:  Thread 2:
X = 0;  1: a = X;  2: X = 1;
Y = 0;

Thread 3:  Thread 4:
3: b = Y;  4: Y = 2;

Figure 3.1. Exponentially growing example

senting the four Mazurkiewicz traces of the program.

To explore the local reachable states in our example, only two test executions are needed regardless of the number of threads: one execution where each thread pair executes the read operation first and a second one where the writes are executed first. In this chapter we describe a testing approach based on net unfoldings that is guaranteed to cover all reachable local states of threads but is not required to explicitly cover all Mazurkiewicz traces.

3.2 Petri nets and Unfoldings

In the following we introduce Petri nets and their unfoldings. An unfolding of a Petri net is another net but with an acyclic structure. The concept of unfoldings was introduced in [NPW81] and McMillan was the first to propose the use of so called complete finite prefixes of unfoldings in the context of verification in [McM92, McM95]. The approach introduced by McMillan can be seen as a method that aims to alleviate the state explosion problem encountered in verification. In his work, McMillan used unfoldings for deadlock checking. After that the use of unfoldings has been well studied and the approach has been improved and used for many other verification problems as well (see, e.g., [CGP00, EH00, EH01, KK01, ERV02, KKV03]). In this thesis we use unfoldings to represent the executions of a multithreaded program in a succinct way. This allows us to reduce the number of redundant test executions.
Definition 6. A net is a triple \((P, T, F)\), where \(P\) and \(T\) are disjoint sets of places and transitions, respectively, and \(F \subseteq (P \times T) \cup (T \times P)\) is a flow relation. Places and transitions are called nodes and elements of \(F\) are called arcs. The preset of a node \(x\), denoted by \(\cdot x\), is the set \(\{y \in P \cup T \mid (y, x) \in F\}\). The postset of a node \(x\), denoted by \(x\cdot\), is the set \(\{y \in P \cup T \mid (x, y) \in F\}\). A marking of a net is a mapping \(P \mapsto \mathbb{N}\). A marking \(M\) is identified with the multiset which contains \(M(p)\) copies of \(p\). A Petri net is a tuple \(\Sigma = (P, T, F, M_0)\), where \((P, T, F)\) is a net and \(M_0\) is an initial marking of \((P, T, F)\).

Graphically markings are represented by putting tokens on circles that represent the places of a net. A net is \(n\)-bounded if in any reachable marking a place contains at most \(n\) tokens. A net is called safe if it is 1-bounded. All the nets considered in this works are safe. A transition \(t\) is enabled in a marking that puts tokens on the places in the preset of \(t\). Firing an enabled transition results in a new marking that is obtained by removing a token from every place in the preset of the transition and adding a token to the places in the postset of the transition.

Definition 7. Causality, conflict and concurrency between nodes \(x\) and \(y\) are defined as follows:

- \(x\) and \(y\) are in causal relation, denoted as \(x < y\), if there is a non-empty path of arcs from \(x\) to \(y\).

- \(x\) is in conflict with \(y\), denoted as \(x \# y\), if there is a place \(z\) that is different from \(x\) and \(y\) such that \(z < x, z < y\) and the paths from \(z\) to \(x\) and \(y\) take different arcs out of \(z\).

- \(x\) and \(y\) are concurrent, denoted by \(x \text{ co } y\), if \(x\) and \(y\) are neither causally related nor in conflict.

It is possible to unwind a directed graph into a tree that represents all the paths through the graph. Similarly, a Petri net can be unfolded into an acyclic net called an occurrence net.

Definition 8. An occurrence net \(O\) is an acyclic net \((B, E, G)\), where \(B\) and \(E\) are sets of conditions (places) and events (transitions) and \(G\) is the flow relation. Occurrence net \(O\) also satisfies the following conditions: for every \(b\) in \(B\), \(|\cdot b| \leq 1\); for every \(x \in B \cup E\) there is a finite number of nodes \(y \in B \cup E\) such that \(y < x\); and no node is in conflict with itself.
For occurrence nets the causality relation is a partial order and we say that \(x\) causally precedes \(y\) if \(x < y\). Furthermore, for occurrence nets it holds that any two nodes \(x\) and \(y\), such that \(x \neq y\), are either causally related, in conflict or concurrent. A set of nodes where all nodes are pairwise concurrent is called a co-set. To avoid confusion when talking about Petri nets and their occurrence nets, the nodes \(B\) and \(E\) are called conditions and events, respectively. If an occurrence net is obtained by unfolding a Petri net, the events and conditions in it can also be labeled with the corresponding transitions and places.

**Definition 9** (Adapted from [KK01]). A labeled occurrence net is a tuple \((O, l) = (B, E, G, l)\) where \(l : B \cup E \rightarrow P \cup T\) is a labeling function such that:

- \(l(B) \in P\) and \(l(E) \in T\);  
- for all \(e \in E\), the restriction of \(l\) to \(\cdot e\) is a bijection between \(\cdot e\) and \(\cdot l(e)\), and similarly for \(e^*\) and \(l(e)^*\);  
- the restriction of \(l\) to \(\text{Min}(O)\) is a bijection between \(\text{Min}(O)\) and \(M_0\), where \(\text{Min}(O)\) denotes the set of minimal elements with respect to the causal relation;  
- for all \(e, f \in E\), if \(\cdot e = \cdot f\) and \(l(e) = l(f)\) then \(e = f\).

Labeled occurrence nets are also often called branching processes in the literature. Different labeled occurrence nets can be obtained by stopping the unfolding process at different times. The maximal labeled occurrence net (possibly infinite) is called the unfolding of a Petri net [EH08]. To simplify the discussion in this thesis, we use the term unfolding for all labeled occurrence nets and not just the maximal one.

**Example 7.** Figure 3.2 shows a Petri net on the left side, a tree representing all possible computations in the middle and the unfolding on the right side. Notice that the conditions and events in the unfolding are labeled with the corresponding places and transitions in the Petri net. For any computation (i.e., a sequence of transitions) in the Petri net, there is a matching computation in the unfolding. If a computation visits the same node multiple times, there is always a new copy of the node in the unfolding. For example, in the figure there are two conditions for \(s_2\).

Let us also consider some nodes in the unfolding. The condition in the
Unfolding a Petri net is typically done using an algorithm similar to the one shown in Fig. 3.3. The algorithm maintains a set of possible extensions that are events that have not yet been added to the unfolding being constructed but can be fired in some reachable marking of it. In other words, the subroutine \text{POSSIBLEEXTENSIONS} returns all events \( e \) for which it holds that (i) there exists a transition \( t \) in \( \Sigma \) such that \( l(\cdot\cdot e) = \cdot\cdot t \), and (ii) \( e \) does not already exist in \( Unf \). Initially the set of possible extensions contains events that correspond to transitions that can be fired in the initial marking of the Petri net. The unfolding algorithm then adds events from the possible extensions one by one to the unfolding and after each added event, updates the set of possible extensions. If the unfolding is finite, the algorithm terminates when no more events can be added to the unfolding.

An unfolding can sometimes be exponentially more succinct than a computation tree representation. We will see later in this chapter that we
can use this property in automated testing to reduce the number of test executions. The succinctness, however, comes with a price. For a computation tree it is trivial to determine how a node should be extended (i.e., what its child nodes should be). With unfoldings, computing possible extensions is no longer trivial. In fact computing possible extensions is typically computationally the most expensive part of the unfolding process of Petri nets and has been shown to be NP-hard in the size of the unfolding [EH08]. There exists possible extensions algorithms that make different space-time tradeoffs. For example, a memory intensive approach presented in [ER99] uses a table to maintain information whether two given conditions are concurrent or not. This table can then be used to find presets of possible extension events efficiently. However, as the unfolding grows, the size of the table grows quickly and updating it becomes expensive. A memory light approach is to compute the co-relation between conditions directly from the unfolding.

A typical Petri net unfolding algorithm computes finite prefixes of unfoldings by using cut-offs [EH08]. This is similar as not extending the computation tree if the same node (i.e., the same program state) is encountered multiple times. A complete finite prefix contains the same amount of information as the Petri net being unfolded. For now we consider unfoldings without cut-offs but we will return to the topic in Chapter 7.

### 3.3 Modeling Multithreaded Programs with Petri Nets

As discussed in the previous chapter, DSE can be seen as an approach that unwinds a control flow graph into a symbolic execution tree. As the control flow graph is an abstraction without data (i.e., without the values of the variables in the program), it overapproximates possible control flow paths. Path constraints can then be used to determine which control flow paths in the symbolic execution tree are feasible and which ones are not. The same basic idea can also be applied to Petri nets and unfoldings. That is, we can assume that we have a Petri net that models a multithreaded program but abstracts away the exact data values and the data manipulation operations on them. Such a model can then be unfolded into an acyclic Petri net and path constrains can be used to eliminate infeasible paths from the unfolding.

To model programs with Petri nets, we model statements in the program with the constructs shown in Fig 3.4. These constructs are similar to the
ones described by Farzan and Madhusudan in [FM06], where they are used to model programs for the purpose of atomicity checking. Figure 3.5 shows a simple program and its Petri net model using these constructs.

To model programs with the constructs we assume that there is for each thread one place for each program location the thread can be in. In Fig 3.4 the label \( pc \) denotes one such place and \( pc' \) the place reached after executing the modeled statement. We also assume that there is one place for each lock in the program (labeled as \( L \) in the figure) and \( n \) places for each shared variable (labeled as \( V_i \) in the figure) where \( n \) is the number of threads in the program under test. The \( n \) places can intuitively be thought of as local copies of the shared variable for each thread. When a thread reads a shared variable, it accesses only its local copy. A write, however, accesses all the copies as such operation can change the value of the variable. This is a standard so called place replication encoding often used in connection with modeling concurrent programs with Petri nets (e.g., see [FM06]).

At a first look the modeling constructs for reads and writes can seem strange. After all, the corresponding transitions take a token from a place denoting a shared variable and then immediate put the token back. This means that the places for shared variables are always marked and do not affect how the model can be executed. The key insight here is that the places for the shared variables are used to denote when the executions of statements belonging to different threads are dependent. To see this, let us consider Fig. 3.6 that shows the unfolding of the model in Fig. 3.5. Note that for the sake of clarity we draw arcs between events and conditions representing shared variables with dashed lines. The semantics of
Unfolding Based Testing

Thread 1:
\[
\begin{align*}
\text{a} &= X; \\
\text{a} &= a + 5; \\
\text{if} \ (a &> 10) \\
\quad &\text{// no-op}
\end{align*}
\]

Thread 2:
\[
\begin{align*}
\text{b} &= \text{input}(); \\
\text{if} \ (b &== 0) \\
X &= b;
\end{align*}
\]

Figure 3.5. A multithreaded program and its Petri net model

such arcs are identical with normal arcs. That is, the difference is purely visual.

The events in the unfolding correspond to operations the program can perform. As the read and write transitions in the model are dependent through a shared condition, firing them in different orders leads to different events in the unfolding. In the example unfolding the write event 13 can only be fired after the read event 1 and vice versa for events 5 and 12. This captures the shared memory access precedence relation between global operations. In fact, it is easy to see that the causality relation between the events in the unfolding matches with the happens-before relation defined in Chapter 2. We will formally prove this later in this chapter. This means that if we want to cover, for example, the event labeled as 7 with a test execution, we know that it can be done by performing a sequence of operations that matches with the events that causally precede the event 7 (i.e., the events 9, 11, 12, 5, 6 and 7). Such a sequence of operations has a specific path constraint that can be used to determine whether the operation sequence can be followed in the concrete program. In this case the path constraint would be the conjunction of \( \text{input}_1 = 0 \) (from the operation corresponding to event 11) and \( \text{input}_1 > 10 \) (from the operation corresponding to event 7). As the path constraint is unsatisfiable, the event 7 cannot be reached when the data values are taken into account and therefore could be removed.

It is also possible to model programs in such a way that only one place per shared variable is used. This, however, means that the model considers two reads of the same shared variable to be always dependent. This
is illustrated in Fig. 3.7 that shows such a model and its unfolding for a program consisting of two concurrent reads. As we will see later in this chapter, covering such an unfolding requires two test executions that explore both interleavings of the reads. By replicating the places for shared variables, the read transitions become independent as shown in Fig 3.8 and the resulting unfolding can be covered with a single test execution.
Modeling Test Executions with Unfoldings

An unfolding of a program can also be constructed without an explicit Petri net model in the same way as standard DSE can construct a symbolic execution tree without a control flow graph. All the information needed for the construction can be obtained by performing test executions. Furthermore, as discussed in the previous chapter, we are only interested in exploring sequences of execution steps and therefore we do not explicitly need to create events for local statements (with the exception of conditional statements depending on input values). For example, in Fig. 3.6 we could merge event 2 with event 1. In the following we show how an unfolding can be constructed from a set of test executions that cover the whole program. In the next section we will then present a testing algorithm that can compute such a set on-the-fly based on the information stored in the unfolding.

To model a set of test executions as an unfolding, a set of conditions corresponding to the initial state of the Petri net model is first created. During the modeling process a marking $M$ is maintained such that at the beginning of each test execution, $M$ is set to be the initial marking. A test execution is then performed both concretely and symbolically after which the operations of this execution are processed one at a time and the unfolded versions of the constructs in Fig. 3.4 are added to the unfolding. The constraints generated by symbolic execution are stored to the corresponding branching events. The preset of each event that is added to the unfolding is obtained from the marking $M$ and after each processed event, $M$ is updated by firing it. Therefore $M$ always represents the current state of the test execution. If the unfolding already contains an event with the same preset (i.e., it has been added during an earlier test run), the marking $M$ is updated by firing this event but no new event is added. This guarantees that there are no duplicate events that represent the same operation.

To simplify the discussion in the rest of this thesis, we refer to conditions corresponding to program locations, locks and shared variables as thread conditions, lock conditions and shared variable conditions, respectively.

Example 8. Let us construct an unfolding of the program in Fig. 3.5 by modeling a set of test executions that are performed both concretely and symbolically. To start the modeling process, the initial state of the program is modeled as the topmost net in Fig. 3.9.
Let the first test execution be such that thread 2 first assigns an input value to the local variable $b$. Assume that the concrete input value being assigned is 0 and the symbolic input value is $\text{input}_1$. This operation is a local one so we do not need to model it with an event. After reading the input value, thread 2 encounters a conditional statement whose result depends on the input value. The concrete execution follows the true branch and the symbolic execution generates a symbolic constraint $\text{input}_1 = 0$. This branching operation is modeled as the event 1 in the unfolding shown in Fig. 3.9. The generated constraint is stored to this event.

Let us assume that the execution turn is next given to thread 1. The thread reads the initial value of the shared variable $X$ which we assume to be 0. This is modeled as event 2 in the unfolding and $M$ is again updated by firing the event. After this the thread performs a local assignment and a branch operation that does not depend on input values. Given that the
thread reads the initial (non-symbolic) value of $X$, these operations are always performed the same way and therefore do not need to be explicitly modeled in the unfolding. After this the only remaining operation left is the write by thread 2, which is modeled as event 3.

The last unfolding in Fig. 3.9 shows the complete unfolding after additional test executions have been modeled in the same way as above. Compared to the unfolding in Fig. 3.6 the new unfolding contains only those events that can be covered with test executions.

Note that an unfolding of a program is an acyclic structure and therefore all control flow paths of individual threads are unwound into a tree-like structure similarly as with standard DSE. The unfolding can therefore be seen as a structure that contains separate symbolic execution trees for each thread and uses lock and shared variable conditions to represent the happens-before relation. This can in some cases make the unfolding more compact compared to a symbolic execution tree. However, the unfolding needs to be kept in memory and therefore the approach has additional memory requirements when compared to depth first exploration based DSE.

**Example 9.** Fig. 3.10 shows an unfolding of the program in Fig. 3.1 such that read events are labeled with $R$ and write events with $W$. Note that all the events and conditions in the unfolding can be covered by two test executions: executing the reads before writes and vice versa. Any execution of the program can be simulated with the unfolding and therefore two test executions are enough to construct the full unfolding and to cover all reachable (symbolic) local states of threads. If additional pairs of threads were added to the program, it would still be possible to cover the program with only two test executions. If the executions were represented as an execution tree, adding new pairs of threads would add two new branches for each of the leaf nodes of the execution tree. Therefore the tree and the number of paths in it would grow exponentially. In cases like these, unfoldings can lead to an exponential reduction to the number of needed test executions.

### 3.4 Systematic Exploration of Execution Paths

Modeling a given set of test executions as an unfolding is simple. However, a systematic testing algorithm needs to generate a set of test executions
that cover the full unfolding of the program (and therefore the reachable local states). We will next present an algorithm shown in Fig. 3.11 that achieves this. We start by describing the algorithm at a high level and give further details of the subroutines used by the algorithm in the following subsections.

The algorithm follows the same basic structure as the simple unfolding algorithm presented in Section 3.2. It starts by constructing an initial unfolding in the same way as discussed in the previous section. The algorithm maintains a set of possible extensions that are events that have not yet been added to the unfolding but can be fired in some reachable marking of the unfolding. Initially the set of possible extensions contains those events that can be fired from the initial marking. These events can be easily obtained from a test execution where each thread is executed until it performs its first global operation. Such execution observes the operations that are enabled in the initial state. The algorithm then selects one possible extension that has not yet been added to the unfolding to be a target for a new test execution (lines 4-5). A test execution using both concrete and symbolic execution is then performed by the EXECUTE subroutine to cover the target event. To do this, EXECUTE computes input values and a schedule for executing threads that force a test execution to follow a path that leads to the target event. To compute input values, a path constraint is constructed and solved like in standard DSE. It is also possible that the target event is not reachable by any input values (i.e., the path constraint is unsatisfiable) in which case the target event is simply removed from the set of possible extensions. To guarantee termination of the test execution, a bound $k$ is used to limit the execution length of each thread. Based on the test execution, EXECUTE gives as output a sequence of executed global and symbolic branching operations. More details of EXECUTE and how it is based on DSE are given in Section 3.5.

The performed test execution is then modeled with events in the same
Input: A program $P$

1: $unf := \text{initial unfolding}$
2: $\text{extensions} := \text{INITIALLY ENABLE EVENTS}(P)$
3: while $\text{extensions} \neq \emptyset$ do
4: choose $\text{target} \in \text{extensions}$
5: if $\text{target} \notin unf$ then
6: operation_sequence := $\text{EXECUTE}(P, target, k)$
7: $M = \text{initial marking}$
8: let $o_1, o_2, \ldots, o_n = \text{operation_sequence}$
9: for $i = 1$ to $n$ do
10: $e = \text{CORRESPONDING EVENT}(o_i, M)$
11: if $e \notin unf$ then
12: add $e$ and its output conditions to $unf$
13: $\text{extensions} := \text{extensions} \setminus \{e\}$
14: $pe := \text{POSSIBLE EXTENSIONS}(e, unf)$
15: $\text{extensions} := \text{extensions} \cup pe$
16: $M = \text{FIRE EVENT}(e, M)$

Figure 3.11. Unfolding based testing algorithm

ways as explained in the previous section. In other words, a marking $M$ denoting the state of the test execution is created and initially set to match the initial marking (line 7). The algorithm then processes the operation sequence one operation at a time. For each operation it is checked whether the unfolding already contains a corresponding event that is enabled in the current marking $M$. If such event exists, the marking is updated by firing the existing event and the algorithm proceeds to process the next operation. If no such event exists, a new one is created and added to the unfolding. For each added event the algorithm also updates the set of possible extensions that act as future test targets. This is done by computing a set of new events that can become enabled in any reachable marking where the newly added event has been fired. A naive way to compute possible extensions would be to iterate through all reachable markings and check if it is known that a thread wants to perform an operation that has not been modeled yet when it reaches the marking in a test execution. An efficient way to compute possible extensions is described in detail in Chapter 4. After there are no more possible extensions left to add to the unfolding, the algorithm terminates.

Example 10. To illustrate the unfolding based testing algorithm, let us
consider the program in Fig. 3.12. The algorithm initially adds the events 1 and 2 shown in Fig. 3.12 to the set of possible extensions. This is because they correspond to the operations that are enabled in the initial state. Let us assume that the algorithm selects the event 1 as the first test target and performs an execution where the operations of thread 1 are executed first.

The operation sequence for this test consist of a read operation \( a = x \) by thread 1 and the operations \( b = x \) and \( x = \text{input()} \) by thread 2. Notice that executing the if-statements do not generate operations for the sequence as they do not depend on input values in this particular execution. The algorithm then determines if there exists an event corresponding to the first operation in the generated sequence. As no such event yet exists, a new one is added to the unfolding. Note that the event is in the set of possible extensions but it is not added to the unfolding until a test execution covers it. This event is the read event labeled with 1. The algorithm then removes the event 1 from the set of possible extensions and checks if it is possible to add any new events to the unfolding that can be executed after event 1. There are no such possible extension events and therefore the event 1 is fired in order to obtain a new marking that corresponds to the current state of the execution.

The algorithm processes the second operation from the sequence and adds the event 2 to the unfolding. For this event the possible extensions algorithm notices that a write event can be performed after the new read event has been fired in the current marking (event 3) or a write event can be performed in a marking where event 2 has been fired but event 1 has not been (event 4). The algorithm knows that thread 2 wants to perform a write operation after executing the event 2 because it can obtain the information directly from the operation sequence generated by the test execution. These two events are added to the set of possible extensions and event 2 is fired to update \( M \). For the last operation, the algorithm adds the event 3 to the unfolding and removes it from the set of possible extensions. No possible extensions are found for this event.

For the second test run, an event from the set of possible extensions is selected as a new test target. In this example the only possibility is the event 4. The algorithm determines from the unfolding that this event can be reached by a schedule that executes two visible operations of thread 2 from the initial state (and after that is free to follow an arbitrary schedule). In the resulting test execution thread 2 assigns a random
Global variables: Thread 1: Thread 2:
X = 0; a = X; b = X;
if (a > 0) if (b == 0)
   error(); X = input();

Figure 3.12. Complete unfolding of the example

input value to x and thread 1 reads this value and branches at the if-
statement based on this value. Let us assume that the false branch is fol-
lowed in this test run. By processing the resulting operation sequence, the
events 4 and 5 get added to the unfolding (for the first operation the COR-
RESPONDINGEVENT subroutine returns the event 2). After the event 5
has been added, the possible extensions algorithm notices that thread 1
will perform next a branch operation that depends on input values. This
leads to adding events for the true and false branches to the set of possible
extensions (i.e., events 6 and 7). Assuming that event 6 corresponds to the
false branch, it is added to the unfolding and removed from the set of pos-
sible extensions. This leaves the event 7 unexplored. For the final test run
this event is selected as the test target. The schedule to reach this event
is computed as before and in addition the path constraint corresponding
to the scheduled execution path is solved. In this case the path constraint
is simply $input > 0$. This constraint is solved to obtain a concrete value
to be returned by the input statement and this value together with the
computed schedule causes the program to reach the error location.
3.5 Integrating the Algorithm with Dynamic Symbolic Execution

The EXECUTE subroutine performs a test execution to cover a previously unvisited event from the set of possible extensions. As an output the subroutine returns a sequence of global and symbolic branching operations in the order in which they are performed during the test execution. In order to do this, the subroutine needs to compute input values and a schedule that forces the execution to cover the target event. From a symbolic execution tree it would be trivial to compute the information needed to cover a specific node in the tree. One just needs to collect the symbolic constraints along the path from the root node to the target node and construct a path constraint from them. This path constraint can be used to obtain the input values. The runtime scheduler can then be forced to execute the execution steps of threads in the same order as they appear in the collected sequence.

With unfoldings the process is similar. To obtain a sequence of events that need to be fired to reach the target, all events that causally precede the target are first collected. With unfoldings this alone is not enough as a valid order in which these events can be fired needs to be determined. This, however, is easy if the testing algorithm labels events with numbers that describe in which order they have been added to the unfolding (i.e., see the labeling with numbers in Figure 3.9). An event with a larger label number cannot causally precede an event with a smaller number. This means that a valid order to fire the events is obtained by sorting the collected events to an ascending order according to the labeling. Given the complete sequence of events, the path constraint and a schedule can be obtained in the same way as from a sequence of edges in a symbolic execution tree as both such sequences correspond to sequences of operations. The path constraint and a schedule can then be used to perform a test execution both symbolically and concretely in the same way as explained in Chapter 2.

**Example 11.** Let us consider the complete unfolding in Figure 3.9 and let us assume that we want to perform a test execution that covers the event labeled with 7. To reach this event, a test execution must execute the operations corresponding to events 7, 6, 5 and 1 (i.e., the target event itself and the events that causally precede it). A valid order to fire these events is obtained by sorting the collected events to an ascending order (1, 5, 6, 7). The path constraint for the execution is a conjunction of the...
constraints stored to events 1 and 7. This constraint is then solved to obtain concrete input values that force a test execution to take the correct branches at conditional statements. The runtime scheduler is additionally forced to perform operations in the same order as they occur in the obtained sequence of events.

Handling Unsatisfiable Path Constraints

If the path constraint that EXECUTE obtains is unsatisfiable, the target event is unreachable and it can be removed from the set of possible extensions. When considering symbolic execution trees, it is clear that a node with an unsatisfiable path constraint is unreachable. However, as an event in an unfolding can be reached by different executions, it may not be immediately clear why it is safe to remove an event from the set of possible extensions in such cases.

In order to cover an event in an unfolding, a test execution must cover all the events that causally precede the target event. The sequence of events computed by EXECUTE corresponds exactly to these events and nothing else. Let $pc$ be the path constraint corresponding to this sequence. It is also possible to reach the target event with an execution that covers additional events. The path constraint $pc'$ of such an execution therefore contains $pc$ and possibly some additional constraints. However, if $pc$ is unsatisfiable, then $pc'$ must be unsatisfiable also as adding constraints to an unsatisfiable path constraint cannot make it satisfiable. Therefore if $pc$ is unsatisfiable, there are no executions that cover the target event.

Further Considerations

Note that the obtained sequence of events up to the target event is not necessarily a prefix of some previously explored sequence as would be the case with algorithms like DPOR. This is because the unfolding algorithm combines information from all previous test execution to compute new test targets (possible extensions). This is one reason why the unfolding based approach can avoid redundant test execution when covering reachable local states of threads. On the downside the path constraints for also other than symbolic branching events need to be checked when they are to be covered by the algorithm. The reason for this is illustrated by the example below. As multiple events can have the same path constraint, caching the solutions for path constraints can be used to avoid unnecessary solver
An example illustrating the need for path constraint checks calls. With symbolic execution tree based approaches, on the other hand, it is possible to cover a different scheduling choice along the previous test execution by re-using the same input values.

**Example 12.** Let us consider the unfolding in Fig. 3.13. In the unfolding it is possible reach a marking where the write event marked with dashed lines is enabled. This means that the testing algorithm adds the event to the set of possible extensions. However, a test execution can never cover the event as the path constraint for it is $input_1 = 2 \land input_1 \neq 2$, which is unsatisfiable. The only way for the testing algorithm to determine that the event cannot be reached when data values are taken into account is to solve the path constraint at some point. The testing algorithm in Fig. 3.11 does this check as part of the EXECUTE subroutine. After noticing that the path constraint is unsatisfiable the algorithm removes the event from the set of possible extensions.

### 3.6 Computing Corresponding Events

The CORRESPONDINGEVENT subroutine determines if a given operation has already been modeled at a given marking $M$ of the unfolding. This can be done by iterating through the events located in the postset of the thread condition in $M$ that belongs to the thread that performs the operation. If an event that is enabled in the given marking is found, the subroutine returns this event. Otherwise the subroutine creates a new event that can be added to the unfolding.

The CORRESPONDINGEVENT subroutine is called for each operation in an operation sequence generated by the testing algorithm. As multiple test executions can have the same operation sequence prefix, the corre-
sponding events for the operations in these prefixes are computed multiple times. If the unfolding has a high branching degree on some of the conditions, this repeated search can be costly. However, a test execution that covers a target event fires the same events used in the computation of the schedule and path constraint. This means that events the test execution will fire are known up to the point where the target event is reached. Therefore, CORRESPONDINGEVENT can be optimized by returning events in the order they are scheduled to be fired instead of searching for enabled events. Only after the target event is reached, the subroutine needs to iterate through events as explained above.

Here we have assumed for simplicity that a test execution always reaches its target event. This is the case for programs that have been fully instrumented to enable symbolic execution. If we also allow testing of only partially instrumented programs (e.g., ones that call system libraries whose source code is not available), additional checks are needed to determine if the test execution follows the assumed execution path.

3.7 Limitations

While discussing the unfolding algorithm presented in this chapter we have assumed that all shared variables are explicitly defined in the beginning and the number of threads is fixed. It is possible to extend the algorithm to support shared variables that are created dynamically at runtime. In order to do this each such variable needs an unique identifier across all possible executions. For example, in the context of Java programs it is possible to obtain such identifiers by adding an event to the unfolding when a new object is created and then identifying a shared variable (a field of this object) by a combination of the object creation event and the field name of the variable. Static variables can be identified by their class and field names. Notice also that the concrete execution in dynamic symbolic execution gives precise aliasing information for the variables. Therefore, it is always known if two references point to the same shared variable.

Handling a dynamic number of threads is more challenging and the implementation of the algorithm in Fig. 3.11 used in the experiments of this thesis does not support dynamic thread creation. The main problem is that when a thread performs a write operation, the corresponding write event accesses shared variable conditions from all threads in the program.
It is not enough to consider only those threads that are running at the
time the operation is executed as it is possible that there exists an execu-
tion of the program where there are additional threads running and the
write affects the behavior of those threads as well. One simple way to
address this problem is to update each write operation in the unfolding to
access the conditions of a new thread when the thread is added to the un-
folding. This, however, is an expensive operation. Another way is to model
the thread creation with an event that reads all the shared variables of
the parent thread and writes the shared variables to the child thread. The
problem with this approach is that operations that create threads can now
be considered to be in race with write operations and this can lead to un-
necessary test executions. A third option is to model the shared variables
differently. In Chapter 5 we present an approach to construct unfoldings
of programs in such a way that dynamic thread creation becomes easy to
model.

3.8 Correctness

In the following we will show that the testing algorithm described in this
chapter covers all feasible control flow paths of each thread in a multi-
threaded program and therefore finds a path to an error statement if it
is reachable in the program. We will first prove some auxiliary lemmas
that help us to prove the main result. Recall the definitions from Chap-
ter 2 that say that an execution of a program is a sequence of performed
operations and that $EX(P)$ is the set of all possible executions.

Lemma 1. Let $o_i$ and $o_j$ be operations in an execution $ex \in EX(P)$ and let
e_i and e_j be events that model $o_i$ and $o_j$, respectively, in an unfolding of $P$.
It holds that $e_i \leq e_j$ if an only if $o_i \rightarrow_{ex} o_j$.

Proof: ($\Leftarrow$): Let us first show that if $o_i \rightarrow_{ex} o_j$, then $e_i \leq e_j$. If $i = j$,
then trivially $e_i = e_j$. Given that $o_i \rightarrow_{ex} o_j$ holds, we can therefore in the
following assume that $i < j$. Let $o_k$ be the operation in $ex$ for which it
holds that $k$ is the largest index such that $k \neq j$ and $o_k \rightarrow_{ex} o_j$. There are
now four possible cases: (i) $o_i <^\text{seq}_{ex} o_j$ and $k = i$, (ii) $o_i <^\text{mem}_{ex} o_j$ and $k = i$,
(iii) $o_k <^\text{seq}_{ex} o_j$ and $k \neq i$, or (iv) $o_k <^\text{mem}_{ex} o_j$ and $k \neq i$.

Let us consider the case (i). If $o_i <^\text{seq}_{ex} o_j$, the operations must be per-
formed by the same thread. From the modelling constructs it is easy to
see that there must exists a path of arcs from the event $e_i$ to the event $e_j$.
through a sequence of thread conditions that belong to the same thread. Therefore if \( o_i \prec^{eq} o_j \), then \( e_i \leq e_j \).

Let us now consider the case (ii). As both operations access the same shared memory location, it is easy to see that they are modeled with events such that there is a common condition in \( (e_i)^\bullet \) and in \( \bullet e_j \). This immediately implies that \( e_i \leq e_j \).

Let us finally consider the cases (iii) and (iv). Let \( e_k \) be the event that models the operation \( o_k \). From cases (i) and (ii) it follows that \( e_k \leq e_j \) in cases (iii) and (iv), respectively. This means that \( e_i \leq e_j \) holds if we can show that \( e_i \leq e_k \) holds. This can be done by recursively analyzing the case \( o_i \rightarrow_{ex} o_k \) in the same way as above. The next recursion step either ends in case (i) or (ii) or has to show that \( o_i \rightarrow_{ex} o_l \) holds, where \( l \geq i \) and \( l < k < j \). Therefore there can be only a finite number of recursion steps before the recursions ends either in case (i) or (ii).

\( \Rightarrow \) : Let us now show the other direction. It is again trivial to see that if \( e_i = e_j \) then the only possibility is that \( i = j \) in which case \( o_i \rightarrow_{ex} o_j \) holds. If \( e_i < e_j \), then directly from the definition of causality between events it follows that there must exist a sequence of events from \( e_i \) to \( e_j \) such that for every adjacent events \( e \) and \( e' \) in the sequence there is a common condition in the postset of \( e \) and in the preset of \( e' \). If the common condition is a thread condition, then the operations \( o \) and \( o' \) modeled by \( e \) and \( e' \), respectively, are sequentially related. If the common condition is either a shared variable or a lock condition, then the corresponding operations must access the same shared memory location. This means that the operations are shared memory access precedence related. Therefore it holds that \( o \rightarrow_{ex} o' \). As this holds for every adjacent events \( e \) and \( e' \) in the sequence, it follows that \( o_i \rightarrow_{ex} o_j \).

**Definition 10.** The local state of a thread \( t \) that is reached after an execution \( ex \in EX(P) \) is denoted as \( l_t(ex) \).

**Definition 11.** Let \( ex \) and \( ex' \) be two executions in \( EX(P) \). We denote that \( l_t(ex) \sim l_t(ex') \) if (i) the program location of thread \( t \) is the same after both executions and (ii) for any local variable \( v \) of thread \( t \) it holds that \( v \) has the same concrete value or the same symbolic value expression in the symbolic local states of \( t \) that are reached when \( ex \) and \( ex' \) are executed symbolically.

**Example 13.** Let the local state \( l_t(ex) \) have the local variable valuations \( \{a = 5, b = 8\} \) and \( l_t(ex') \) have the local variable valuations \( \{a = 5, b = 0\} \). Furthermore, let the program locations of thread \( t \) that are reached after
ex and ex′ be the same. The local states are not equal because the value of the local variable b is not the same in both of them. Let s and s′ be the symbolic local states reached by symbolically executing ex and ex′, respectively, such that the variable valuations in both symbolic states are \{a = 5, b = input_1 + 1\}. In this case it would hold that \(l_t(ex) \sim l_t(ex').\) However, if the valuation of b in s′ would be, for example, \(b = input_1 + 2\), then \(l_t(ex) \sim l_t(ex')\) would not hold. Note that the path constraints for the executions ex and ex′ are not required to be the same for \(l_t(ex) \sim l_t(ex')\) to hold.

In the following we assume that if an input operation can be concurrently enabled with another input operation, it always returns the same symbolic input value expression regardless in which order the input operations are performed. This can be achieved by having a thread t to return a symbolic input value \(input^t_1\) for the first input operation it performs in any given execution, \(input^t_2\) for the second input operation in the same execution, and so forth. This way the order in which different threads execute input statements do not affect the symbolic value expressions.

**Lemma 2.** Let ex and ex′ be two executions in EX(P). If the last operation of thread t is modeled by the same event e in the unfolding of P in both executions, then \(l_t(ex) \sim l_t(ex').\)

**Proof.** Let o and o′ be the last operations of t in ex and ex′, respectively. Let us consider an execution ex₁ that is otherwise the same as ex except that it performs an operation op if and only if \(op \rightarrow_{ex} o\). Let ex′₁ be an execution constructed similarly from ex′ and o′. It is easy to see that \(l_t(ex) = l_t(ex₁)\) and \(l_t(ex') = l_t(ex'₁)\).

As both o and o′ are modeled by the event e, from Lemma 1 it follows that ex₁ and ex′₁ are both linearizations of the \(\rightarrow_{ex₁}\) partial order and perform the same operations but possibly in different orders. It also holds that it is possible to transform the sequence ex₁ into ex′₁ by performing a finite number of swaps between adjacent independent operations. This is because if some dependent operations are performed in different orders in the executions, then o and o′ would not be modeled by the same event according to Lemma 1. As swapping adjacent independent operations do not affect the visited local states of threads in the executions, it is easy to see that \(l_t(ex₁) \sim l_t(ex'₁)\) holds, which implies \(l_t(ex) \sim l_t(ex')\).
Thread 1:
1: \( a = X \);
2: if (\( a == 0 \))
3: \( X = 1 \);

Thread 2:
4: \( X = 2 \);

**Figure 3.14.** An example of an incorrect way to model test executions

the same control flow path after the event until the next symbolic branching or scheduling point in both executions. From Lemma 2 it is easy to see that this holds for the algorithm in Fig. 3.11. If this were not the case, the testing algorithm would not know for certain what types of events can be in the postsets of thread conditions and therefore does not have enough information to compute all possible extensions. As an example, let us consider what happens if the write operations were modeled in the same way as read operations. Figure 3.14 shows a simple program and an unfolding that incorrectly models an execution where the write operation of thread 2 is performed before the read of thread 1. As both ways to interleave the read and write operations lead to the same marking, the testing algorithm incorrectly assumes that thread 1 always terminates after the execution step that starts with the read operation is performed. Therefore it never explores the true branch at line 2. In this example the local state \( l \) of thread 1 after the read event has a valuation \( \{ a = 0 \} \) in an execution that performs the read first. If the write is performed first, the local state \( l' \) of \( t \) after the read has a valuation \( \{ a = 1 \} \). Therefore \( l \sim l' \) does not hold and the incorrect way of modeling violates Lemma 2.

The modeling constructs discussed in this chapter are not the only ones that can be used correctly with the testing algorithm. For example, it would be possible to use constructs without place replication as shown in Fig. 3.7. However, the algorithm would generate more test cases as it would explicitly explore different interleavings of read operations. In Chapter 5 we discuss a third kind of a modeling approach that is even more compact than the approach discussed here but still satisfies Lemma 2.

**Lemma 3.** For every execution \( ex \in EX(P) \) there is a set \( S \) of events in the unfolding of \( P \) that represents \( ex \).

**Proof.** Let us assume that the statement above does not hold and \( S \) does
not exist. Let $ex'$ be the longest prefix of $ex$ that is still represented by a set $S'$ of events in the unfolding. Let $o$ be the next symbolic branch or global operation that is performed in $ex$ after the prefix $ex'$. If $ex'$ is empty, $o$ corresponds to an event enabled in the initial marking of the unfolding. The subroutine `INITIALLYENABLEEVENTS` trivially finds all such events and adds them to the set of possible extensions.

Let us now consider the case where $ex'$ is not empty. Let $t$ be the thread that performs $o$ and let $e_t$ be the last of those events that model operations of $t$ in $S'$. A test execution $ext$ performed by the testing algorithm has covered the event $e_t$. Otherwise $e_t$ would not be in the unfolding. Given Lemma 2, it is easy to see that after $e_t$ the type of the next operation (that is modeled in the unfolding) that $t$ performs is the same in both $ex_t$ and $ex$. This means that the testing algorithm knows the type of the operation $o$.

As $S$ does not exist, an event $e$ corresponding to $o$ must be missing from the unfolding. The marking $m$ of the unfolding that is reached after firing the events in $S'$, by construction, contains thread conditions for all threads and also shared variable conditions for all shared variables. This means that if $o$ is a symbolic branching, unlock, read or write operation, the event $e$ is enabled in the marking $m$. If $o$ is a lock operation, there are either no acquires of the same lock in $S'$ or the previous operation accessing the lock releases it. Otherwise $o$ would not be enabled after executing $S'$. In the first case $m$ contains the lock condition in the initial marking and in the latter case $m$ contains the lock condition in the postset of the unlock event that corresponds to the last unlock in $S'$. This means that in all cases $e$ is enabled in $m$. As the algorithm knows the type of $o$, it has enough information to compute possible extensions and therefore it must have added $e$ to the set of possible extensions.

The testing algorithm does not add a possible extension to the unfolding if its path constraint is unsatisfiable. Therefore we still need to show that the path constraint for the possible extension is satisfiable. The path constraint $p$ constructed during a symbolic execution that follows the original execution up to and including $o$ is clearly satisfiable. As discussed in Sect. 3.5, the path constraint used by the testing algorithm to determine if a possible extension event is reachable, is either $p$ or a more relaxed version of it (i.e., the set of symbolic constraints is a subset of those in $p$). As $p$ is satisfiable, the path constraint used by the testing algorithm must be satisfiable also. As the algorithm does not terminate before all possible
extensions with a satisfiable path constraint have been added to the unfolding, the event $e$ must exist in it. This contradicts our assumption and therefore the lemma must hold.

Based on Lemma 3, an unfolding of a program represents all possible executions. The unfolding algorithm, however, does not explicitly test them all but still guarantees that all feasible control flow paths of individual threads get explicitly explored by at least one test execution.

**Theorem 1.** The algorithm in Fig. 3.11 explores a control flow path of a thread in a multithreaded program $P$ if and only if the path is feasible.

**Proof.** Let us first show that if a control flow path of a thread is feasible, the testing algorithm performs a test execution that follows that path. Let us assume that there is a control flow path $p$ of thread $t$ that is not explored. If $p$ is feasible, there must exist an execution $ex \in EX(P)$ where $t$ follows $p$. From Lemma 3 we know that this execution is represented by a set of events in the unfolding constructed by the testing algorithm. Let $e$ be an event that corresponds to the last operation of $t$ that is modeled in the unfolding. The testing algorithm must have performed a test execution that covers $e$ in order for it to be in the unfolding. As the test execution and $ex$ both cover $e$, it is easy to see from Lemma 2 that $t$ has followed the same control flow path in the test execution as well as in $ex$. This contradicts the assumption that $p$ is not explored by the testing algorithm.

The direction stating that if a control flow path is explored by the algorithm, it is feasible, holds trivially. This is because the test executions are performed on a concrete program and therefore they cannot follow a control flow path that is not feasible.

**Corollary 1.** If an error statement is reachable in a multithreaded program $P$, the algorithm in Fig. 3.11 performs a test execution that leads to the error statement.

**Proof.** An error statement is a local statement of some thread and it is reachable if and only if there is a feasible control flow path of that thread that leads to the statement. The theorem therefore follows directly from Theorem 6.
3.9 Discussion

We will next discuss some additional properties of the unfoldings.

**Covering multiple possible extensions with a single test.** In the set of possible extensions there can be multiple events that are concurrent. It is therefore possible that one test run covers more than one possible extension. This property can provide further reduction to the number of needed test runs especially in situations where there are independent components in the program under test and the random scheduler and random inputs have a good chance of exploring different execution paths. It is also possible to compute a schedule and input values for the next test execution such that it will cover the maximum number of possible extension events. This computation, however, is potentially computationally expensive (we conjecture that it is NP-hard).

**Isomorphism of the unfoldings.** Another property of the generated unfoldings is that if the testing process is repeated multiple times, the resulting unfoldings are isomorphic. This is not necessarily true when using trees and partial order reduction based approaches. It is possible to take advantage of this property to partially check the correctness of the implementation of the algorithm. For example, it is possible to compare the resulting unfoldings of the same program from two different implementations to check if they are isomorphic. It is also possible to exhaustively test all possible interleavings of a program, which is easy to implement correctly, build an unfolding based on these tests and then compare the result with the unfolding based testing algorithm. Although this cannot show that the implementation is correct, it can be an useful testing and debugging aid while developing unfolding based algorithms.

**The length of a test execution to reach a possible extension.** The unfolding approach can also in some cases detect errors that DPOR can miss when using the same bound on the execution length. This is because the execution that DPOR follows to a test target (i.e., a backtrack point) is a prefix of some earlier test execution. Such execution, however, is not necessarily the shortest one to the target. The unfolding approach, however, does not execute any unnecessary operations to reach a target event. This means that the unfolding approach can detect some errors with shorter executions and therefore with a smaller bound on execution lengths. Naturally, DPOR can detect the same error if it follows the same execution
as the unfolding approach. However, it is possible that DPOR ignores the part of the execution tree containing the shortest path to the error. This observation relates to the fact that the parts of the execution trees explored by DPOR can differ depending on the order in which it explores the execution paths.

**Combining with other test generation approaches.** It is also possible to generate an initial unfolding based on tests performed with random input values or by some heuristic approach, model these test executions as an unfolding and then compute possible extensions to increase the coverage of the initial tests. If the test executions were represented by an execution tree, this would not be as easy as two executions belonging to the same Mazurkiewicz trace can follow different branches in the execution tree making it difficult to use partial order reduction algorithms efficiently. With unfoldings the events corresponding to the executions following the same Mazurkiewicz trace are always the same.

**Global state reachability.** Finally, the generated unfolding represents all interleavings, all deadlocks are also represented in it. Therefore with an additional search, it is possible to extend the unfolding based testing algorithm to detect deadlocks as well. Also, if the testing algorithm collects additional information during testing, such as stores the concrete or symbolic values of shared variables to the shared variable conditions in the unfolding, it is possible use the unfolding to answer reachability questions such as is there a global state in the program where the sum of two shared variables is over some given limit. We will show how deadlocks and violations of other global state reachability properties can be detected in Chapter 6.
4. Efficient Possible Extensions Computation

In this chapter we describe an efficient way to compute possible extensions for the types of unfoldings generated by the algorithm described in Chapter 3. If we constructed an explicit Petri net model of the program, this could be done by using any existing possible extensions algorithm developed for Petri nets. These algorithms, however, are typically designed for arbitrary Petri nets and they are computationally the most expensive part of the unfolding process. In this section we show that the structure of the unfoldings generated by the testing algorithm is limited in such a way that it is possible to compute possible extensions more efficiently than for unfoldings of arbitrary Petri nets.

4.1 Basic Idea

A possible extension is an event that has not yet been added to an unfolding but could be fired in some reachable marking of it. If possible extensions are computed every time an event is added to the unfolding, it is enough to consider only those reachable markings that contain at least one condition from the postset of the event that has most recently been added to the unfolding [ER99]. This is because for other reachable markings the possible extensions have already been computed. In the context of our testing algorithm the conditions in a postset of any event can be either thread conditions, shared variable conditions or lock conditions. This means that it is possible to handle these three cases separately as done in Fig. 4.1 that shows the top level of the possible extensions algorithm that we will describe in this chapter.

As discussed in the previous chapter, a thread always performs the same type of an operation (and therefore the same type of an event) next when its execution reaches a thread condition. This means that to find possible
Figure 4.1. Top level of the possible extensions algorithm: dividing the computation into three cases

extensions from a thread condition \( t \), the algorithm needs to find all sets of conditions that can form a preset for an event that models the operation the thread wants to perform next. For example, for a read operation this means finding all shared variable conditions of the variable being read that are concurrent with \( t \) and belong to the thread performing the read operation. A naive and inefficient way to perform the search is to iterate through all suitable shared variable conditions and check if they are concurrent with \( t \).

As the events correspond to operations in a multithreaded program, there is another way to think of the search problem. For example, assume that we want to find possible extensions that correspond to read operations performed by thread 1. If one such extension corresponds to a read performed after a write by thread 2, a good candidate for another possible extension is a read event that is performed before the write. That is, possible extensions can be found by considering different interleavings of dependent transitions. In the following we show that such a search can be done efficiently for the unfoldings generated by our algorithm. Furthermore, we show that the search finds all possible extensions that \texttt{EXTFROMTHREADCONDITION} in Fig. 4.1 needs to return. This approach is similar to the backtrack search performed by DPOR. In fact, the possible extensions algorithm presented in this thesis was originally inspired by DPOR.

The possible extensions from a shared variable or lock condition \( c \) are either read, write or lock type events. This means that the possible extensions algorithm needs to find a thread condition \( t \) that is concurrent
with $c$ such that a thread wants to perform an operation that accesses the shared variable or lock after reaching $t$. For write events additional shared variable conditions to complete the preset of the extension event need to be also searched. It is again possible to perform a naive search that goes through all potential conditions. However, due to the structure of the unfoldings, the search space can be greatly limited as we will show later in this chapter.

### 4.2 Structural Connection Between Read And Write Events

Before showing how the subroutines in Fig. 4.1 can be implemented, we describe structural properties of the unfoldings generated by our testing algorithm that allow possible extensions to be computed efficiently.

**Definition 12.** Let $S_1$ and $S_2$ be two sets of shared variable conditions in an unfolding of a program such that $|S_1| = |S_2|$ and $S_1 \neq S_2$. The sets $S_1$ and $S_2$ are adjacent if there exists either a read or a write event $e$ such that firing $e$ from any reachable marking containing $S_1$ leads to a marking containing $S_2$.

**Example 14.** Let us consider the net in Fig. 4.2 and the following sets of shared variable conditions: $a = \{c_1, c_2\}$, $b = \{c_1, c_4\}$, $c = \{c_3, c_4\}$ and $d = \{c_5, c_6\}$. The sets $a$ and $b$ are adjacent as $a$ can be "transformed" into $b$ by firing the event 2. The sets $a$ and $c$ are not adjacent as transforming $a$ into $c$ requires two events to be fired. The sets $c$ and $d$ are again adjacent through the event 5.
Efficient Possible Extensions Computation

It is easy to see that from a shared variable condition in the initial marking it is possible to get to any other shared variable condition representing the same shared variable and belonging to the same thread through a sequence of adjacent shared variable conditions. For example, in Fig. 4.2 $c_1$ is adjacent to $c_3$ which is adjacent to $c_5$. Adjacent shared variable conditions therefore can be seen as forming a tree where the shared variable condition in the initial marking is the root of the tree. This is illustrated in Fig. 4.3 where all the shown conditions represent the same shared variable and belong to a thread $t$. Let us assume that we want to compute read type possible extensions from a thread condition $c_t$ belonging to the thread $t$. This means that we want to determine which of the conditions from $c_1$ to $c_5$ in Fig. 4.3 are concurrent with $c_t$. A naive algorithm could start from $c_1$ and then traverse the edges in Fig. 4.3 to find the rest of the shared variable conditions. For each of the found conditions it would then perform a co-check.

It is often possible to speed up such a search considerably. Let us assume that $c_1$ is concurrent with $c_t$ but for $c_2$ it holds that $c_t < c_2$ or $c_t \neq c_2$. It is then easy to see that also for conditions $c_3$, $c_4$ and $c_5$ it holds that they are either causally related to $c_t$ or in conflict with it. Therefore a search algorithm does not need to consider the conditions $c_3$, $c_4$ and $c_5$ at all if $c_2$ is not concurrent with $c_t$.

Let us consider another example where the search is started from some other condition than $c_1$. Let us assume that we start from $c_3$ and notice that it is concurrent with $c_t$. We could then move up in the tree and find the condition $c_2$. Let us assume that it holds that $c_2 < c_t$. Then it is
easy to see that also for $c_1$ it holds that $c_1 < c_t$. Therefore the search for shared variable conditions that are concurrent with $c_t$ can be limited in such a way that sometimes only a small part of the unfolding needs to be considered. In the following we generalize this idea also to write and lock type possible extensions.

**Lemma 4.** Let $L = S_1, S_2, ..., S_n$ be a sequence of sets of shared variable conditions such that for all $1 \leq i < n$, it holds that $S_i$ is adjacent to $S_{i+1}$ such that at least one condition in $S_i$ causally precedes a condition in $S_{i+1}$. If the sets $S_1$ and $S_n$ are concurrent with a condition $t$, then all sets in the sequence $L$ are as well.

**Proof:** Let $c_1$ be a condition in $S_1$ and let $c_n \in S_n$ and $c' \in S_i$, where $1 < i < n$, be conditions that represent the same shared variable as $c_1$ and belong to the same thread. It is easy to see that for the conditions it must hold that $c_1 \leq c' \leq c_n$.

If $c' = c_1$ or $c' = c_n$, then $c'$ is trivially concurrent with $t$. Therefore assume that $c_1 < c' < c_n$ and $c'$ is not concurrent with $t$. There are three possible cases: (i) $c' < t$, (ii) $t < c'$ and (iii) $t \neq c'$. Case (i) implies that $c_1 < t$, case (ii) implies that $t < c_n$ and case (iii) implies $t \neq c_n$. All these cases contradict the assumption that $c_1$ and $c_n$ are concurrent with $t$. Therefore it must hold that $c'$ co $t$. As the same analysis can be done for all conditions in the sequence $L$, they must be concurrent with $t$. □

**Theorem 2.** Let $t$ be a thread condition in an unfolding such that the next operation from $t$ is either a read or a write. Let $S$ be a set of all sets of shared variable conditions that together with $t$ can form a preset of a read or a write type event. Finally, let $G$ be a graph such that there is a vertex in $G$ for every set in $S$ and an edge between two vertices if the respective sets are adjacent. Then the graph $G$ is connected.

**Proof:** Let $G'$ be a graph that has a vertex for every co-set of shared variable conditions that represent the same shared variable as the sets in $S$ and has an edge between two vertices if their respective sets are adjacent. That is, the graph $G'$ is similar to $G$ except that the requirement for the conditions corresponding to vertices to be concurrent with $t$ has been removed. Clearly $G$ is a subgraph of $G'$. Furthermore, $G'$ is connected as there exits a vertex $S_0$ in $G'$ such that $S_0$ contains the conditions that are initially marked in the unfolding and all other co-sets representing the same shared variable must be reachable from this initial marking.
Let us now assume that $G$ is not connected. This means that there exits two vertices $S_1$ and $S_2$ such that there is no path between $S_1$ and $S_2$ in $G$. Let us consider the vertex $S_0$. If $S_0$ is also a vertex of $G$ and $S_0 = S_1$ or $S_0 = S_2$, then according to Lemma 4, there must exist a path from $S_1$ to $S_2$ such that all conditions in the intermediate vertices are concurrent with $t$. If $S_0 \neq S_1$ and $S_0 \neq S_2$, there exists a path from $S_1$ to $S_2$ via $S_0$. Therefore in these cases the graph $G$ is connected.

Let us now consider the case where $S_0$ is not a vertex of $G$. This means that there exists a condition in $S_0$ that causally precedes $t$. Let $M$ be the marking that is reached by firing all events that causally precede $t$ from the initial marking. No matter in which valid order the events are fired, the resulting marking $M$ is always the same. Let $S'_0 \subset M$ be the set representing the same shared variable as the sets in $S$. $S'_0$ must exist as the unfolding is constructed such that in every reachable marking one condition corresponding to each thread's shared variable copy is marked. The conditions $S'_0$ are trivially concurrent with $t$. Furthermore, all sets that are concurrent with $t$ and represent the same shared variable as $S'_0$ must be reachable by firing sequences of events from a marking containing $S'_0$. Therefore there exists a path in $G$ from the vertex corresponding to $S'_0$ to the vertex corresponding to $S_1$ and also similarly to $S_2$. According to Lemma 4, the intermediate vertices in these paths must be concurrent with $t$ and therefore there exist a path between $S_1$ and $S_2$ in $G$. \hfill \Box

Based on Theorem 2, when we want to compute possible extensions starting from a thread condition $t$ and we already known one event from $t$, the rest of the possible extensions can be found by using the set of shared variable conditions in the preset of the event as a starting point. The search can then recursively consider adjacent sets of shared variable conditions and backtrack when the set under consideration is not concurrent with $t$. In typical cases this can speed up possible extensions computation considerably but in the worst case the whole unfolding still needs to be searched.

### 4.3 Structural Connection Between Lock Events

When considering possible extensions relating to lock events, we cannot directly use the result from the previous section. This is because with lock conditions the assumption that in each reachable marking there is
one condition representing the lock does not hold. However, a similar result can be obtained for locks as well.

**Definition 13.** Let \( l_1 \) and \( l_2 \) be lock conditions such that \( l_1 \in \cdot e_1 \) and \( l_2 \in \cdot e_2 \). The lock conditions \( l_1 \) and \( l_2 \) are adjacent if (i) they represent the same lock, (ii) \( e_1 < e_2 \), and (iii) there is no lock event \( e_3 \) such that \( e_1 < e_3 < e_2 \) where the lock condition in \( \cdot e_3 \) represents the same lock as \( l_1 \) and \( l_2 \).

Intuitively adjacent lock conditions correspond to executions where a thread acquires a lock (event \( e_1 \)), performs some computation and then releases the lock (event \( e_2 \)).

**Definition 14.** Let \( l_1 \) and \( l_2 \) be lock conditions in postsets of different unlock events and let both conditions represent the same lock. Let \( e_1 \) and \( e_2 \) be the closest causally preceding lock events of \( l_1 \) and \( l_2 \), respectively, such that they correspond to acquiring the lock represented by \( l_1 \) and \( l_2 \). If \( e_1 = e_2 \), then the conditions \( l_1 \) and \( l_2 \) are alternatives.

Intuitively alternative lock conditions result from executions where the same lock acquire event is fired but after that the executions branch into different paths. The conditions in the postsets of the next unlock events in these paths are alternatives.

**Example 15.** In Fig. 4.4 the lock conditions \( l_1 \) and \( l_2 \) are adjacent. The lock conditions \( l_2 \) and \( l_3 \) are alternatives. The conditions \( l_2 \) and \( l_4 \), however, are not alternatives although both of them are preceded by the same lock condition \( l_0 \). This is because the closest preceding lock events for these conditions are not the same (i.e., events 1 and 6 are different).

Note that lock conditions that are adjacent or alternative to a given lock condition can be found efficiently if two kinds of mappings are maintained. One that maps lock events to their corresponding unlock events (e.g., event 1 is mapped to events 3 and 5 in Fig. 4.4) and a second one that maps unlock events to their corresponding lock events (i.e., event 3 is mapped to event 1).

**Theorem 3.** Let \( t \) be a thread condition in an unfolding such that the next operation from \( t \) is a lock acquire. Let \( S \) be a set of all lock conditions that are concurrent with \( t \) and represent the lock that a thread wants to acquire when reaching the state represented by \( t \). Let \( G \) be a graph such that there is a vertex in \( G \) for every lock condition in \( S \) and an edge between two
vertices if the respective lock conditions are adjacent or alternatives. Then the graph $G$ is connected.

Proof. Let $G'$ be a graph that has a vertex for every lock condition that represents the same lock as the lock conditions in $S$ and has an edge between two vertices if their respective lock conditions are adjacent. Note that no edges between alternative lock conditions are added. It is easy to see that the graph $G'$ is a tree such that the vertex corresponding to the initial lock condition is the root of the tree. As each lock condition can be reached from the initial lock condition by an execution where the observed lock conditions are pairwise adjacent, $G'$ contains all vertices in $G$ (an possibly more). In $G'$ there must therefore exist a minimal subtree that contains both vertices corresponding to $c_1$ and $c_2$. Let $c_0$ be the condition corresponding to the root of this subtree.

There are two possible cases: (i) $c_0 = c_1$ or $c_0 = c_2$, or (ii) the paths from $c_0$ to $c_1$ and $c_2$ in $G'$ take different branches at the vertex corresponding to $c_0$. Let us consider the case (i). It must hold that either $c_1 < c_2$ or $c_2 < c_1$. Let us assume that $c_1 < c_2$ holds (the other case is symmetrical). For $G$ to be not connected, there has to be a vertex in $G'$ such that the lock condition $c'$ corresponding to this vertex is not concurrent with $t$ and $c_1 < c' < c_2$. As $c'$ is not concurrent with $t$, it has to be either causally related or in conflict with it. However, neither of these cases is possible. If $c'$ is causally related to $t$, then either $c_1$ or $c_2$ has to be causally related to $t$, which contradicts the assumption that these conditions both are concurrent with $t$. Similarly, if $c'$ is in conflict with $t$, then $c_2$ is also in conflict with $t$. 

Figure 4.4. A net with alternative and adjacent lock conditions
Therefore $c'$ cannot exist and the same path as in $G'$ from $c_1$ to $c_2$ exists also in $G$.

Let us now consider the case (ii). If $c_0$ is concurrent with $t$, then by the same reasoning as in case (i) all the conditions on paths from $c_0$ to $c_1$ and to $c_2$ are also concurrent with $t$. If $c_0$ is not concurrent with $t$, there are three possibilities: $t < c_0$ or $c_0 \notin t$, $c_0 < t$. The first two cases imply $t < c_1$ and $t \notin c_1$, respectively, which contradicts the assumption that $t \in c_1$.

The last case implies that there must exist a lock event $e$ that has $c_0$ in its preset such that $e < t$. If the paths from $c_0$ to $c_1$ and $c_2$ require firing different events at a marking containing $c_0$, then either $c_1$ or $c_2$ has to be in conflict with $t$ which is again a contradiction. Therefore the only possibility is that $e < c_1$ and $e < c_2$. As $c_1$ and $c_2$ are in different branches in $G'$, the next unlock events after firing $e$ must be different in paths from $c_0$ to $c_1$ and to $c_2$. This means that the next vertices with conditions $c'_1$ and $c'_2$ in the paths from $c_0$ to $c_1$ and $c_2$, respectively, are alternative. Let us consider the conditions $c'_1$ and $t$. If $c'_1 < t$, then $c_2 \notin t$. As $c'_1 = c_1$ or $c'_1 < c_1$ it holds that if $t < c'_1$ or $t \notin c'_1$, then $t < c_1$ or $t \notin c_1$, respectively. This contradicts the assumption that $c_1 \in t$ and means that the only possibility is that $c'_1 \in t$. The same holds symmetrically for $c'_2$ as well. Based on case (i) there is a path from $c'_1$ to $c_1$ and $c'_2$ to $c_2$ in $G$. As $c'_1$ is alternative to $c'_2$, there is a path in $G$ from $c_1$ to $c_2$. As such a path can be found in all cases, $G$ is connected.

4.4 Possible Extensions from Thread Conditions

We can now describe how the EXTFROMTHREADCONDITION subroutine in Fig. 4.1 can be implemented. As discussed earlier, all events in the postset of a thread condition $t$ have the same type. Therefore when computing possible extensions from $t$, all the possible extension events have the same type. As there are five different types of events in the unfoldings constructed by the testing algorithm, we provide a case analysis of how to handle each of these event types. In other words, the EXTFROMTHREADCONDITION subroutine checks what type of an operation the thread wants to perform after reaching $t$ and based on that follows one of the five cases below.

**Case 1: branching on a symbolic value.** If a symbolic branch operation is performed after reaching the thread condition $t$, computing possible
extensions is trivial as the only possible events are events for the true and false branches. This means that no search needs to be performed as both of these events can be added directly to the set of possible extensions. Note that the EXECUTE subroutine in the testing algorithm is responsible for checking if the path constraints for these events are satisfiable.

Case 2: reading a shared value. If the events in the postset of \( t \) are read events, there can be multiple read type possible extensions. If one of the possible extension events is known, the rest can be found by recursively searching the adjacent shared variable conditions starting from the one in the preset of the known possible extension. In other words, it is checked if any condition \( c \) adjacent to the initially known shared variable condition is concurrent with \( t \). In case it is, an read type event with \( t \) and \( c \) in its preset is added to the set of possible extensions. The search is then continued recursively by checking those conditions that are adjacent to \( c \) and have not yet been covered by the search. By Theorem 2 this search finds all read type possible extensions from \( t \).

Finding the possible extension to act as the starting point for the search is also easy. This is because in any reachable marking, exactly one condition for each copy of the shared variable is marked (i.e., in the Petri net model the shared variable places are permanently marked). This means that in any reachable marking that contains \( t \), there is also a shared variable condition that is guaranteed to be concurrent with \( t \). As the test execution that resulted in the addition of the thread condition \( t \) is tracked by the marking \( M \) maintained by the testing algorithm, a shared variable condition that is concurrent with it can be directly obtained from \( M \).

Case 3: writing a shared value. Finding write type possible extensions can be done in the same way as with read type events. The only difference is that the initially found possible extension has a set of shared variable conditions in its preset. The recursive search must then be performed on sets of shared variable conditions instead on single conditions.

Case 4: acquiring a lock. If a thread wants to acquire a lock \( l \) after reaching \( t \), finding the initial possible extension is not as straightforward. This is because it is not guaranteed that in the marking \( M \) maintained by the testing algorithm there is a lock condition labeled with \( l \). If the marking \( M \) contains such a condition, then the search for possible extensions can be done in the same way as for read events except that alternative lock conditions need to be searched in addition to adjacent ones. Based on
Theorem 3 such search finds all lock type possible extensions.

In the case where the marking $M$ does not contain the lock condition, a starting point for the search needs to be obtained differently. Note that if in the test execution leading to $M$ the thread holding the lock were to release it as the next operation, then the lock condition in the postset of the corresponding unlock event would be concurrent with $t$. Therefore a starting point for a recursive search can be obtained by temporarily adding an unlock event to the unfolding. Naturally the initial possible extension with the temporary lock condition is not added to the set of possible extensions.

**Case 5: releasing a lock.** From a thread condition there cannot be multiple release lock events as the presets of such events consists of only a single thread condition. Therefore if a thread wants to release a lock after reaching $t$, a release lock event with $t$ in its preset can be added directly to the set of possible extensions.

**Example 16.** To illustrate computing possible extensions from thread conditions, let us consider the net in Fig. 4.5. Assume that the event 6 has just been added to the unfolding and we want to compute possible extensions from the thread condition $t$. Furthermore, assume that the marking in the figure corresponds to the marking reached by the current test execution. If a thread wants to read the shared variable $x$ after reaching the thread condition $t$, all shared variable conditions representing $x$ and belonging to thread 2 needs to be found. One such condition can be obtained directly from the current marking (i.e., condition $c_3$) and this leads to the initial possible extension shown in Fig. 4.5. After this the possible extensions algorithm checks if the conditions adjacent to $c_3$ are concurrent with $t$. Adjacent conditions can easily be located by traversing the arcs and events between shared variable conditions. In the example the only shared variable condition adjacent to $c_3$ is $c_1$. If $c_1$ is concurrent with $t$, a new possible extension event is created and the search proceeds to shared variable conditions adjacent to $c_1$. As $c_3$ has already been processed, the only new such condition is $c_2$.

### 4.5 Possible Extensions from Shared Variable Conditions

Possible extensions from a shared variable condition $c$ are computed when a read or write type event $e$ is added to the unfolding such that $c \in$
Any such possible extension is another read or write type event. If a possible extension and \( e \) belong to the same thread, the subroutine \texttt{EXTFROMTHREADCONDITION} also finds the possible extension. This is because in such case the thread condition in \( e^* \) must also be in the preset of the possible extension. Therefore we can limit our discussion here to cases where \( e \) and the possible extensions belong to different threads.

Intuitively a possible extension can exist only if the transitions corresponding to \( e \) and to the possible extension in a Petri net model of the program can be co-enabled (i.e., there exists a data race). The reason for this is that the possible extension can be fired right after firing \( e \) and as executing a read or write transition does not disable or enable read and write transitions of other threads in the Petri net model, it is possible to execute the transitions also in a different order. To be more precise, let us consider the following lemma.

**Lemma 5.** Let \( e \) be the most recent event added to the unfolding and let \( c \) be a shared variable condition in the postset of \( e \). Let \( e_p \) be a possible extension event from \( c \) and let \( t \) be the thread condition in \( e_p \). Additionally, let \( c' \) be the shared variable condition in \( e \) that is adjacent to \( c \). Then the conditions \( t \) and \( c' \) are concurrent.

**Proof.** The events and conditions discussed in the lemma are illustrated in Fig. 4.6. The conditions \( t \) and \( c' \) cannot be in conflict because in such case \( c \) would be in conflict with \( t \). Similarly \( t < c' \) implies \( t < c \). If \( c' < t \) then there are two possibilities: either \( e < t \) or there has be another event \( e' \in e^* \) such that \( e' < t \). The first case is impossible because when the possible extensions from \( c \) are being computed, there does not exist any other conditions in the unfolding that are causally preceded by \( e \) than the ones in the postset of \( e \) and \( t \) is not among them. The second case directly
implies that $t$ and $c$ are in conflict. As all the cases above are impossible, the only remaining possibility is that $t$ and $c'$ are concurrent.

It is easy to see that in addition to $t$, Lemma 5 holds for any condition in the preset of $e_p$ that is not also in the postset of $e$. It is also possible to replace $c'$ with any condition in the preset of $e$ and the lemma still holds. This means that for every possible extension $e_p$, the complete unfolding of a program contains an event that has the same preset as $e_p$ except that the shared variable conditions that are also in the preset of $e$ are replaced with the conditions in the postset of $e$. In Fig. 4.6 this means that $c'$ and $c''$ are also concurrent and therefore in the complete unfolding there exits an event that has $\{c', t, c''\}$ in its preset (i.e., $c$ has been replaced with $c'$).

As all the conditions in such presets have been added to the unfolding before the event $e$, the events with the replaced conditions must either already exist in the unfolding or be in the set of possible extensions prior to this. As there is such an event for every possible extension, the search for possible extensions from $c$ can therefore be done with the following steps.

1. Collect all events, including known possible extensions, in the postset of the shared variable condition that is adjacent to $c$ (i.e., $c'$ in the preset of $e$ in Fig. 4.6).

2. For each collected event construct a set of conditions that is the same as the preset of the event except that the conditions also in the preset of $e$ are replaced with the conditions in the postset of $e$.

3. If the formed set is a co-set, add a possible extension with the co-set as its preset.

Note that if a possible extension exists, there also exist an event with a preset obtained by the replacement discussed above. The converse, how-
ever, does not necessarily hold. Therefore the co-check in the steps above is necessary. Note also that if $e$ is a write event (i.e., it has multiple shared variable conditions for which possible extensions need to be computed) and a candidate event collected by the search algorithm during step 1 is also a write, it is not necessary to consider each shared variable condition in the postset of $e$ separately. This is because the possible extension needs to have all of the shared variable conditions in it preset and therefore considering the shared variable conditions separately would lead to finding the same possible extension multiple times.

**Example 17.** Consider the net shown in Fig. 4.7 where events 1, 2 and 4 are write type events and event 3 is a read type event. To compute possible extensions from the shared variable condition $c_2$, the algorithm collects events 1, 2 and 3 as they occur in the postset of $c_2'$. For event 1 the algorithm checks if $\{c_1, c_2, t_2\}$ is a co-set (i.e., the conditions $c_1'$ and $c_2'$ have been replaced from the preset of the event). If it is, a possible extension has been found. For event 2 the algorithm constructs a set $\{c_2, t_3, c_3\}$. This cannot be a co-set as a write type possible extension has to have all the shared variable conditions in the postset of event 4 in its preset. Finally, for event 3 it is checked if $t_4$ and $c_2$ are concurrent. If they are, a read type possible extensions has been found.

### 4.6 Possible Extensions from Lock Conditions

Possible extensions from a lock condition $l$ are computed when an unlock event is added to the unfolding. This can be done by locating all thread conditions that are concurrent with $l$ and represent a state where a thread wants to acquire the lock represented by $l$. We again limit the discussion to cases where the thread conditions belong to different threads than the unlock event as the same reasoning as with shared variable conditions holds here as well.
The search for possible extensions can be implemented using the same basic idea as in the shared variable condition case. That is, the lock condition $l'$ adjacent to $l$ acts as the condition $c'$ through which thread conditions concurrent with $l$ can be found. There are, however, two key differences to the shared variable condition case that need to be taken into account. First, between adjacent lock conditions there can be multiple events in addition to the lock and unlock ones. Second, it is possible that no event or known possible extensions from $t$ exists when the possible extensions from $l$ are computed.

Due to the first difference we need to extend the search algorithm presented in the previous section such that in step 1 events are collected through conditions that are adjacent or alternative to $l'$. Due to the second difference we need to maintain a list of thread conditions that have no events in their postsets and no known possible extensions. The search algorithm then needs to go through this list as well. In typical programs the size of such a list remains small as a thread condition has an event in its postset unless every test execution that has reached it has ended up in a deadlock. The complete steps to find possible extensions from a lock condition are therefore the following:

1. Collect all events in the postsets of lock conditions that are either adjacent or alternative to $l$.
2. For each of the events check if the thread condition in the preset of it is concurrent with $l$. If it is, add the thread condition to a set of collected thread conditions.
3. For each thread condition in the list of thread conditions that have no events in their postsets and no known possible extensions, check if the thread condition corresponds to a state where it wants to acquire the lock represented by $l$ and if the thread condition is concurrent with $l$. If both of these conditions are satisfied, add the thread condition to the set of collected thread conditions.
4. For each $t$ in the set of collected thread conditions, add a lock type possible extension with $t$ and $l$ in its preset.

**Example 18.** Let us consider the net in Fig. 4.4. Assume that we want to compute possible extensions from the lock condition $l_2$. The lock condition
$l_1$ is adjacent to $l_2$ and $l_3$ is alternative to $l_2$. This means that the possible extensions algorithm collects the events 6 and 7 in the postsets of these conditions. After this the algorithm checks whether the thread conditions $t_2$ and $t_9$ in the presets of the collected events are concurrent with $l_2$.

Let us finally show that the algorithm above finds all possible extensions. Let us first assume that $t$ is concurrent with $l$ and has an event or a known possible extension in its postset. There are four possibilities: $t \text{co} l'$, $t < l'$, $l' < t$ or $t \# l'$. The cases $t < l'$ and $t \# l'$ are impossible which can be shown with the same reasoning as in the proof of Lemma 5. If $t \text{co} l'$, then the algorithm above finds $t$ and adds a possible extension. If $l' < t$, then $t$ must be concurrent with some lock condition that is alternative to $l$. This is because in this case for $t$ and $c$ to be concurrent, they both must be causally preceded by the same lock event in the postset of $l'$. Furthermore, if $t$ is in conflict with all alternative lock conditions, it cannot have any known events in its postset, which contradicts our assumption. Also if $t$ is causally related to an alternative lock condition, then $t$ and $l$ cannot be concurrent. This means the algorithm above again finds $t$ and constructs a possible extension. If $t$ does not have an event or a possible extension in its postset, it must be in the list that the algorithm processes during step 3.

4.7 Checking For Concurrency

A part of the possible extensions computation is to perform checks to determine if a set of conditions are concurrent. This co-check can be done by collecting the events and conditions that causally precede the conditions for which the co-check is performed. If two events are found that have the same condition in their presets, the conditions are in conflict. If one of the conditions is in the set of causally preceding conditions of another condition, the conditions are causally related. Otherwise the set of conditions are concurrent. Such check is in the worst case linear to the size of the unfolding but is often significantly faster.
The unfolding based testing algorithm presented in Chapter 3 can cover the reachable symbolic local states of multithreaded programs sometimes with less test executions than there are Mazurkiewicz traces. Some of the tests performed by the algorithm, however, can still be redundant. One way to further reduce the number of needed test executions is to construct an even more succinct unfolding that can be covered with less test executions. In this chapter we describe such an approach that is based on using contextual net unfoldings instead of regular Petri net unfoldings.

5.1 Motivation

To illustrate a case where the testing algorithm from Chapter 3 can be further improved, let us consider the program shown in Fig. 5.1. The unfolding based testing algorithm needs four test executions to cover the unfolding of the program. However, to cover the reachable local states of threads, only two test executions are needed. This is because threads 1 and 3 can read only two possible values (i.e., $X = 0$ and $X = 5$) and both of these possibilities can be covered by two executions where the first execution performs both of the reads before the write and the second execution performs the write before the reads. Note that the final local state of thread 2 is the same regardless of how the other threads are executed.

Note also that the replication of shared variable conditions for each thread can result in complex unfoldings even for simple programs as illustrated in Fig. 5.1. This indicates that perhaps it is possible to express the happens before relation between operations that are modeled as events more succinctly. In the following we describe how contextual nets can be used to achieve this.
Global variables: Thread 1: Thread 2: Thread 3:
\[X = 0; \quad b = X; \quad X = 5; \quad c = x;\]

**Figure 5.1.** An example program and its unfolding generated by the algorithm in Chapter 3

## 5.2 Contextual Nets

In this section we define contextual nets that extend regular Petri nets with an additional concept called read arcs. In the literature contextual nets are also sometimes called as nets with test arcs, read arcs or activator arcs.

**Definition 15.** A contextual net (c-net) is a tuple \((P, T, F, C, M_0)\), where \(P\) and \(T\) are disjoint sets of places and transitions, respectively, \(F \subseteq (P \times T) \cup (T \times P)\) is a flow relation, and \(C \subseteq P \times T\) is a context relation. Places and transitions are called nodes, elements of \(F\) are called arcs and elements of \(C\) are called read arcs. The preset of a node \(x\), denoted by \(\bullet x\), is the set \(\{y \mid (y, x) \in F\}\). The postset of a node \(x\), denoted by \(x^\bullet\), is the set \(\{y \mid (x, y) \in F\}\). The context of a transition \(x\), denoted by \(x_C\), is the set \(\{y \mid (y, x) \in C\}\). A marking of a net is a mapping \(P \rightarrow \mathbb{N}\) and \(M_0\) is the initial marking.

As with regular nets, markings of contextual nets are represented by putting tokens on circles that represent the places of a net. Read arcs are represented with undirected lines. In this thesis we consider only c-nets where the same transition cannot have the same place in its preset and context. In such c-nets a transition \(t\) is enabled in a marking \(m\) if for each place \(p\) in the preset or context of \(t\) it holds that \(m(p) > 0\). A transition
with a read arc therefore requires that the place connected to the read arc
has a token in it before it can be fired. Like regular nets, firing an enabled
transition $t$ leads to a new marking where tokens are removed from the
places in the preset of $t$ and added to the places in the postset of $t$. Note
that the presence of read arcs do not affect the resulting marking. A c-net
is $n$-bounded if any reachable marking contains at most $n$ tokens and safe
if it is 1-bounded. Figure 5.2 shows an example of a contextual net. To
fire the transition $e_1$, there has to be a token in both in $c_2$ and $c_3$. Firing
$e_1$, however, does not consume the token from $c_3$.

**Definition 16.** The causality relation $<$ in a safe contextual net is the
transitive closure of $F \cup \{(t, t') \in T \times T \mid t^* \cap t' \neq \emptyset\}$. A set of causes of a
node $x$ is defined as $[x] = \{t \in T \mid t < x \lor t = x\}$.

**Definition 17.** Let $t$ and $t'$ be transitions in a safe c-net. The transitions
are in asymmetric conflict, denoted by $t \not\succ t'$, iff (i) $t < t'$, or (ii) $t^* \cap t' \neq \emptyset$,
or (iii) $t \neq t' \land t^* \cap t' \neq \emptyset$.

**Example 19.** Let us consider the c-net in Fig. 5.2. The places $c_2$ and $c_5$ are
causally related. The set of causes of $e_2$ is the set $\{e_1, e_2\}$. For transitions
$e_1$ and $e_4$ it holds that $e_1 \not\succ e_4$ as $e_1^* \cap e_4^* = \{c_3\}$.

**Definition 18 (Adapted from [BCKS08]).** An occurrence c-net is a safe
acyclic c-net such that: (1) for every place $s$, $|s| \leq 1$, (2) the causal relation
is irreflexive and its reflexive closure $\preceq$ is a partial order such that $[t]$ is
finite for any $t \in T$, (3) the initial marking is the set of $\preceq$-minimal places,
and (4) $\not\succ [t]$ is acyclic for every $t \in T$.

Like regular nets, contextual nets can be unfolded into occurrence c-
nets. As before, we call the transitions and places in an occurrence c-net
as events and conditions, respectively. The c-net in Fig 5.2 is also an
occurrence c-net.

**Definition 19.** Nodes $x$ and $y$ in an occurrence c-net are concurrent, de-
tonated by $x$ co $y$, if (1) $x$ and $y$ are not causally related, and (2) the set of
causes $A = [x] \cup [y]$ is finite and $\not\succ A$ is acyclic (i.e., there is no cycle of
asymmetric conflicts in the set $A$).

**Example 20.** Let us consider the occurrence c-net in Fig. 5.2. The conditions
$c_8$ and $c_{11}$ are in conflict. This is because to reach a marking with
both of these conditions, all the events in the c-net need to be fired (i.e.,
$A = \{e_1, e_2, e_3, e_4, e_5\}$). In addition to this there exists a cycle $e_5 \not\succ e_2 \not\succ$
Figure 5.2. An occurrence e-net

$e_3 \not\rightarrow e_4 \not\rightarrow e_5$. Intuitively $e \not\rightarrow e'$ means that if both $e$ and $e'$ are to be fired, $e$ needs to be fired first. In our example the event $e_3$ needs to be fired before $e_4$ because firing $e_4$ removes a token from $c_3$ and therefore disables $e_3$. The events $e_5$ and $e_2$ are restricted in a similar way.

5.3 Modeling Test Executions with Contextual Unfoldings

To model test executions with contextual unfoldings, we can again assume that there exists an implicit contextual net (i.e., a model of the program under test) that is being unfolded. As before, the infeasible execution paths can be eliminated by using path constraints. The modeling constructs for building a contextual net model are shown in Fig. 5.3. The only difference in these constructs compared to the ones used in Chapter 3 is how read and write statements are modeled. The shared variable places are no longer replicated for each thread. Recall that the reason for the place replication in Chapter 3 is to make two concurrently enabled read operations independent in the unfolding. With contextual nets this can be achieved by using read arcs. If we model two concurrently enabled reads with the constructs in Fig. 5.3, the unfolding of the program contains two read events that both have a read arc to the same initial shared variable condition. Firing one of these events does not disable the other and therefore the events model both ways to interleave the reads.

To illustrate the new constructs, let us model a test execution of the program in Fig. 5.1. The modeling process is done the same way as in Chap-
Figure 5.3. Modeling constructs adapted to contextual unfoldings

First, the conditions corresponding to the initial state are added to the unfolding. This is shown as the topmost net in Fig. 5.4. Assume that we want to model a test execution that executes threads 1, 2, and 3 in that order. The read operation performed by thread 1 is modeled as event 1. Note that firing event 1 does not remove the token from the initial shared variable condition for $X$ as the value of the shared variable does not change. The write performed by thread 2, however, can change the value and therefore it is modeled by event 2 that creates a new shared variable condition modeling the value of the shared variable after the write. Thread 3 then reads this value and therefore it is modeled by an event that has a read arc to the new shared variable condition.

The net in the middle of Fig. 5.4 shows the unfolding after the first test execution.

The unfolding can be completed by performing additional test executions and the complete unfolding is shown also in Fig. 5.4. Note that the final unfolding is significantly smaller, namely has less events and conditions than the regular unfolding shown in Fig. 5.1, but it still represents all executions of the program. As discussed earlier, the unfolding algorithm from Chapter 3 requires four test executions to cover the unfolding. The unfolding in Fig. 5.4, however, can be constructed by modeling only two test executions (e.g., the ones discussed in Sect. 5.1). Note also that if the number of thread that read the shared variable $x$ are increased, the number of test executions needed by the algorithm in Chapter 3 (and the number of write events in the unfolding) grows exponentially. With contextual unfoldings such exponential blow up is avoided as only one write event is enough regardless of the number of reads. Contextual nets there-
fore not only make unfoldings smaller but also allow the reachable local states of threads to be covered with a smaller number of test executions.

5.4 Systematic Testing with Contextual Unfoldings

Contextual unfoldings of programs can be constructed systematically in a similar way as place replication based unfoldings. The new unfolding based testing algorithm that has been adapted to contextual unfoldings is shown in Fig. 5.5. Note that at a general level it operates exactly in the same way as the algorithm presented in Fig. 3.11. However, as test executions are modeled differently, some changes are needed to the subroutines that compute possible extensions and inputs for test executions. As the high level functionality of the testing algorithm remains the same, we do not replicate the discussion of the algorithm in Fig. 3.11 here. Instead we highlight through an example the differences to the approach in Chapter 3 that need to be taken into account. We will then describe the necessary changes to the subroutines CONTEXTUALEXECUTE and CONTEXTUALPOSSIBLEEXTENSIONS in the subsequent sections.

Example 21. Figure 5.6 shows a program with two threads and its con-
Input: A program $P$

1: $unf := \text{initial contextual unfolding}$

2: $\text{extensions := INITIALYENABLEEVENTS}(P)$

3: $\textbf{while} \text{extensions} \neq \emptyset \textbf{ do}$

4: $\textbf{choose target} \in \text{extensions}$

5: $\textbf{if} \text{target} \notin unf \textbf{ then}$

6: $\text{operation\_sequence := CONTEXTUALEXECUTE}(P, \text{target}, k)$

7: $M = \text{initial marking}$

8: $\textbf{let} o_1, o_2, ..., o_n = \text{operation\_sequence}$

9: $\textbf{for} i = 1 \textbf{ to } n \textbf{ do}$

10: $e = \text{CORRESPONDINGEVENT}(o_i, M)$

11: $\textbf{if} e \notin unf \textbf{ then}$

12: $\text{add } e \text{ and its output conditions to } unf$

13: $\text{extensions := extensions \setminus \{e\}}$

14: $\text{pe := CONTEXTUALPOSSIBLEEXTENSIONS}(e, unf)$

15: $\text{extensions := extensions \cup pe}$

16: $M = \text{FIREEVENT}(e, M)$

**Figure 5.5.** Contextual unfolding based testing algorithm

textual unfolding. Let us assume that we have only performed two test executions: one that corresponds to an event sequence (1, 2, 3) and another that corresponds to a sequence (4, 5, 6, 2, 3). After these test executions there are two possible extensions that correspond to events 7 and 8. Assume that the testing algorithm wants to cover the event 7. To reach this event, the approach presented in Chapter 3 would construct a thread schedule that corresponds to firing events that causally precede the event 7 in the order they were added to the unfolding. In other words, it would try to follow an event sequence (2, 4, 5, 7). However, in the contextual unfolding in Fig. 5.6 the events cannot be fired in that order. This is because if event 2 is fired, event 5 will never get enabled. It is, however, possible to fire the events in the order (4, 5, 2, 7). The read arcs can therefore be seen as generating additional scheduling constraints. More specifically, if a read event and a write event have the same shared variable condition in their context and preset, respectively, the read event must always be fired first in an execution where both of them are fired.

The scheduling constraints resulting from read arcs also affect possible extensions computation, or more specifically, how it is checked whether given conditions are concurrent. It is possible that the conditions are in
Global variables: Thread 1: Thread 2:
X = 0;
1: X = 5;
2: a = X;
3: b = input();
4: if (b == 0)
5: c = X;
6: d = X;

Figure 5.6. A program and its contextual unfolding illustrating the systematic testing algorithm.

canfict because the scheduling constraints form a cycle. Consider the occurrence net in Fig. 5.2. The net can also be seen as a partial unfolding of a program with two threads that perform read and write operations. Performing a co-check on the conditions \(c_8\) and \(c_{11}\) by using the approach presented in Chapter 3 does not work as it does not take contextual cycles into account.

5.5 Computing Input Values and Schedules for Test Executions

A test execution performed by the testing algorithm attempts to extend the unfolding by covering a possible extension event. To do this, the test execution must perform a sequence of operations that corresponds to firing all the causes of the target event. Let \(S\) denote the set of these events. A path constraint for the test execution is simply the conjunction of the symbolic constraints stored to the events in \(S\). If the path constraint is unsatisfiable, it is safe to remove the target event from the set of possible extensions as any execution reaching the target event must fire the events in \(S\). Therefore the same reasoning as in Chapter 3.5 applies here.
as well. However, as discussed in the previous section, the order in which threads must execute their operations cannot be obtained the same way as with regular unfoldings.

Figure 5.7 shows an algorithm that solves the execution order problem for the types of unfoldings generated by the testing algorithm. The algorithm starts by collecting a set of events that need to be fired in order to cover the target event. These events are then sorted into a list $L$ in the order they have been added to the unfolding (line 1). The sorting guarantees that the algorithm does not try to process an event before all its causally preceding events have been processed. After collecting the events, the algorithm attempts to fire the events one by one starting from an initial marking $m$. Each iteration of the loop starting at line 4 searches through the sorted list of collected events in order to find an event that is enabled in the current marking. The first enabled event in the list can be fired if it is not a write event. If it is a write, it can be fired if all the collected read events that have a read arc to a shared variable condition in the preset of the write event have already been fired (line 9). In this way, firing an event $e$ cannot disable any event $e'$ that has not yet been fired unless $\cdot e \cap \cdot e' \neq \emptyset$, in which case the target event is not reachable at all.

When an event is fired, the current marking is updated (line 10) and the event is removed from the set of events that still need to be fired (line 11). If the event under consideration cannot be fired (i.e., it is not enabled or it is a write that needs to wait for read events to be fired), the thread identifier ($tid$) of the event is added to a set of postponed threads. The event that was fired is also added to the end of the schedule being constructed. If all threads that still have events to be fired get postponed, there is a $\nearrow$-cycle with the collected events. This means that the target event is not reachable. In these cases the algorithm returns the value false. The check of how many threads still have events to be fired can be performed, for example, by recording the number of events belonging to different threads while sorting the events. Firing an event then decreases the number of events for a thread. If the algorithm successfully fires all events in $A$, the constructed schedule is returned. A test execution can then be performed by taking control of the runtime scheduler and forcing it to execute the operations in the same order as the corresponding events occur in the schedule returned by the algorithm in Fig. 5.7. This schedule together with the input values obtained from a path constraint force the test execution to follow a path that covers the target event.
**Input:** event $target$

1. $L := \text{SORT}(|target|)$
2. $m := \text{initial marking}$
3. $schedule := \text{empty list of events}$
4. **while** $L$ is not empty **do**
5. \hspace{1em} $postponed := \emptyset$
6. \hspace{1em} **let** $e_1, e_2, ..., e_n = L$
7. \hspace{1em} **for** $i = 1$ to $n$ **do**
8. \hspace{2em} **if** $(postponed \cap \text{tid}(e_i)) = \emptyset \land enabled(e_i, m)$ **then**
9. \hspace{3em} **if** $e_i$ is not a write or $\bullet e_i \cap L = \emptyset$ **then**
10. \hspace{4em} $m := \text{FIRE}(e_i, m)$
11. \hspace{4em} remove $e_i$ from $L$
12. \hspace{4em} add $e_i$ to the end of $schedule$
13. \hspace{2em} **break**
14. \hspace{1em} $postponed := postponed \cup \{\text{tid}(e_i)\}$
15. \hspace{1em} **if** $|postponed| = \text{number of threads in } L$ **then**
16. \hspace{2em} **return** false
17. **return** $schedule$

**Figure 5.7.** Algorithm to compute a schedule to a target event

### 5.6 Computing Possible Extensions

Computing possible extensions can be done efficiently by using the same approach as described in Chapter 4. In fact, as place replication is not used for shared variables, computing possible extensions becomes conceptually simpler. However, due to possible $\nearrow$-cycles, checking if given conditions are concurrent is not as straightforward as before. For completeness we first show that the same connection between possible extensions as discussed in Chapter 4 holds for contextual unfoldings of programs as well. After that we consider how the cases for finding possible extensions from thread, shared variable and lock conditions differ from the approach presented in Chapter 4.

**Structural Connections Between Possible Extensions**

Possible extensions in a contextual unfolding of a program can be computed through adjacent and alternative conditions much in the same way as in place replication based unfoldings. As the shared variable conditions are not replicated in contextual unfoldings, the definition for adjac-
cent shared variable conditions can be simplified.

**Definition 20.** Shared variable conditions $c_1$ and $c_2$ in a contextual unfolding are adjacent if there exists a write event $e$ such that $c_1 \in \bullet e$ and $c_2 \in e^\ast$.

**Theorem 4.** Let $t$ be any condition in a contextual unfolding of a program and $x$ be any shared variable. Let $G$ be a graph such that there is a vertex in $G$ for every shared variable condition that is concurrent with $t$ and labeled with $x$. If an edge is added between those vertices that correspond to adjacent shared variable conditions, then the graph $G$ is connected.

*Proof.* Let $G'$ be a graph that is constructed the same way as $G$ but without the requirement that the shared variable conditions corresponding to vertices are concurrent with $t$. It is easy to see that $G'$ is a tree where any shared variable condition corresponding to a vertex is causally preceded by a shared variable condition represented by the parent vertex.

Let $c_1$ and $c_2$ be any vertices in $G$. As $G'$ is a tree, there is only one path from $c_1$ to $c_2$ in $G'$. For $G$ to be connected, every shared variable condition in this path must be concurrent with $t$. There are two possible cases: (i) a path from the root of $G'$ to $c_1$ is a prefix of a path from the root to $c_2$ (or vice versa), or (ii) the paths from the root of $G'$ to $c_1$ and $c_2$ take different branches at some point.

Let us consider the case (i). It must hold that either $c_1 < c_2$ or $c_2 < c_1$. Let us assume that $c_1 < c_2$ holds (the other case is symmetrical). For $G$ to be not connected, there has to be a condition $c'$ that is not concurrent with $t$ and $c_1 < c' < c_2$. As $c'$ is not concurrent with $t$, it has to be either causally related with it or there has to exist a cycle of asymmetric conflicts in $|c'| \cup |t|$. However, neither of these cases is possible. If $c'$ is causally related to $t$, then either $c_1$ or $c_2$ has to be causally related to $t$, which contradicts the assumption that these conditions are concurrent with $t$. Similarly, if there exists a cycle of asymmetric conflicts, then the same cycle exists also in $|c_2| \cup |t|$. Therefore $c'$ cannot exist.

Let us now consider the case (ii). There must exist a vertex from which the paths to $c_1$ and $c_2$ in $G'$ follow different branches. Let $c'$ be the shared variable condition represented by this vertex. If $c'$ is concurrent with $t$, then by case (i) all the conditions on paths from $c'$ to $c_1$ and $c_2$ are also concurrent with $t$. If $c'$ is not concurrent with $t$, there are three possibilities: $c' < t$, $t < c'$ or $\mathcal{F}[c'| \cup |t|]$ contains a cycle. The first case implies that either $c_1$ or $c_2$ has to be in conflict with $t$ as the computations to a marking

89
with $c_1$ and $c_2$ need to fire different write events that both have $c'$ in their presets. The remaining two cases imply $t < c_1$ and $\mathcal{C}[c_1 \cup t]$ contains a cycle, respectively. Therefore $c'$ must be concurrent with $t$. As in all cases there is a path from $c_1$ to $c_2$ also in $G$, the graph $G$ is connected.

As the modeling constructs for lock acquires and releases are the same as in Chapter 3, the same definitions as before for adjacent and alternative lock conditions can be used.

**Theorem 5.** Let $t$ be a thread condition in a contextual unfolding such that the next operation from $t$ is a lock acquire. Let $S$ be a set of all lock conditions that are concurrent with $t$ and represent the lock that a thread wants to acquire when reaching the state represented by $t$. Let $G$ be a graph such that there is a vertex in $G$ for every lock condition in $S$ and an edge between two vertices if the respective lock conditions are adjacent or alternatives. Then the graph $G$ is connected.

**Proof.** The Theorem 3 presented in the previous chapter states that the same property holds for regular unfoldings. The proof of Theorem 3 is presented in such a way that it holds also for contextual nets. Note that the causality and conflict relations are different (i.e., we have to extend the conflict relation used in the proof to take cycles of asymmetric conflicts into account) for regular unfoldings and contextual unfoldings. However, if in the proof they are used only in a way that holds both for regular and contextual unfoldings.

**Possible Extensions From Thread Conditions**

Based on Theorems 4 and 5, possible extensions from a thread condition $t$ can be obtained by performing a recursive search starting with a shared variable or lock condition that is known to be concurrent with $t$. This search can be implemented in the same way as discussed in Chapter 4. Also it is easy to see that the condition acting as the starting point for the search can be obtained the same way as described in Chapter 4. The main difference to using regular unfoldings is that each possible extension has at most one shared variable condition in its preset (instead of a set). This somewhat simplifies the possible extensions computation.
Possible Extensions From Shared Variable Conditions

Possible extensions from a shared variable condition $c$ are computed only when a write event $e$ gets added to the contextual unfolding. This is because adding a read event does not increase the number of shared variable conditions in the unfolding. The same reasoning as in Chapter 4 can be used to show that the shared variable condition $c' \in \cdot e$ must be concurrent with any thread condition $t$ that can form a preset and a context of a possible extension event together with $c$. This leads to similar search steps as in Chapter 4:

1. Collect all events, including previously known possible extensions, that are in the postset or context of the shared variable condition that is adjacent to $c$ (i.e., the shared variable condition in the preset of the write event $e$).

2. For each collected event construct a set consisting of $c$ and the thread condition in the preset of the event.

3. If the formed set is a co-set, add an event to the set of possible extensions. This event is a write event with the co-set as its preset if the next operation from $t$ is a write and a read event with $t$ in the preset and $c$ in the context otherwise.

The main differences to the steps in Chapter 4 are that in step 1 the read arcs need to be taken into account and step 2 is simplified as the constructed sets have always only two conditions.

Possible Extensions From Lock Conditions

The approach described in Chapter 4 to compute possible extensions from lock conditions can be applied for contextual unfoldings of programs without modifications. This is because locking and unlocking is modeled in the same way in both types of unfoldings.

Checking For Concurrency

The possible extensions computation needs to be able to determine if two conditions $c_1$ and $c_2$, that can form a preset of an event $e_p$, are concurrent. It holds that the conditions are concurrent if there is a reachable marking
where both of the conditions contain a token. In such a marking the event $e_p$ is also enabled. Therefore, the check can be performed by using the algorithm in Fig. 5.7 to determine if there is a computation that starts from the initial marking and eventually fires the event $e_p$. If the algorithm returns false, the conditions are not concurrent. As the co-check is done for all possible extensions, the schedule computed by the algorithm in Fig. 5.7 can be stored to these events so that the EXECUTE subroutine does not need to compute the obtained schedule again.

**Complexity of Possible Extension Computation**

In a contextual unfolding of a multithreaded program the combined size of the preset and the context of any event can be at most two conditions. Furthermore, when possible extensions are computed, one condition in the preset or context of the extensions is already known as discussed earlier. This means that a possible extensions algorithm needs in the worst case consider $O(n)$ conditions, where $n$ is the number of conditions in the unfolding. This is potentially a huge improvement over normal unfoldings, where $O(n^k)$ conditions, where $k$ is the number of threads, need to be considered. Thus computing possible extensions becomes polynomial instead of NP-hard [EH08] even in the worst case.

**5.7 Handling Dynamic Thread Creation**

One additional benefit of using contextual unfoldings is that dynamic thread creation can be modeled easily. As discussed in Chapter 3, the main problem in handling thread creation with regular unfoldings is that new conditions for shared variables need to be created when threads are created. With contextual unfolding there is no need for this, as there are no local copies of the shared variable conditions for each thread. This means that it is possible to model thread creation simply with an event that has a thread condition of the parent thread in its preset and two thread conditions in its postset: one for the original thread and one for the new thread.

**Example 22.** As an example let us consider a program that initially contains two threads such that thread 1 first reads the value from a shared variable $X$ after which it creates a new thread before proceeding with its own computation. Thread 2 simply writes a value to $X$ and the thread cre-
ated by thread 1 reads the value of $X$. The contextual unfolding of such a program is shown in Fig. 5.8. Note that the third thread does not have a single initial condition. Instead the starting point of the created thread depends on which local state the parent thread is when the new thread is created. In our example there are therefore two possible starting points for the third thread (marked with a darker color in Fig. 5.8). This way the unfolding represents the causality between events of the new thread and those that must be fired before the thread is created. Furthermore, if the creating thread passes some arguments to the new thread that affects its behavior, this way of modeling handles such cases correctly as well.

### 5.8 Correctness

We will next show that the testing algorithm in Fig. 5.5 explores all feasible control flow paths of individual threads in a multithreaded program. We follow the same proof strategy as in Chapter 3. For place replication based unfoldings the causality relation between events matches with the happens before relations between operations. With contextual unfoldings of programs this, however, is no longer the case. For example, in Fig. 5.6 the events 2 and 5 are not causally related but the write and read operations corresponding to these events are shared memory access precedence related in any execution where both of the operations are performed.

Intuitively, the only case where the causality relation in a contextual
unfolding of a program differs from the $\rightarrow_{ex}$ relation is when in $ex$ there is a read operation of thread $t$ and a write operation of thread $t'$ such that after the read, $t$ does not perform any operations that affect the execution of $t'$ (either directly or through some other thread) until $t'$ has performed the write operation. From the point of view of thread $t'$, there is no difference whether $t$ has performed the read or not (at least not before the write operation has been performed). Therefore, if we are interested only in the local state of thread $t'$ right after the write operation, the read of thread $t$ does not necessarily need to happen before the write. As the only operations that can affect the execution of other threads are either writes, locks or unlocks, we can formalize this with the following modified happens before relation.

**Definition 21.** Let $o_i$ and $o_j$ be operations in an execution $ex \in EX(P)$. We say that $o_i$ contextually happens before $o_j$ in $ex$, denoted as $o_i \xrightarrow{ctx}_{ex} o_j$, if and only if at least one of the following conditions is satisfied:

- $o_i$ and $o_j$ belong to the same thread, or
- $o_i \rightarrow_{ex} o_j$ and $o_i$ is not a read operation, or
- $o_i \rightarrow_{ex} o_j$ and there exist an event $o_k \in ex$ such that $o_i <^seq_{ex} o_k$, $o_k \rightarrow_{ex} o_j$, and $o_k$ is either a write, lock or unlock operation.

We can now show that the causality relation between events in a contextual unfolding of a program matches with the contextual happens before relation.

**Lemma 6.** Let $o_i$ and $o_j$ be operations in an execution $ex \in EX(P)$ and let $e_i$ and $e_j$ be events that model $o_i$ and $o_j$, respectively, in a contextual unfolding of $P$. It holds that $e_i \preceq e_j$ if and only if $o_i \xrightarrow{ctx}_{ex} o_j$.

**Proof.** ($\Leftarrow$): Let us first show that if $o_i \xrightarrow{ctx}_{ex} o_j$, then $e_i \preceq e_j$. If $i = j$, then trivially $e_i = e_j$. Given that $o_i \xrightarrow{ctx}_{ex} o_j$ holds, we can therefore in the following assume that $i < j$. Let $o_k$ be the operation in $ex$ for which it holds that $k$ is the largest index such that $k \neq j$ and $o_k \xrightarrow{ctx}_{ex} o_j$. There are now four possible cases: (i) $o_i <^seq_{ex} o_j$ and $k = i$, (ii) $o_i <^mem_{ex} o_j$, $o_i$ is not a read operation and $k = i$, (iii) $o_k <^seq_{ex} o_j$ and $k \neq i$, or (iv) $o_k <^mem_{ex} o_j$, $o_k$ is not a read operation and $k \neq i$. Note that in cases (ii) and (iv) the operations $o_i$ and $o_k$ cannot be read operations because this would imply that $o_i \xrightarrow{ctx}_{ex} o_j$ and $o_k \xrightarrow{ctx}_{ex} o_j$ do not hold.

1It can also be shown that $o_i \rightarrow_{ex} o_j$ if and only if $e_i \succ e_j$. 

94
Let us consider the case (i). If \( o_i <_{\text{seq}} o_j \), the operations must be performed by the same thread or the thread of \( o_j \) has been created by the thread of \( o_i \) after the operation \( o_i \). From the modelling constructs it is easy to see that there must exists a path of arcs from the event \( e_i \) to the event \( e_j \) through a sequence of thread conditions. Therefore if \( o_i <_{\text{seq}} o_j \), then \( e_i \leq e_j \).

Let us now consider the case (ii). It is easy to see that the operation pair \((o_i, o_j)\) must correspond to one of the following cases: (write, write), (write, read), (lock, unlock) or (unlock, lock). That is, (read, write) is not possible. In each of these cases the operations are modeled by events such that there is a common condition in \( (e_i)^* \) and in \( *(e_j) \). This immediately implies that \( e_i \leq e_j \).

Let us finally consider the cases (iii) and (iv). Let \( e_k \) be the event that models the operation \( o_k \). From cases (i) and (ii) it follows that \( e_k \leq e_j \) holds in cases (iii) and (iv), respectively. This means that \( e_i \leq e_j \) holds if we can show that \( e_i \leq e_k \) holds. This can be done by recursively analyzing the case \( o_i \xrightarrow{\text{ctx}}_{\text{ex}} o_k \) in the same way as above. The next recursion step either ends in case (i) or (ii) or has to show that \( o_i \xrightarrow{\text{ctx}}_{\text{ex}} o_l \) holds, where \( l \geq i \) and \( l < k < j \). Therefore there can be only a finite number of recursion steps before the recursions ends either in case (i) or (ii).

(\( \Rightarrow \)) : Let us now show the other direction. It is again trivial to see that if \( e_i = e_j \) then the only possibility is that \( i = j \) in which case \( o_i \xrightarrow{\text{ctx}}_{\text{ex}} o_j \) holds. If \( e_i < e_j \), then directly from the definition of causality between events it follows that there must exist a sequence of events from \( e_i \) to \( e_j \) such that for every adjacent events \( e \) and \( e' \) in the sequence there is a common condition in the postset of \( e \) and either in the preset or context of \( e' \). If the common condition is a thread condition, then the operations \( o \) and \( o' \) modeled by \( e \) and \( e' \), respectively, are sequentially related. If the common condition is either a shared variable or a lock condition, then the corresponding operations must access the same shared memory location. This means that the operations are shared memory access precedence related which implies that \( o \xrightarrow{\text{ctx}}_{\text{ex}} o' \) unless \( o \) is a read operation. In the case that \( o \) is a read operation, the only condition in the postset of \( e \) is a thread condition. This implies that \( o \) and \( o' \) are sequentially related which implies \( o \xrightarrow{\text{ctx}}_{\text{ex}} o' \). As for every adjacent events \( e \) and \( e' \) in the sequence and their corresponding operations \( o \) and \( o' \) it holds that \( o \xrightarrow{\text{ctx}}_{\text{ex}} o' \), it follows that \( o_i \xrightarrow{\text{ctx}}_{\text{ex}} o_j \).

**Lemma 7.** Let \( ex \) and \( ex' \) be two executions in \( EX(P) \). If the last operation
of thread $t$ is modeled by the same event $e$ in the contextual unfolding of $P$ in both executions, then $l_t(ex) \sim l_t(ex')$.

**Proof.** Let $o$ and $o'$ be the last operations of $t$ in $ex$ and $ex'$, respectively. Let us consider an execution $ex_1$ that is otherwise the same as $ex$ except that it performs an operation $o_p$ if and only if $o_p \xrightarrow{ctx\_ex} o$. Let $ex'_1$ be an execution constructed similarly from $ex'$ and $o'$. It is easy to see that $l_t(ex) = l_t(ex_1)$ and $l_t(ex') = l_t(ex'_1)$.

As both $o$ and $o'$ are modeled by the event $e$, from Lemma 6 it follows that $ex_1$ and $ex'_1$ are both linearizations of the $ctx\_ex_1$ partial order and perform the same operations but possibly in different orders. It also holds that it is possible to transform the sequence $ex_1$ into $ex'_1$ by performing a finite number of swaps between adjacent independent operations. This is because if some dependent operations are performed in different orders in the executions, then $o$ and $o'$ would not be modeled by the same event. For pairs of dependent operations except (read, write) this is easy to see. For a read operation $o_r$ and a write operation $o_w$ it does not necessarily hold that $o_r \xrightarrow{ctx\_ex_1} o_w$. However, if the operations are swapped, then the read operation in the modified execution would be modeled by different event than in the original and both of the read events cannot causally precede $e$. As swapping adjacent independent operations do not affect the local states of threads in the executions, it is easy to see that $l_t(ex_1) \sim l_t(ex'_1)$ which implies $l_t(ex) \sim l_t(ex')$.

The following lemma and theorems can be proven with the same reasoning as the in the proofs of their counterparts in Chapter 3. This is because Lemma 7 shows that the same property (i.e., Lemma 2) that is needed in the proofs holds also for contextual unfoldings of programs.

**Lemma 8.** For every execution $ex \in EX(P)$ there is a set $S$ of events in the contextual unfolding of $P$ that represents $ex$.

**Theorem 6.** The algorithm in Fig. 5.5 explores a control flow path of a thread in a multithreaded program $P$ if and only if the path is feasible.

**Corollary 2.** If an error statement is reachable in a multithreaded program $P$, the algorithm in Fig. 5.5 performs a test execution that leads to the error statement.
6. Testing Global State Reachability Properties

The unfolding based algorithms discussed in the previous chapters cover all feasible control flow paths of threads. This means that the algorithms preserve, for example, all assertion violations on local variables. As the generated unfoldings represent all interleavings of execution steps, it is also possible to use unfoldings to determine if specific global states are reachable. As an example, it is possible to determine from an unfolding if a program has a deadlock. It is also possible to check for other types of properties given that the testing algorithm collects additional information while constructing the unfolding. Such properties can be global invariants requiring, for example, the sum of some shared values to be less than a specified limit at all times, or assertions on invariants over global states that must hold at specific points in the program’s execution. To detect violations of global state reachability based properties, it is possible to perform an additional search to explore the reachable global markings that have not been explicitly covered by any of the test executions. In this chapter we show how such a search can be done by translating a global state reachability problem into a SMT instance and also how deadlocks can be searched directly from unfoldings.

6.1 Motivating Example

As an unfolding of a program represents all executions, it can be used to determine that some global states in the program are reachable even if those states are not explicitly covered by any of the test executions used to construct the unfolding. As an example, let us consider the program shown in Fig. 6.1. If we perform a test execution where the variable \( a \) gets an input value 20 and \( b \) gets a value 0, the global states with valuations \((X = 0, Y = 0)\) and \((X = 20, Y = 0)\) get observed. Similarly in an
Global variables: Thread 1: Thread 2:

\[ X = 0; \quad 1: a = \text{input}(); \quad 4: b = \text{input}(); \]
\[ Y = 0; \quad 2: \text{if} (a > 10) \quad 5: \text{if} (b > 10) \]
\[ 3: \quad X = a; \quad 6: \quad Y = b; \]

Figure 6.1. A simple program and its unfolding for demonstrating global state reachability

execution where \(a\) gets a value 0 and \(b\) gets a value 20, the observed global states are \((X = 0, Y = 0)\) and \((X = 0, Y = 20)\).

Let us assume that we are interested in determining if a global state satisfying a constraint \(X > 40 \land Y = 15\) is reachable. Analysis of the two concrete test executions above does not find such a state. If the test executions are executed symbolically and modeled as an unfolding, we get the unfolding shown in Fig. 6.1. Note that the concrete or symbolic values of the variables are shown in brackets next to the corresponding shared variable conditions. From this unfolding it is easy to see that a marking where both write(X) and write(Y) events have been fired is reachable even though neither of the test executions covered the marking. The valuation of shared variables in the global state represented by the marking can be expressed by a constraint \(X = \text{input}_1 \land \text{input}_1 > 10\land Y = \text{input}_2 \land \text{input}_2 > 10\). The conjunction of this constraint and the constraint \(X > 40 \land Y = 15\) is satisfiable. This means that the global state we are interested in is reachable. Assuming that the unfolding is complete, it is therefore possible to determine whether a symbolic global state is reachable even if none of the performed test executions observe that state.
6.2 SMT Translation Based Reachability Checking

Satisfiability modulo theories (SMT) is a generalization of boolean satisfiability (SAT). A SMT instance can be seen as a first order logic formula where the interpretation of symbols is constrained by a background theory. Examples of such theories are linear arithmetic, bit-vector and array theories. The theory of linear arithmetic, for example, gives a meaning to symbols such as + and -. One way to determine if a global state is reachable in an unfolding of a program is to translate the unfolding into SMT formulas and use a SMT-solver as a search engine. In this section we show how such a translation can be constructed and used to check various global properties.

Base Translation For Regular Unfoldings

The reachable markings of an unfolding correspond to a set of symbolic reachable global states of the program under test. By encoding the reachable markings of an unfolding as a SMT formula, we get a translation that is satisfied only for the reachable global states of the system. One possible way to do such an encoding is the conjunction of the following formulas:

For each event \( e \):

\[
e \Rightarrow \bigwedge_{e_i \in \mathcal{E}(e)} e_i
\]  
(B1)

For each branching event \( e \) with a symbolic constraint \( g \):

\[
e \Rightarrow g
\]  
(B2)

For each condition \( c \) and each event \( e \in c \) :

\[
e \Rightarrow \bigwedge_{e_i \in \mathcal{E}(e) \setminus \{e\}} \neg e_i
\]  
(B3)

For each condition \( c \) and event \( e \in c \) :

\[
c \iff e \land \neg \left( \bigvee_{e_i \in e} e_i \right)
\]  
(B4)

The encoding above is similar as the one presented in [EH08] with the exception that there are additional formulas of type (B2) that capture the path constraints. In the presented encoding there is a boolean variable for each event and each condition in the unfolding. A variable corresponding to a condition is true if and only if the condition is marked in the reached marking. A variable corresponding to an event is true if and only if the
Testing Global State Reachability Properties

Figure 6.2. An example of a SMT translation

\[ (B1) \]
\[ e_5 \implies e_2 \quad e_1 \implies input_1 \leq 10 \quad e_6 \implies e_2 \quad e_1 \implies input_1 > 10 \quad e_6 \implies e_3 \]
\[ e_4 \implies e_5 \]

\[ (B2) \]
\[ (B3) \]
\[ e_5 \implies \neg e_2 \quad e_1 \implies \neg e_2 \quad e_5 \implies \neg e_3 \]
\[ e_6 \implies \neg e_1 \quad e_3 \implies \neg e_5 \]
\[ e_6 \implies \neg e_5 \quad e_4 \implies \neg e_3 \]

\[ (B4) \]
\[ c_1 \iff \neg(e_1 \lor e_2) \quad c_6 \iff e_2 \land \neg(e_5 \lor e_6) \quad c_{11} \iff e_4 \]
\[ c_2 \iff \neg(e_5 \lor e_3) \quad c_7 \iff e_3 \land \neg e_6 \quad c_{12} \iff e_4 \]
\[ c_3 \iff \neg(e_3 \lor e_4) \quad c_8 \iff e_3 \quad c_{13} \iff e_6 \]
\[ c_4 \iff \neg(e_3 \lor e_4) \quad c_9 \iff e_3 \quad c_{14} \iff e_5 \]
\[ c_5 \iff e_1 \quad c_{10} \iff e_4 \quad c_{15} \iff e_6 \]
\[ c_{16} \iff e_5 \land \neg e_4 \]

event has been fired in order to reach the marking denoted by the conditions. The formulas of type (B1) encode the fact that if an event \( e \) has been fired, then all the events that causally precede \( e \) must have been fired as well. Formulas of type (B2) require that the path constraints generated by symbolic execution must be satisfiable for the reachable marking. Formulas of type (B3) encode the fact that if a set \( S \) of events have a common condition in their presets, then at most one of them can be fired. The size required by formulas of type (B3) grows quadratically to the number of events in the set \( S \) as such a formula is needed for each event in \( S \). It is possible to optimize this with a linear encoding. Our implementation uses an encoding similar to the one presented in [LBHJ04] to achieve this. Finally, formulas of type (B4) denote that a condition \( c \) is marked if and only if the event that places a token in \( c \) has been fired but none of the events that take the token from \( c \) has been fired. Figure 6.2 shows an example of
Testing Global State Reachability Properties

Base Translation For Contextual Unfoldings

To encode the reachable markings of a contextual unfolding, the same encoding as above can be used to model the arcs, events, and symbolic constraints. As contextual nets also contain read arcs, they need to be also taken into account. This can be done with formulas of the following type.

For each read type event $e$ and $e' \in \cdot e$:

$$e \Rightarrow e' \quad \text{(C1)}$$

Intuitively (C1) means that if a read event $e$ is fired, the write event $e'$ that has most recently updated the value being read must have been fired also. A translation consisting of formulas (B1)-(B4) and (C1) is an overapproximation of reachable markings. This is because the translation does not take possible $\Rightarrow$-cycles into account. For example, recall the contextual unfolding in Fig. 5.2 and that the conditions $c_8$ and $c_{11}$ are not concurrent because there exist a cycle $e_5 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5$. To accurately capture the reachable markings of a contextual unfolding, additional constraints are needed to make the translation unsatisfiable if a marking implies a $\Rightarrow$-cycle. To complete the translation, let $n_i$ be a natural number associated with event $e_i$. Intuitively the numbers associated with events describe the order in which they are fired (i.e., event with a number 4 must be fired before an event with a number 7). The firing order that eliminates $\Rightarrow$-cycles can then be expressed with the following formulas.

For each event $e_i$ and each event $e_j \in \cdot(\cdot e_i) \cup \cdot e_i$:

$$e_i \Rightarrow n_j < n_i \quad \text{(C2)}$$

For each read event $e_i$ and write event $e_j$ such that $\cdot e_j \cap e_i \neq \emptyset$:

$$e_i \Rightarrow n_i < n_j \quad \text{(C3)}$$

The formulas of type (C2) enforce that if the event $e_j$ causally precedes the event $e_i$, then $e_j$ needs to be fired before $e_i$ (i.e., it has a lower execution turn number). The intuition behind formulas of type (C3) is that if a read event and write event try to access the same shared variable place,
Testing Global State Reachability Properties

(B1)
\[ e_2 \Rightarrow e_1 \]
\[ e_3 \Rightarrow e_2 \]
\[ e_5 \Rightarrow e_4 \]

(B4)
\[ c_1 \Leftrightarrow \neg e_2 \]
\[ c_2 \Leftrightarrow \neg e_1 \]
\[ c_3 \Leftrightarrow \neg e_4 \]
\[ c_4 \Leftrightarrow \neg e_4 \]

(C2)
\[ e_2 \Rightarrow n_1 < n_2 \]
\[ e_3 \Rightarrow n_2 < n_3 \]
\[ e_5 \Rightarrow n_4 < n_5 \]

\[ c_5 \Leftrightarrow e_1 \land \neg e_2 \]
\[ c_6 \Leftrightarrow e_2 \]
\[ c_7 \Leftrightarrow e_2 \land \neg e_3 \]
\[ c_8 \Leftrightarrow e_3 \]
\[ c_9 \Leftrightarrow e_4 \]

(C3)
\[ e_1 \Rightarrow n_1 < n_4 \]
\[ e_3 \Rightarrow n_3 < n_4 \]
\[ e_5 \Rightarrow n_5 < n_2 \]

\[ c_{10} \Leftrightarrow e_4 \land \neg e_5 \]
\[ c_{11} \Leftrightarrow e_5 \]

Figure 6.3. An example of a SMT translation of a contextual unfolding

the read event must be fired first as it does not consume the token from the shared variable place. Figure 6.3 shows a SMT translation of the contextual unfolding in Fig. 5.2. Note that no constraints of type (C1) are needed for this particular example. Note also that if \( c_8 \) and \( c_{11} \) are set to have true values, they imply that all boolean variables for events must be true as well. This further implies that \( n_5 < n_2 < n_3 < n_4 < n_5 \) must hold. However, this is unsatisfiable and therefore a marking with both \( c_8 \) and \( c_{11} \) is not reachable.

Properties on Shared Memory

A reachable marking of an unfolding of a program represents a global state of the program. It is possible to augment the base translations given above to also describe the values that shared variables can have in such global states. In order to do this, we need to create a mapping between shared variable conditions and the values of shared variables that the conditions represent. Let \( \text{variable}(c) \) be the name of the shared variable corresponding to a shared variable condition \( c \) and let \( \text{value}(c) \) be a concrete value or a symbolic value expression of the variable when \( c \) is reached during an execution. Note that each shared variable condition has a unique concrete or symbolic value expression that it represents. We will prove this property later in this chapter. The mapping can then be
constructed as follows.

For each shared variable condition $c$:

$$c \Rightarrow \text{variable}(c) = \text{value}(c) \quad (S1)$$

As an example, let us consider again the unfolding in Fig. 6.2. Let us assume that the condition $c_2$ is an initial shared variable condition representing a shared variable $X$. If the initial value of $X$ is 0, then the formula (S1) gives us the following mapping: $c_2 \Rightarrow (X = 0)$. The event $e_3$ in the unfolding corresponds to a write event. If $e_3$ models a write operation that writes a concrete value 5 (i.e., a value that does not depend on input values) to the variable $X$, then the formula for condition $c_7$ becomes $c_7 \Rightarrow (X = 5)$. If $e_3$ models an operation that writes a value that depends on inputs, then the formula for $c_7$ could be, for example, $c_7 \Rightarrow (X = \text{input}_2 + 1)$.

In a regular unfolding of a program each thread has its own copy of the shared variable conditions. As each such copy represents the same value of the shared variable, it is enough to consider the shared variable conditions of a single thread. In the following we assume this thread to be the first thread (i.e., the thread with the smallest thread identifier). In other words, there is no need to create formulas of type (S1) for shared variable conditions that do not belong to the first thread. With contextual unfoldings there are no such copies and therefore all the shared variable conditions need to be taken into account.

By using this encoding it is then possible to append properties such as $X > 5 \land Y < 10$, where $X$ and $Y$ are names of shared variables, to the SMT translation. This makes the translation satisfiable only if there is a reachable state in the program where the property corresponding to the appended formula holds.

**Properties on Program Location**

It is also possible to use an unfolding to check if threads can be at certain lines of code at the same time or to require that some property on the global state must hold whenever a thread is at a given line of code. In order to do this, a mapping between lines of code and thread conditions in the unfolding needs to be constructed. For example, assume that there are two separate executions where thread 2 executes line 10. Let $c_7$ and $c_{10}$ be the thread conditions that are marked when thread 2 executes the line 10
in the first and second executions, respectively. We can now construct a formula $pc_2^{10} \iff c_7 \lor c_{15}$ to describe the mapping ($pc_j^i$ is true if thread $j$ is at line $i$). By requiring, for example $pc_2^{10}$ and $pc_3^{75}$ to be true, we can ask a SMT solver to decide if there is a reachable state where thread 2 is at line 10 and thread 3 is at line 75 at the same time. The program location based properties can naturally be combined with shared memory valuation based properties as well.

**Deadlock Detection**

It is also possible to extend the SMT translation to check for deadlocks. A deadlock occurs if in the unfolding there is a reachable marking such that at least one of the threads has not yet reached its final state and there are no events enabled. One possible way to capture this is by using an encoding that contains the base translation formulas and in addition the following ones.

For each event $e$:

$$\bigwedge_{c_i \in \bullet e} c_i \Rightarrow enabled$$

(D1)

Let $F_t$ be the set of thread conditions corresponding to the final states of thread $t$. For each thread $t$:

$$final_t \iff \bigvee_{c_i \in F_t} c_i$$

(D2)

Let $W_t$ be the set of thread conditions of thread $t$ such that the event in the postset of these conditions are lock acquire events. For each thread $t$:

$$atLock_t \iff \bigvee_{c_i \in W_t} c_i$$

(D3)

Let $n$ be the number of threads in the program.

$$\neg enabled \land \bigvee_{t=1}^{n} atLock_t \land \bigwedge_{t=1}^{n} (atLock_t \lor final_t)$$

(D4)

The intuition behind this deadlock formulation is that it captures reachable markings where there are no events enabled, at least one thread is trying to acquire a lock and the reachable marking has to be such that all threads have either reached their final states or are trying to acquire a lock.
6.3 Checking Deadlocks Directly from Unfoldings

It is also possible to check if a program under test contains a deadlock by searching markings corresponding to deadlocks directly from the unfolding without translating it into SMT. In this section we describe a backtrack search that can efficiently search for deadlocks in an unfolding of a program. As discussed earlier, a deadlock corresponds to a reachable marking where there are no events enabled and at least one of the threads has not reached its final state. The problem now is to find such reachable markings.

As the only operations that can block in the programs we are considering in this thesis are lock operations, we can restrict the deadlock detection to two different cases: either some thread acquires a lock and never releases it or there is a circular dependency on the locks such that a number of threads wait on each other.

Deadlocks corresponding to the first case are easy to detect. A search algorithm needs to find all thread conditions where a thread wants to acquire a lock that is held but not released by a thread that has terminated. If the terminated thread released the lock, the lock acquire by the other thread would become enabled. This means that deadlocks can be found by temporarily adding release events to the final thread states of terminated threads that are holding locks and then computing possible extension for the temporary events. Each such possible extension corresponds to a case where a thread is waiting for a release event that never happens.

Detecting circular deadlocks is more challenging. For a circular deadlock to occur, there must exist a reachable state where a number of threads form a set such that each thread in the set holds at least one lock. In addition to this, each thread in the set wants to acquire a lock held by some other thread in the set. If the locking dependencies are circular, the threads cannot proceed. It is possible to locate such circular dependencies by starting a search from each thread condition that holds at least one lock and is followed by an acquire lock operation. A recursive backtrack search algorithm to detect circular deadlocks is shown in Figure 6.4.

The algorithm uses the following notation: \( \text{wants}(c) \) denotes the lock that a thread wants to acquire after reaching a thread condition \( c \), and \( \text{holds}(c) \) denotes the set of locks being held by a thread after reaching \( c \). The search algorithm is assumed to be called every time when a thread condition \( c \) is added to the unfolding such that \( \text{holds}(c) \neq \emptyset \) and \( \text{wants}(c) \neq \emptyset \).
Input: An unfolding $unf$, a thread condition $c$
Output: A set of found deadlocks containing $c$

1: $wantedResource = wants(c)$
2: $foundDeadlocks = \emptyset$
3: $SEARCH(c, \{c\})$
4: $foundDeadlocks = FILTER UNSAT(foundDeadlocks)$
5: return $foundDeadlocks$

6: function $SEARCH(c$: condition, $coSet$: set of conditions)
7: $candidates = FIND NEXT(c)$
8: for all $candidate$ in $candidates$ do
9: if $coSet \cup \{candidate\}$ is a co-set then
10: $S = coSet \cup \{candidate\}$
11: if $holds(candidate) \in wantedResource$ then
12: $foundDeadlocks = foundDeadlocks \cup S$
13: else
14: $SEARCH(candidate, S)$

15: function $FIND NEXT(c$: condition)
16: $S = \{a | a \in co, wants(a) \in holds(c) \land holds(a) \neq \emptyset\}$
17: return $S$

Figure 6.4. Algorithm for deadlock detection

$\emptyset$. This corresponds to a state in the execution of a thread where a circular deadlock is possible. The algorithm starts constructing a set of concurrent thread conditions ($coSet$ at line 6) that are candidates for forming a circular deadlock. Initially this set contains only the thread condition $c$ given as an input to the algorithm. At each search step (i.e., a call to the function $SEARCH$) the algorithm searches for all such thread conditions $t$ that are concurrent with the current candidate set, wants to acquire some lock held by the previous thread condition that was added to the set (argument $c$ at line 6) and holds some lock itself. This search is done in the function $FIND NEXT$. For each such thread condition the algorithm starts a recursive search by adding the thread condition to the candidate set. If the search cannot find a thread condition to augment the candidate set, it backtracks (i.e., removes the most recently added thread condition from the candidate set and explores another way to extend the candidate set). If the found thread condition holds a lock that the starting thread
condition \( c \) wants to acquire, then the set of thread conditions correspond
to a state with a circular deadlock.

The deadlocks found by the algorithm are only potential deadlocks as the
search is done without taking the path constraints collected by dy-
namic symbolic execution into account. Therefore it is possible that there
are no concrete input values that lead to the deadlock. For this reason the
path constraint corresponding to the reachable marking for the potential
deadlock must be checked (function \texttt{FILTER_UNSAT}).

The \texttt{FINDNEXT} function that searches for the next candidate thread con-
dition can be implemented naively by iterating though all thread condi-
tions but this is naturally inefficient. Fortunately in typical cases it is
possible limit the search space.

\textbf{Lemma 9.} Let \( c_1 \) be a thread condition in an unfolding of a program \( P \)
such that \( \text{holds}(c_1) \neq \emptyset \) and \( \text{wants}(c_1) \neq \emptyset \). Let \( c_2 \) be a thread condition
that is concurrent with \( c_1 \) and \( \text{wants}(c_2) \in \text{holds}(c_1) \). Let \( e_1 \) be the lock
event that causally precedes \( c_1 \) and corresponds to acquiring the lock re-
turned by \( \text{wants}(c_2) \). It then holds that either (i) \( c_2^* = \emptyset \) or (ii) there exists
an event or a possible extension \( e_2 \in c_2^* \) such that \( \text{\( e_1 \cap e_2 \neq \emptyset \) or \( e_3 \cap e_2 \neq \emptyset \)}, \)
where \( e_3 \) is an unlock event that releases the lock acquired by \( e_1 \).

\textbf{Proof:} Let \( l_1 \) be the thread condition in \( *e_1 \). As \( l_1 \neq c_2 \), there are four
possible cases: (1) \( c_2 < l_1 \), (2) \( l_1 \# c_2 \), (3) \( l_1 \co c_2 \) or (4) \( l_1 < c_2 \). From our
assumptions we know that \( l_1 < c_1 \). Now cases (1) and (2) directly imply
that \( c_1 \) and \( c_2 \) are not concurrent which contradicts our assumptions and
therefore both of these cases are impossible. In case (3) we know that
\( l_1 \co c_2 \) and the next operation from \( c_2 \) wants to acquire the same lock
that \( l \) represents. Therefore an event or a possible extension in \( c_2^* \) must
exist. That events corresponds to the event \( e_2 \) and therefore \( e_2 \) exists. Let
us finally consider the case (4). Since \( l_1 < c_2 \), it must hold that \( e_1 < c_2 \).
Otherwise \( c_1 \) and \( c_2 \) would be in conflict which leads to a contradiction.
Let us assume that there exists an unlock event \( e_3 \) that releases the lock
acquired by \( e_1 \) and \( c_1 < e_3 \). Then as \( c_1 \) and \( c_2 \) are concurrent, \( c_2 \) must be
concurrent with the unlock event \( e_3 \) and the lock condition in its postset.
This implies that there exists an event or a possible extension \( e_2 \in c_2^* \) such
that \( e_3 \cap e_2 \neq \emptyset \). If \( e_3 \) does not exists, then there are no events or possible
extensions in the postset of \( c_2 \). \( \square \)

The last thread condition that has been added to the candidate set in
the algorithm of Figure 6.4 can be seen to correspond to the condition \( c_1 \)
1: function FINDNEXT(c: condition)
2: $S := \text{empty set}$
3: $E := \text{empty set}$
4: for all lock in holds(c) do
5: $e_1 := \text{FINDPRECEDINGLOCKEVENT}(c, \text{lock})$
6: $lc := \text{lock condition in } *e_1$
7: add all events and known possible extension events in $lc^*$ to $E$
8: $U := \text{FINDFOLLOWINGUNLOCKEVENTS}(c, \text{lock})$
9: for all $e_3$ in $U$ do
10: $lc := \text{lock condition in } e_3^*$
11: add all events and possible extension events in $lc^*$ to $E$
12: for all $e_2$ in $E$ do
13: $c_2 := \text{thread condition in } *e_2$
14: if ($e_2$ and $e_1$ do not belong to the same thread) then
15: if (holds($c_2$) $\neq \emptyset$ and $c_2 \text{ co c}$) then
16: add $c_2$ to $S$
17: for all $c_2$ in the list of thread conditions such that $c_2^* = \emptyset$ do
18: if (holds($c_2$) $\neq \emptyset \land \text{wants}(c_2) \neq \emptyset \land c_2 \text{ co c}$) then
19: add $c_2$ to $S$
20: return $S$

Figure 6.5. Efficient FindNext algorithm

in Lemma 9. Then any condition returned by FINDNEXT corresponds to $c_2$ in Lemma 9. Now based on the lemma, we can restrict the search for new candidates by considering only those thread conditions that are in the presets of lock events that correspond to either $e_2$ or $e_3$ in Lemma 9. Finding such events can be done efficiently by traversing the arcs in the unfolding. In addition to this, we also need to check thread conditions with empty postsets. This can be done efficiently by maintaining a separate list of such conditions. This leads to the algorithm shown in Fig. 6.5. The \textsc{FindPrecedingLockEvent} subroutine returns the lock event $e_1$ such that $e_1 < c$ and the lock acquired by $e_1$ has not been released before $c$. The \textsc{FindFollowingUnlockEvents} subroutine returns the unlock events that release the lock acquired by $e_1$ (note that there can be several of them).

The algorithm in Fig. 6.4 can be used find all circular deadlocks in the program under test if it is executed for each added thread condition such that the executing thread both holds a lock and wants to acquire a lock.
If there are no such thread conditions (e.g., in the case where the program has only one lock or no locks at all), there is no need to perform a search and in these cases the deadlock checking does not cause additional overhead.

6.4 Correctness

The approaches discussed in this chapter are based on finding markings of unfoldings that represent reachable global states satisfying a given property. From Chapters 3 and 5 we know that all executions of a program are represented in the unfoldings generated by the testing algorithms in Fig. 3.11 and Fig. 5.5. As each reachable shared state (i.e., a valuation of the shared variables and locks) is reachable by some execution, we know that all of them are represented by reachable markings in the unfolding. To use the approaches discussed in this chapter, we still need to show that any execution reaching the same marking in the unfolding observes the same symbolic shared state and that the testing algorithms do not generate unnecessary markings that represent unreachable states. Note that the following discussion applies both for the regular and contextual unfoldings of programs.

In the following we say that two symbolic shared states (i.e., shared states reached by using symbolic execution) are the same if they both represent the same sets of concrete shared states. We also say that a marking $m$ of an unfolding of a program has a satisfiable path constraint if the conjunction of symbolic constraints stored to the events that causally precede the conditions in $m$ is satisfiable.

**Lemma 10.** A marking $m$ with a satisfiable path constraint is reachable in an unfolding of a program $P$ if and only if there exists an execution $ex \in EX(P)$ that is represented by a set $S$ of events in the unfolding of $P$ and firing these events leads to the marking $m$.

**Proof.** Let us first show that if $m$ is reachable and has a satisfiable path constraint, then $ex$ exists. Let us show this by a contradiction by assuming that $ex$ does not exist. This means that the set $S$ does not correspond to any execution in $EX(P)$. There, however, must exist a subset $S'$ of $S$ such that the marking $m'$ reached after firing the events in $S'$ is reachable by some execution $ex' \in EX(P)$ but firing an additional event $e \in S$ from $m'$ leads to a marking that is not reachable by any execution. Such $S'$ and $m'$
must exist as at least the initial marking is reachable by any execution
(i.e., \( S' \) is an empty set and \( m' \) is the initial marking).

As all executions are represented in the unfolding according to Lemma 3
(and Lemma 8 for contextual unfoldings), it must be the case that the
operation \( o \) modeled by \( e \) cannot be performed in the state represented
by \( m' \). Let \( t \) be the thread that is supposed to perform \( o \). As the testing
algorithms add \( e \) to the unfolding only when a concrete test execution
covers the event, we know that \( t \) wants to perform an operation of the
same type as \( o \) after reaching \( m' \). Read, write and unlock operations are
enabled in any state where a thread wants to perform such an operation.
If \( o \) is a lock operation, it must be enabled after following \( ex' \). This is
because \( ex' \) is modeled by \( S' \) and for \( e \) to be enabled in \( m' \), either the lock
in question has not been accessed in \( ex' \) or the last operation accessing the
lock is an unlock. This means that if \( o \) is one of these operation types, it is
possible to extend \( ex' \) by performing \( o \). In this case \( o \) would be modeled by
\( e \) which leads to a contradiction. Note that the testing algorithms check if
an existing event of the same type is enabled in a marking before adding
a new event. Therefore there cannot be any duplicate events for \( o \) in the
unfolding.

There is one last possibility: \( o \) corresponds to following either a true or
false branch at a conditional statement depending on symbolic values. In
this case the only possibility when \( o \) cannot be performed is that the path
constraint leading to \( m \) becomes unsatisfiable. This contradicts our initial
assumption and therefore \( ex \) must exist.

Let us now show the other direction. If \( ex \) exists and is modeled by the
events in \( S \) such that firing them leads to \( m \), then \( m \) is trivially reachable.
Furthermore the path constraint for \( m \) must be satisfiable as it is reached
by an execution \( ex \) in \( EX(P) \) and the path constraints of executions in
\( EX(P) \) are trivially satisfiable.

\begin{lemma}
Let \( ex \) and \( ex' \) be executions in \( EX(P) \). If firing the sets of
events that represent these executions lead to the same marking \( m \) in the
unfolding of \( P \), then the symbolic shared states after the executions are the
same.
\end{lemma}

\begin{proof}
In order for both of the executions to lead to the marking \( m \), they
both need to fire the same sets of events. This means that both execu-
tions need to perform the same symbolic branch operations and therefore
the path constraints for both executions are the same. At each reach-

\end{proof}
able marking a shared variable is represented by a set of shared variable conditions (or a single shared variable condition in the case of contextual unfoldings) such that the closest causally preceding write event of these conditions is unique. Based on Lemma 2 (and Lemma 7 for contextual unfoldings), after this write event has been fired, the same concrete value or symbolic value expression has always been written to a shared variable. This means that if $ex$ and $ex'$ both reach $m$, then for every shared variable it holds that $ex$ and $ex'$ both observe the same concrete or symbolic value for it. As the path constraints for the executions are the same, so are the symbolic shared states.

Based on Lemma 11, the shared variable and lock conditions in a marking $m$ of an unfolding represents a specific symbolic shared state. In the following we denote this symbolic shared state as $\text{shared}(m)$.

**Theorem 7.** A shared state $s$ is reachable in a program $P$ if and only if there is a reachable marking $m$ with a satisfiable path constraint in the unfolding of $P$ such that $s$ is one of the concrete shared states represented by $\text{shared}(m)$.

**Proof.** Let us first show that if $s$ is reachable, then the marking $m$ exists. As $s$ is reachable, it must be reachable by some execution in $EX(P)$. Based on Lemma 3 (and Lemma 8 for contextual unfoldings) this execution is represented in the unfolding by a set of events and firing them leads to the marking $m$. The testing algorithm must have performed executions that have covered the events leading to $m$. Based on Lemma 11, the executions made by the testing algorithm have observed the same concrete or symbolic value expressions after each write event. Therefore $\text{shared}(m)$ is a symbolic shared state that covers $s$.

Let us now show the other direction. That is, if there exists a reachable marking $m$ with a satisfiable path constraint such that $\text{shared}(m)$ represents a concrete shared state $s$, then $s$ is reachable in $P$. As the path constraint of $m$ is satisfiable, based on Lemma 10 there then exists an execution in $EX(P)$ that reaches $m$. As $\text{shared}(m)$ represents $s$, then $s$ is reachable in $P$.

Let us finally show that unfoldings can be used detect deadlocks.

**Theorem 8.** A program $P$ contains a deadlock if and only if there is a reachable marking $m$ with a satisfiable path constraint in the unfolding of $P$ such that no events are enabled in $m$ and at least one thread condition in $m$ does not correspond to a final state of a thread.
Proof. Let us first show that if $P$ contains a deadlock, then $m$ exists. As a global state of $P$ with a deadlock exists, there must be an execution in $EX(P)$ that reaches this global state. The events that model this execution lead to a marking $m$ that, based on Lemma 10, has a satisfiable path constraint. As the execution deadlocks, any thread that has not reached its final state wants to acquire a lock that is already taken by some earlier lock operation but not released by an unlock operation. It is therefore easy to see that in such case the lock condition that represents the lock is not in $m$. It is also easy to see that the thread condition $c$ in $m$ for the thread that is waiting for the lock to be released does not correspond to a final state. Based on Lemma 2 (and Lemma 7 for contextual unfoldings) the thread always wants to perform the same type of operation after reaching $c$. This means that there can only be a lock event in the postset of $c$ and as the lock condition is not in $m$, none of the events in the postset of $c$ are enabled. Therefore $m$ as described in the theorem exists.

Let us now show the other direction: if $m$ exists, $P$ contains a deadlock. As $m$ has a satisfiable path constraint, based on Lemma 10 there is an execution $ex$ in $EX(P)$ that reaches $m$. Let $c$ be any thread condition in $m$ that does not correspond to a final state. This means that no event in the postset of $c$ is enabled. If a thread wants to perform a read, write, branch, or unlock operation after reaching $c$, one event corresponding to such operation would be enabled in $m$ by construction. Therefore the thread wants to perform a lock operation after reaching $c$. As no events are enabled in $m$, the lock condition corresponding to the lock operation is not in $m$. Therefore to reach $m$, $ex$ must have performed a lock operation that has acquired the lock and that the lock has not been released afterwards. As the same holds for all threads that have not reached their final states, it must hold that the execution $ex$ will deadlock after reaching $m$. \qed
7. Lightweight Stateful Approach For Testing Multithreaded Programs

The testing approaches presented so far in this thesis are based on exploring different interleavings of dependent operations. Sometimes exploring all such interleavings is unnecessary in order to cover the reachable local states of a program. As an example, let us consider a program that has an array of values in shared memory such that the accesses to the array are protected by a lock (i.e., a thread needs to acquire the lock before it can read a value from the array or write to it). If two threads access the array such that they read values from it or update distinct indices of the array, the accesses are independent of each other. This means that to in order to cover the local states of threads, it does not matter in which order the threads access the array. However, as the threads need to acquire and release the same lock, the testing approaches discussed so far explore both ways to interleave the lock acquires and therefore both ways to interleave the accesses to the array.

In the example above, the program ends up in the same global state regardless in which order the two threads are executed. One way to reduce the number of redundant test executions is to identify when a test execution reaches a program state that has already been covered and stop the execution as the subsequent states observed by the test execution are covered by other tests.

The testing algorithms discussed so far are so called *stateless algorithms*. That is, they do not store any state information that could be used to determine when a test execution covers an already explored state. Capturing and storing program states, for example to a hash table, is in principle simple. However, a full state of a complex multithreaded program can contain a large number of variables (i.e., large arrays) and therefore storing all the observed states can require a considerable amount of memory. Furthermore, if a part of a state is expressed symbolically, as is the case
when using DSE, comparing two states becomes more complicated. For example, in a symbolic state a variable \( x \) could have any value that satisfies a constraint \( x > 0 \). If another execution path leads to an identical symbolic state except that the constraint for \( x \) is \( x > 5 \), the first symbolic state subsumes the second one (i.e., the first state represents all concrete states of the second symbolic state). To determine if a symbolic state has been visited before, a subsumption check by a constraint solver is needed. This can be computationally expensive if the constraints are complex.

In this chapter we describe a lightweight approach to capture partial state information in an abstract way that does not require storing concrete values of variables nor subsumption checks that require constraint solvers. The new approach is based on the observation that sometimes it is easy to see that interleavings even with dependent transitions commute and thus lead to the same state. The concurrent accesses to the array as discussed above is an example of this. Note that in the array example we do not need to know the values stored in the array to be able to determine that the two test executions lead to the same state.

To be able detect such cases, we construct an abstract model of the program under test as a Petri net. This abstract model has a similar structure to the unfoldings discussed in earlier chapters. For example, we unroll all loops in the abstract model. However, we do not require the model to be acyclic and therefore non-equivalent executions can lead to the same abstract state in the model (i.e., to the same marking in the Petri net). We will present a new testing algorithm that constructs such Petri net models on-the-fly and uses them to avoid generating irrelevant test executions.

### 7.1 Finite Prefixes of Unfoldings

Before presenting the new testing algorithm, we will first extend the discussion of the unfoldings of Petri nets to include the concept of complete finite prefixes (see, e.g., [McM95, ERV02, EH08]). The reachable markings of a Petri net can be explored by traversing its computation tree. Exploring the full computation tree, however, is not always necessary. For example, let us consider the Petri net and its computation tree in Fig. 7.1. The marking of the Petri net after firing the transition \( t_1 \) is \( \{s_2, s_3\} \). Firing the transition sequence \( t_1t_2t_4 \) leads to the same marking and therefore there is no need to extend the computation tree further from the node that corresponds to the transition sequence. In a similar way, it is also possi-
able to compute a finite prefix of the unfolding that captures all reachable markings of a Petri net. However, as an unfolding is a more succinct representation than the computation tree, computing such a finite prefix is not as straightforward. The use of complete finite prefixes in the context of verification problems was first proposed by McMillan [McM92] and their construction has been well studied since. In the following we review the central concepts behind finite prefixes and for this purpose we use notation similar to [EH08].

Definition 22. The local configuration of an event $e$ in an unfolding is the set $\{e' | e' \leq e\}$.

Definition 23. Let $e$ be an event in the unfolding of a Petri net $N$. $\text{St}(e)$ denotes the marking of $N$ reached after firing the transitions corresponding to the events in the local configuration of $e$.

Definition 24. Let $\prec$ be a partial order on the events of an unfolding. An event $e$ in a prefix of the unfolding is a terminal (also known as a cut-off event) with respect to $\prec$ if there exists an event $e'$ of the unfolding such that $e' \prec e$ and $\text{St}(e') = \text{St}(e)$.

A finite prefix of an unfolding can be constructed by leaving out any events that are causally preceded by a terminal event. That is, if an event $e'$ has been added to the unfolding before an event $e$ and $\text{St}(e') = \text{St}(e)$, the unfolding process can be stopped at event $e$.

Example 23. Let us consider the Petri net and the unfolding in Fig. 7.1. The numbers on the events in the unfolding denote the order in which

Figure 7.1. A Petri net and its infinite computation tree and unfolding.
they have been added to the unfolding and therefore follow the \( \prec \) partial order. Let us assume that \( e_n \) denotes the event labeled with \( n \). The local configuration of \( e_4 \) consists of events \( e_1, e_2 \) and \( e_4 \). Firing the transitions corresponding to these events leads to \( \text{St}(e_4) = \{s_2,s_3\} \), which is a marking of the Petri net. Note that event \( e_1 \) has been added to the unfolding before \( e_4 \) and \( \text{St}(e_1) = \text{St}(e_4) \). This means that \( e_4 \) is a terminal and no events that are causally preceded by it need to be added to the unfolding. Similarly the event \( e_5 \) is also a terminal.

A prefix of the computation tree representation can also be constructed by not extending the nodes that correspond to a marking that has already been covered by some other node. Note, however, that such a computation tree can sometimes be considerably larger than the unfolding as is the case in Fig. 7.1.

It is important to note that not all partial orders \( \prec \) lead to complete prefixes [EH08]. A complete prefix not only represents all reachable markings of a Petri net but it also needs to contain enough information so that the full unfolding could be constructed based on the information in the prefix alone. It has been shown that if events are added to the unfolding in a so called adequate order, the construction of the unfolding can be stopped at terminal events without losing information. In other words, adequate orders lead to complete prefixes.

**Definition 25.** A partial order \( \prec \) is preserved by extensions if it satisfies the following property. Let \( e_1 \) and \( e_2 \) be events such that \( \text{St}(e_1) = \text{St}(e_2) \) and let \( e_1' \) and \( e_2' \) be any events that are reached from the local configurations of \( e_1 \) and \( e_2 \), respectively, by firing sequences of events \( E_1 \) and \( E_2 \). If \( l(E_1) = l(E_2) \) (i.e., \( E_1 \) and \( E_2 \) are labeled with the same transitions) and \( e_1 \prec e_2 \), then it holds that \( e_1' \prec e_2' \).

**Definition 26.** A partial order \( \prec \) on the events of an unfolding is an adequate order if \( \prec \) is well-founded and it is preserved by extensions.

It has also been shown that if \( \prec \) is preserved by extensions, it is well-founded [CK07]. Therefore the requirement for \( \prec \) to be preserved by extensions is enough to guarantee that it is adequate. One example of an adequate order is to define that \( e \prec e' \) holds if the local configuration of \( e \) contains less events than the local configuration of \( e' \) [McM95]. This adequate order, however, does not necessarily lead to a minimal complete prefix as shown in [ERV02]. In the remainder of this section we will summarize how to compute a so called ERV-adequate order (\( \prec_{ERV} \)) as
described in [ERV02]. The complete prefix obtained by using the ERV-
adequate order leads to a complete prefix that is minimal in the sense
that if \( \text{St}(e) = \text{St}(e') \), then either \( e \prec e' \) or \( e' \prec e \) (i.e., one of the events
is a terminal). There are also other notions of minimality, for example,
see [Hel99].

**Definition 27.** Let \( T_1 \) and \( T_2 \) be sequences of transitions and let \( \ll \) be any
total order on the transitions. The transition sequence \( T_1 \) is lexicographi-
cally smaller than \( T_2 \) if either (i) \( T_1 \) is a prefix of \( T_2 \) or (ii) \( T_1 = T_w t_a T_u \) and
\( T_2 = T_w t_b T_v \), where \( T_w \) is the longest common prefix of \( T_1 \) and \( T_2 \) and for
the transitions \( t_a \) and \( t_b \) it holds that \( t_a \ll t_b \).

**Definition 28.** Let \( \ll \) be any total order on the transitions of a Petri net
and let \( E \) be a set of events in the unfolding of the Petri net. Let \( \varphi(E) \) be a
sequence of transitions ordered according to \( \ll \) such that every transition \( t \)
in \( \varphi(E) \) contains a unique event in \( E \) that is labeled with \( t \). We denote that
\( \varphi(E_1) \ll \varphi(E_2) \) if \( \varphi(E_1) \) is lexicographically smaller than \( \varphi(E_2) \).

If \( E \) contains \( n \) events then \( \varphi(E) \) contains \( n \) transitions. The \( \varphi(E) \) can
therefore be seen as a Parikh vector.

**Example 24.** Let us assume that all transitions are labeled as \( t_n \), where
\( n \) is a natural number. Let us define \( \ll \) such that for any \( i \) and \( j \) it holds
that \( t_i \ll t_j \) if \( i < j \). Given sets of events \( E_1, E_2 \) and \( E_3 \) such that \( \varphi(E_1) =
t_1 t_3 t_4 t_3 t_4, \varphi(E_2) = t_1 t_2 t_5 \) and \( \varphi(E_3) = t_1 t_3, \) it holds that \( \varphi(E_2) \ll \varphi(E_1) \) and
\( \varphi(E_3) \ll \varphi(E_1) \).

**Definition 29.** A Foata normal form of a local configuration \( C \) is a se-
quence of sets of events \( FC = C_1 \ldots C_n \) such that \( C_1 \cup \ldots \cup C_n = C \). Further-
more, for any \( 1 \leq i \leq n \) it holds that \( C_i = \text{Min}(S) \), where \( S \) is a set that
is obtained by removing all those events from \( C \) that are included in the
sequence \( FC \) prior to \( C_i \).

**Definition 30.** Let \( FC_1 = C_{11} \ldots C_{1n} \) and \( FC_2 = C_{21} \ldots C_{2m} \) be the Foata
normal forms of two local configurations. We say that \( FC_1 \ll FC_2 \) if there
exists \( 1 \leq i \leq n \) such that (i) for every \( 1 \leq j < i \) it holds that \( \varphi(C_{1j}) =
\varphi(C_{2j}) \), and (ii) \( \varphi(C_{1i}) \ll \varphi(C_{2i}) \).

We are now finally ready to describe the ERV-adequate order.

**Definition 31.** Let \( C \) and \( C' \) be the local configurations of events \( e \) and
\( e' \). Additionally, let \( FC \) and \( FC' \) be the Foata normal forms of \( C \) and \( C' \),
respectively. It holds that \( e \ll_{ERV} e' \) if:
• *C* contains less events than *C* ′ (i.e., \(|C| < |C'|\)), or

• \(|C| = |C'| \text{ and } \varphi(C) \ll \varphi(C')\), or

• \(|C| = |C'| \text{ and } \varphi(C) = \varphi(C') \text{ and } FC \ll FC'\)

To sort two events according to \(\prec_{ERV}\), the sizes of their local configurations are first compared. If this is not enough to differentiate the events, the lexicographical order is used. Finally, if even this is not enough, the Foata normal forms are compared.

### 7.2 Capturing Program States with Petri Nets

To be able to capture abstract state information that can be used for state matching, we use Petri nets to model the program behavior that has been observed during test executions. To construct such a model, we proceed similarly as in the previous chapters where we constructed unfoldings of programs. This time, however, we want to make the model more compact by reusing places that correspond to local states of threads, shared variables or locks whenever possible. For example, a lock can be modeled with a single place such that acquire events remove a token from this place and release events put the token back.

To be more precise, the initial state of the program is modeled by having a place for each thread, shared variable and lock in the program under test. Places for threads are abstract representations of their local states. The places for shared variables represent valuations of that shared variable. To model the behavior observed during test executions, the constructs shown in Fig. 7.2 are used. For clarity, the places corresponding to shared variables and locks have a darker color than places for abstract local states.

The intuition behind the modeling constructs is as follows. When a thread is in a specific local state such that the next operation to be executed is a write, the operation always results in the same subsequent local state and the same value to be written to the shared memory regardless of the current valuation of the shared variable. This is represented in Fig. 7.2 such that if the write transition marked with dashed lines is added to the model after the transition with solid lines from the same local state, both transitions have the same places representing the
lightweight stateful approach for testing multithreaded programs

Figure 7.2. Modeling constructs

local state and the shared variable in their postsets. For read operations
the resulting local states are always different if the shared variable places
are different (i.e., in cases where the shared values being read might be
different). This is again illustrated by a second read transition marked
with dashed lines in Fig. 7.2. However, reading a value does not change
it and therefore a read operation can be modeled with a transition that
returns the token back to the original shared variable place. Locks con-
tain no other state than their locked status. Thus for each lock there is
always only one lock place and acquiring a lock takes a token from this
place and releasing the lock puts the token back to the same place. As
with the approaches in the previous chapters, local operations of threads
are not modeled explicitly as a thread executes them always in the same
way until the next global operation is encountered. The symbolic con-
straints resulting from executing conditional statements symbolically as
part of DSE are stored to the corresponding transitions for the true and
false branches. As such constraints restrict the possible values in a local
state, branching transitions always lead to new abstract local states.

Example 25. Let us consider the program shown in Fig. 7.3 and a test
execution that executes the statements on lines 1,2,3,4,5,6 in that order.
Modeling this execution starts with an initial marking \( \{s_1, s_2, x_1, y_1, t_1\} \). In
the initial state the lock acquire operations of both threads are enabled.
These are modeled as the transitions \( t_1 \) and \( t_2 \). The lock transition belong-
ing to thread 1 is then fired and a marking \( \{s_3, s_2, x_1, y_1\} \) is obtained. In
this new state the operation \( x = 1 \) is the only one that is enabled. As the
Thread 1:
1: acquire(lock);
2: X = 1;
3: release(lock);

Thread 2:
4: acquire(lock);
5: Y = 1;
6: release(lock);

Figure 7.3. Locking example

model does not contain a transition that is enabled in the current marking, the transition $t_3$ is added to the model and fired. The rest of the test execution is processed in a similar manner to obtain the net in Fig. 7.3. Note that if a second test execution is made such that thread 2 performs its operations first, no new transitions need to be added to the model. Furthermore, both of these executions end up in the same Petri net marking indicating that the resulting global program states are the same.

Example 26. Fig. 7.4 shows another program where three threads write concurrently to the same shared variable. The partial model on the top of the figure is obtained by performing a test execution where thread 1 is executed first, thread 2 second and thread 3 last. In the initial state the writes for all threads are enabled and this is modeled by the transitions $t_1$, $t_2$ and $t_3$. After executing the write of thread 1, the enabled writes of thread 2 and thread 3 are modeled as transitions $t_4$ and $t_5$. The final write of the test execution is modeled as transition $t_6$. The model on the bottom shows the complete model for the program. In this case there are six possible ways to interleave the write operations. However, there are only three possible end states (markings) for these interleavings and therefore if the program continues after the writes, it is possible to cut the exploration of some of these interleavings.

7.3 Naive Stateful Testing Algorithm

A simple way to construct a complete model of a program under test is to initially model a random test execution and start traversing the computa-
Thread 1:  
1: \( X = 1 \);

Thread 2:  
2: \( X = 2 \);

Thread 3:  
3: \( X = 3 \);

Figure 7.4. Concurrent writes example

...
**Input:** A program $P$

1. $model := \text{empty Petri net}$
2. $visited := \emptyset$
3. extend model with a random test execution
4. $\text{EXPLORE}(M_0, \emptyset)$

5. **procedure** $\text{EXPLORE}(M, S)$

6. \hspace{1em} \textbf{if } $M \notin visited$ \textbf{then}

7. \hspace{2em} $visited := visited \cup \{M\}$

8. \hspace{2em} $\text{PREDICTTRANSITIONSFROMMODEL}(M)$

9. \hspace{2em} \textbf{if } model is incomplete at $M$ \textbf{then}

10. \hspace{3em} $\text{EXTENDMODEL}(P, S, k)$

11. \hspace{2em} \textbf{for all } transitions $t$ enabled in $M$ \textbf{do}

12. \hspace{3em} $M' := \text{FIRE}(t, M)$

13. \hspace{3em} $S' := S$ appended with $t$

14. \hspace{3em} $\text{EXPLORE}(M', S')$

**Figure 7.5.** Naive testing algorithm

Consider the program and its model after the first test execution in Fig. 7.4. Let us assume that we are exploring a marking $m = \{s_1, s_5, s_3, x_3\}$. The model is incomplete at this marking because no transition for thread 1 is enabled in this state. However, we known that each transition from a place representing a local state of a thread has the same type (i.e., from a given local state, the operation the thread wants to perform is always the same). As there is a write transition in the postset of $s_1$, we know that thread 1 wants to perform a write operation. In cases like this, the missing transition ($t_7$ in our example) can be added to the model without actually performing a concrete test execution. The subroutine $\text{PREDICTTRANSITIONSFROMMODEL}$ performs such analysis for each visited marking. To be more precise, $\text{PREDICTTRANSITIONSFROMMODEL}$ checks the postsets of the places for local thread states to determine the operations the threads want to perform and adds any missing transitions to the model. For reads and writes this is trivial. For lock operations it needs to be checked that the lock is free in the current state (i.e., the marking contains the respective lock place). Using $\text{PREDICTTRANSITIONSFROMMODEL}$ can sometimes significantly reduce the need for concrete test executions. For example, the final model for program in Fig. 7.4 can be constructed with information obtained from a single test execution instead of six executions that cover all the interleavings. This optimization can have
Lightweight Stateful Approach For Testing Multithreaded Programs

a significant performance improving impact.

**Computing Inputs For Test Executions** The `EXTENDMODEL` subroutine performs a test execution both concretely and symbolically. The subroutine takes as input a sequence $S$ of transitions that leads to the state that is being explored. Bound $k$ for the execution length is used to guarantee termination. To get concrete input values for the test execution, all the symbolic constraints associated with the branching transitions in $S$ are collected and their conjunction is solved using a constraint solver. The sequence $S$ is given to a runtime scheduler that schedules the execution in such a way that the operations are performed in the same order as the corresponding transitions in $S$. After reaching the target state, the scheduler is free to follow any schedule.

We call the algorithm in Fig. 7.5 naive because it explores interleavings of global operations even if they are independent. This is often unnecessary to find errors such as assertion violations or to preserve the reachability of local states of threads. Naturally different interleavings of independent operations lead to the same state and the algorithm backtracks in such cases. The number of such backtracks, however, can grow rapidly if the program under test contains a large number of threads. To avoid exploring unnecessary interleavings, we present next a similar algorithm based on unfolding the model. Like the unfolding approaches discussed earlier in this thesis, such an approach is guaranteed to cover all feasible control flow paths of threads and to detect all assertion violations. Detecting all deadlocks during the testing itself is not guaranteed. Another approach would be to use partial order reduction algorithms such as DPOR to avoid exploring irrelevant interleavings. As DPOR computes backtracking points based on state transitions observed after the backtracking point, naively combining the state matching approach discussed here would actually lead to an incomplete algorithm. We will return to this topic in Section 7.5.

**7.4 Stateful Unfolding Based Testing Algorithm**

To explore an unfolding of the model instead of the computation tree, we make a small modification to the modeling approach: instead of modeling a shared variable with a single place in each reachable marking, we
Lightweight Stateful Approach For Testing Multithreaded Programs

**Input:** A program $P$

1. $model := \text{empty Petri net}$
2. $unf := \text{initial unfolding}$
3. $visited := \emptyset$
4. extend model with a random test execution
5. $extensions := \text{events enabled in the initial state}$
6. **while** $extensions \neq \emptyset$ **do**
   7. choose $\prec$-minimal event $e$ from $extensions$
   8. $M := St(e)$
   9. $\text{PREDICTTRANSITIONSFROMMODEL}(M)$
   10. **if** model is incomplete at $M$ **then**
       11. $\text{EXTENDMODEL}(P, e, k)$
   12. **else**
       13. add $e$ to $unf$
       14. $extensions := extensions \setminus \{e\}$
       15. **if** $M \notin visited$ **then** // $e$ is not a terminal
           16. $visited := visited \cup \{M\}$
       17. $extensions := extensions \cup \text{POSSIBLEEXTENSIONS}(e, unf)$

**Figure 7.6.** Unfolding algorithm

replicate the place for each thread. In other words, we use the same place replication approach as in Chapter 3. If place replication is not used, the unfolding process would explicitly explore different interleavings of read transitions. This change also makes the unfolding of the model isomorphic to the one constructed by the testing algorithm in Chapter 3.

The unfolding based testing algorithm is shown in Fig. 7.6 and the idea behind it is similar to the naive algorithm. Initially a random test execution is performed to start the model construction. The algorithm maintains a set of events that can be used to extend the unfolding. Initially such events are those that are enabled in the initial state. The algorithm then starts adding these extensions to the unfolding in the order specified by the partial order $\prec$ (lines 5-6). Any adequate order (see Sect. 7.1) could be used here. In the experiments in Chapter 8, we use the ERV adequate order described earlier in this chapter.

To be able to avoid exploring states multiple times, the algorithm computes $St(e)$ for each event $e$ added to the unfolding. This can be seen as the state that is reached by following the shortest execution path to the event $e$. For the obtained marking, the algorithm performs the same analysis
for missing transitions as the naive algorithm does (line 9). If the model
is incomplete at the marking after adding the predicted transitions, a new
test execution is performed to update the model. Otherwise the algorithm
adds the selected event to the unfolding and determines if it is a termi-
nal. This is done by checking if an event with the same marking \( St(e) \)
has already been added to the unfolding (line 15). If the event is not a
terminal, the algorithm computes a set of new events that can be added
to the unfolding and adds these events to the set of possible extensions.

As the unfolding being constructed is the same as the unfolding con-
structed by the algorithm in Chapter 3, computing possible extensions
can be done in the exact same way as in Chapter 4. Similarly computing
inputs and thread schedules for the test executions can be done as before.
The main difference to the earlier algorithms is therefore the addition of
terminal events that makes it possible to cut the unfolding process and
therefore to reduce the number of required test executions.

**Example 27.** To illustrate the unfolding based algorithm, let us consider
the Petri net model and its unfolding in Fig. 7.7. The model represents a
program with two threads that acquire a lock and read a shared variable \( x \). The first thread also branches its execution based on input values at
the end. To construct the net in Fig. 7.7, the algorithm first performs a
random test execution. In this example, any execution provides enough
information to model all the transitions shown in the Petri net model.
However, depending on which branch the first thread follows at the end,
the model remains incomplete at place \( s_9 \) or \( s_{10} \) as the corresponding local
state is not explored.

From the initial state it is possible to fire transitions \( t_1 \) and \( t_2 \). The
events 1 and 2 correspond to these transitions and are added to the set
of possible extensions. The algorithm selects event 1 to be added to the
unfolding and this results in a new reachable marking where it is possible
to fire event 3. The found event is added to the set of possible extensions
and the same process is continued until the algorithm selects the event 12
to be added to the unfolding. The marking computed at line 7 is the same
for this event as well as for event 11. Therefore event 12 is a terminal
(marked with a cross in Fig. 7.7) and possible extensions for it are not
computed.

Let us assume that the initial test execution did not explore the state
corresponding to place \( s_9 \). To add the event 13 to the unfolding, the algo-

\[ 125 \]
enough information to compute possible extensions for event 13. This is achieved by a test execution that follows the transitions corresponding to the events 1, 3, 5 and 13. Let us assume that the symbolic constraint associated with $t_7$ is $\text{input}_1 > 5$. Solving this constraint gives the test execution a concrete input value (e.g., a value 6). After performing the test execution, the algorithm knows that $s_9$ corresponds to an end state and can continue the unfolding process by adding the events 13 and 14.

7.5 Discussion

Other partial order reduction approaches. The state capture approach presented in this chapter can also be combined with other partial order reduction based approaches. It would be, for example, possible to use DPOR to improve the naive algorithm testing algorithm. This, however, is not completely straightforward. If the exploration of an execution is stopped due to reaching a previously visited abstract state, the DPOR algorithm might miss some backtrack points that would be added if the test execution were not stopped. In other words, the state transitions after reaching an already visited state may result in backtrack points in the earlier part of the execution. Therefore some information of what operations can be performed after a test execution is stopped is needed. In [YCGK08] an approach that computes summaries describing the operations that might be executed after a given state is presented. Adapting a similar approach is also possible for the state capturing method presented in this chapter. Investigating such combinations of partial order reduction methods and
the state capturing approach is left for future work.

**Memory requirements.** The presented approach is lightweight in the sense that full state information does not need to be stored. The testing algorithms based on this approach do, however, need to store markings of the model. This can in some cases still lead to a considerable memory overhead when compared to stateless testing algorithms as each stored marking contains a place for each shared variable in the program. The approach is therefore well suited for cases where the local states of threads consists of large amounts of data values but the size of the shared memory is relatively small.

**Constructing even more compact models.** The cases where the modeling approach presented in this chapter can determine that two executions lead to the same state are limited. One reason for this is that the approach does not include any complex reasoning on symbolic data values. All loops are also unrolled and therefore the approach cannot determine if the same local state is seen more than once in a single execution. One way to extend the modeling approach is to store the values of shared variables to their corresponding places in the model. It is then possible to use the same shared variable place whenever the shared variable has the same concrete value. Investigating such possibilities to take some of the data value information into account is left for future work.
8. Experiments

In this chapter we provide an experimental evaluation of the new testing algorithms presented in this thesis. As this thesis concentrates on developing testing approaches that are based on net unfoldings, the main question we want an answer to is whether unfolding based techniques are practical in the context of testing multithreaded programs. The main concern is the scalability of such approaches. This is because when arbitrary Petri nets are unfolded, the cost of computing possible extensions grows quickly as the size of the unfolding grows. We also compare the new algorithms to DPOR to get a better understanding of how unfoldings compare to other partial order reduction methods.

8.1 Implementation

We have implemented the testing algorithms used in the experiments in a prototype tool that supports a subset of Java programs. To enable symbolic execution and to control the scheduling of threads, the program under test is instrumented with additional code. This is done by first translating a Java bytecode representation of the program into a simplified intermediate language called Jimple by using the Soot Java optimization framework [VRCG+99]. This intermediate language representation is then instrumented and finally translated back into bytecode. As a constraint solver the tool uses Z3 version 4.3 [dMB08].

Note that in [KSH12] we have compared unfolding based testing against a tool called jCUTE [SA06a] that combines DSE and race detection and flipping algorithms to test Java programs. The tool, however, fails to explore all Mazurkiewicz traces for certain types of programs and it is no longer maintained. For this reason our new algorithms are not compared against jCUTE. Also Symbolic Java PathFinder [PMB+08] is not
used in the experiments as based on our experience it is not limited to cover Mazurkiewicz traces but also considers additional interleavings of threads.

8.2 Evaluation Subjects

The following benchmarks are used in the experiments. The *Filesistem* benchmark is from [FG05] where it was used to evaluate DPOR. Two variations of this benchmark are used, one with 16 threads and another with 18 threads. *Parallel Pi* is an example of a typical program that divides a task (in this case the computation of the value of π) to multiple threads and then merges the results of each computation. *Fib* and *Szymanski* are programs from the 1st International Competition on Software Verification (SV-COMP)\(^1\) except that they have been simplified by limiting how many times some potentially infinite loops are executed. This has been done so that the testing algorithms can explore the programs fully and therefore provide a fair comparison between the algorithms. *Dining* implements a dining philosophers problem where each philosopher eats twice. *Pairs* is an artificial example similar to the one in Fig. 3.1 in which the unfolding based approaches can achieve exponential reduction when compared to approaches that aim to cover each Mazurkiewicz trace. In *Locking* all accesses to the shared variables are protected by a single lock and a number of threads perform operations that access these shared variables. *Updater* contains a set of threads where some threads update values in shared memory and other threads read these values and perform work based on these values. *Writes* is similar to the program in Fig. 7.4 except that it has more threads and more writes per thread. Finally, synthetic benchmarks perform randomly generated sequences of operations on input values and on global variables.

Benchmarks with multiple variants are similar with each other except that the number of threads increases or the program otherwise increases in complexity. Benchmarks that contain deadlocks are denoted by writing the name of the benchmark in italics in Table 8.1. All the benchmarks above are available online\(^2\).


8.3 Experimental Evaluation

The experiments were conducted on a PC with a 2.66 GHz Intel Q9400 processor with four processor cores and 8GB of random access memory. The operating system used was Ubuntu version 12.04. A time limit of 30 minutes was used for each experiment. As the number of test executions performed by the testing algorithms can vary depending on the order in which the execution paths are explored (e.g., due to the initial random input values), the experiments were repeated 10 times. The average runtimes are reported.

8.3.1 Stateless Algorithms

Table 8.1 shows the results of using the testing algorithms from Chapter 3 (Unfolding), Chapter 5 (Contextual Unfolding) and DPOR combined with sleep sets as described in Chapter 2 to fully test the benchmark programs. The tests column shows the number of test executions performed by the algorithms. The time column shows the total time used by the testing algorithms. The new unfolding based approaches preserve the reachability of local states while the reachability of global states requires one to examine the resulting symbolic unfolding. Thus a direct comparison between the unfolding based algorithms and DPOR is not completely fair as the unfolding based algorithms do not necessarily cover all Mazurkiewicz traces and therefore are not guaranteed to detect deadlocks. Any deadlock can, however, be detected from the resulting unfolding and for this reason the DL column shows the total additional time required to detect all deadlocks (i.e., all different reachable markings of unfoldings that correspond to deadlocks) by using the search algorithm described in Chapter 6.3.

It has also been recently shown in [AAJS14] that the DPOR algorithm can be made optimal in the sense that it will cover each Mazurkiewicz trace only once. To be able to compare the number of tests performed by unfolding based approaches to optimal DPOR algorithms, Table 8.1 also shows the number of Mazurkiewicz traces in the programs. Based on the experiments in [AAJS14] the optimal DPOR algorithm is computationally more demanding (i.e., it requires more time per test execution) than

---

3The number of Mazurkiewicz traces have been computed by testing a program with the DPOR algorithm and then subtracting the number of executions, where all threads ended up in a sleep set, from the total number of performed executions. As discussed in Chapter 2, this gives the number of Mazurkiewicz traces.
Experiments

Table 8.1. Experimental Results of Stateless Algorithms

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>M. traces</th>
<th>Unfolding</th>
<th>Contextual Unfolding</th>
<th>DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Time</td>
<td>DL</td>
<td>Tests</td>
</tr>
<tr>
<td>Filesystem(16)</td>
<td>8</td>
<td>3</td>
<td>0m 0s</td>
<td>+0s</td>
</tr>
<tr>
<td>Filesystem(18)</td>
<td>32</td>
<td>3</td>
<td>0m 0s</td>
<td>+0s</td>
</tr>
<tr>
<td>Parallel Pi 1</td>
<td>720</td>
<td>720</td>
<td>0m 2s</td>
<td>+0s</td>
</tr>
<tr>
<td>Parallel Pi 2</td>
<td>5040</td>
<td>5040</td>
<td>0m 19s</td>
<td>+0s</td>
</tr>
<tr>
<td>Fib 1</td>
<td>19605</td>
<td>19605</td>
<td>0m 17s</td>
<td>+0s</td>
</tr>
<tr>
<td>Fib 2</td>
<td>218243</td>
<td>218243</td>
<td>4m 18s</td>
<td>+1s</td>
</tr>
<tr>
<td>Dining 1</td>
<td>831</td>
<td>798</td>
<td>0m 3s</td>
<td>+0s</td>
</tr>
<tr>
<td>Dining 2</td>
<td>5852</td>
<td>5746</td>
<td>0m 14s</td>
<td>+1s</td>
</tr>
<tr>
<td>Dining 3</td>
<td>38787</td>
<td>36075</td>
<td>1m 28s</td>
<td>+5s</td>
</tr>
<tr>
<td>Dining 4</td>
<td>248012</td>
<td>205161</td>
<td>12m 55s</td>
<td>+1m 3s</td>
</tr>
<tr>
<td>Szymanski 1</td>
<td>65138</td>
<td>65138</td>
<td>2m 3s</td>
<td>+0s</td>
</tr>
<tr>
<td>Szymanski 2</td>
<td>82008</td>
<td>82008</td>
<td>2m 40s</td>
<td>+0s</td>
</tr>
<tr>
<td>Szymanski 3</td>
<td>132532</td>
<td>132532</td>
<td>6m 20s</td>
<td>+0s</td>
</tr>
<tr>
<td>Pairs</td>
<td>4096</td>
<td>8</td>
<td>0m 0s</td>
<td>+0s</td>
</tr>
<tr>
<td>Locking 1</td>
<td>2520</td>
<td>2520</td>
<td>0m 4s</td>
<td>+0s</td>
</tr>
<tr>
<td>Locking 2</td>
<td>22680</td>
<td>22680</td>
<td>0m 49s</td>
<td>+1s</td>
</tr>
<tr>
<td>Locking 3</td>
<td>113400</td>
<td>113400</td>
<td>4m 14s</td>
<td>+4s</td>
</tr>
<tr>
<td>Locking 4</td>
<td>-</td>
<td>-</td>
<td>(&gt; 30m)</td>
<td>-</td>
</tr>
<tr>
<td>Updater 1</td>
<td>33269</td>
<td>33269</td>
<td>2m 22s</td>
<td>+1s</td>
</tr>
<tr>
<td>Updater 2</td>
<td>33830</td>
<td>33504</td>
<td>2m 21s</td>
<td>+1s</td>
</tr>
<tr>
<td>Writes</td>
<td>-</td>
<td>-</td>
<td>(&gt; 30m)</td>
<td>-</td>
</tr>
<tr>
<td>Synthetic 1</td>
<td>1316</td>
<td>915</td>
<td>0m 2s</td>
<td>+1s</td>
</tr>
<tr>
<td>Synthetic 2</td>
<td>14969</td>
<td>8205</td>
<td>0m 38s</td>
<td>+4s</td>
</tr>
<tr>
<td>Synthetic 3</td>
<td>19942</td>
<td>11462</td>
<td>1m 6s</td>
<td>+4s</td>
</tr>
</tbody>
</table>

the original DPOR algorithm presented in [FG05] but not considerably so. Therefore an optimal DPOR algorithm would likely be slightly slower than the version used in these experiments in cases where the number of tests performed by our DPOR implementation is close to the number of Mazurkiewicz traces but on the other hand, significantly faster in the other cases.

Discussion of the results
The unfolding based approaches perform at most one execution per Mazurkiewicz trace and therefore it is natural that they never perform more tests than the DPOR algorithm. As a contextual unfolding of a program can be even exponentially more succinct than a place replication based unfolding (see Chapter 5.3), it is also not surprising that the contextual unfolding based algorithm requires the least number of test executions. The only case where it does not perform the least amount of tests is Dining 3. The reason for this is that the Dining benchmarks contain deadlocks and
due to the randomness in the test executions, the contextual unfolding algorithm has covered more of the deadlocks than the place replication based one. On average, however, both unfolding algorithms perform the same number of test runs in this particular benchmark.

The place replication based unfolding algorithm often performs the same number of tests as there are Mazurkiewicz traces in the program. This happens in benchmarks where the threads are tightly coupled. That is, all threads can affect the executions of other threads throughout the whole program. However, if the program under test contains reachable states after which some threads do not interact with each other anymore, the unfolding based algorithms can provide substantial reductions to the number of test executions and therefore to the runtime of the algorithm. This is well illustrated by the Filesystem and Pairs benchmarks. As discussed in Chapter 3, the difference to the number of Mazurkiewicz traces can even be exponential in such cases. The contextual unfolding based algorithm can provide additional reduction in cases where there are many read and write operations of multiple threads that are in data race. In programs with such data races, the use of read arcs allows their possible interleavings to be represented in a much more compact way allowing the testing algorithm to run less tests than there are Mazurkiewicz traces. This can lead to a substantial reduction to the number of test executions as illustrated by the Fib benchmarks that contain three threads and a large number of such races. However, when data races do not occur, contextual unfoldings do not provide additional reduction over the place replication based unfolding algorithm.

As discussed in the beginning of this chapter, the main concern regarding the new algorithms is whether they are computationally too expensive to be practical. Based on the experiments, this is not the case. In fact, in cases where all algorithms need to perform close to the same number of test executions, the place replication based unfolding algorithm offers similar performance as the DPOR algorithm. Furthermore, when different instances of the same benchmark (that increase in complexity) are compared, the unfolding based algorithms do not scale significantly worse than DPOR. This suggests that the optimized possible extensions algorithm can often in these benchmarks avoid searching the full unfolding for extensions. If this were not the case, the runtime of the algorithm would grow more quickly compared to DPOR that searches backtracks points by analyzing only the current test execution. There are, however, cases
where DPOR is clearly faster and scales better than the unfolding based approaches, such as the Szymanski benchmarks. In these benchmarks the possible extensions algorithm finds a large number of conditions that could form a preset of a possible extension event but the conditions are not concurrent.

When comparing the contextual and place replication based unfolding algorithms, the contextual unfolding algorithm is typically faster if it requires less test executions. However, when both algorithm need to perform close to the same number of test executions, the place replication approach is faster. This is due to the fact that finding possible extensions requires a large number of co-checks to be performed and the presence of read-arcs in contextual nets requires the co-check algorithm to detect contextual cycles which can slow down the algorithm.

The DPOR algorithm is guaranteed to detect all deadlocks. To make the comparison between the new algorithms and DPOR more fair for deadlock detection, the time needed to search deadlocks from the unfoldings needs to be taken into account. Based on the experimental results the additional overhead needed by the deadlock search algorithm is a modest one. Therefore the new algorithms are also feasible when the aim is to detect deadlocks as well.

As a summary, the results roughly speaking suggests that DPOR algorithm is a good fit for cases where the threads are tightly coupled. That is, all threads can affect the executions of other threads throughout the whole program. This is especially true for an optimal DPOR algorithm as non-optimal versions can sometimes perform large number of redundant tests, as illustrated by the Filesystem benchmarks. The place replication based unfolding algorithm, however, also offers very competitive performance when testing such programs.

For programs where some threads can become independent at some point of an execution (e.g., a point after which the threads access only different parts of the shared memory), the unfolding based algorithms can clearly outperform algorithms like DPOR. The contextual unfolding based algorithm, on the other hand, is a good fit for programs that contain a large number of data races. For such programs it can outperform the other two approaches by a clear margin. However, if there are no data races, the place replication based unfolding algorithm is likely to be faster.

Note also that the algorithms are primarily suitable for testing and not fully verifying programs. This is because for complex programs, stateless
algorithms like the ones used in the experiments, can generate so many test cases that exploring them all is not feasible. This true even for single threaded programs that are tested by using DSE. However, compared to random testing, the algorithms presented in this work avoid exploring equivalent execution paths multiple times and are also more likely to cover corner cases that are difficult to cover by purely random testing.

8.3.2 SMT Translation Based Checking of Global Properties

To evaluate the feasibility of the SMT translation based property checking approach discussed in Chapter 6, various global properties were checked on the benchmark programs. The results of this evaluation are shown in Table 8.2. The property column shows the global property being checked. The variables in the properties are shared variables in the programs. The SAT column describes whether the program contains a global state satisfying the property (SAT) or no such global state exists (UNSAT). The Unf column shows the time required by the constraint solver Z3 to solve the translation generated from a place replication based unfolding and similarly the last column shown the time required to solve the translation based on contextual unfoldings.

The results of the experiments are not as encouraging as would be hoped and therefore, for example, deadlocks were not checked with this approach for all of the benchmarks. Determining if a global shared state

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Property</th>
<th>SAT</th>
<th>Unf.</th>
<th>Contextual unf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updater</td>
<td>$x + y &gt; 200 \land y &lt; 100$</td>
<td>UNSAT</td>
<td>0m 49s</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Updater</td>
<td>$x + y &gt; 200$</td>
<td>SAT</td>
<td>0m 47s</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Synthetic 3</td>
<td>$i + j = 50 \land k = -32 \land i &gt; 152$</td>
<td>SAT</td>
<td>2m 2s</td>
<td>0m 46s</td>
</tr>
<tr>
<td>Fib 1</td>
<td>$i \geq 32 \lor j \geq 32$</td>
<td>SAT</td>
<td>0m 41s</td>
<td>0m 12s</td>
</tr>
<tr>
<td>Fib 1</td>
<td>$i \geq 144 \lor j \geq 144$</td>
<td>UNSAT</td>
<td>0m 39s</td>
<td>0m 38s</td>
</tr>
<tr>
<td>Fib 2</td>
<td>$i \geq 32 \lor j \geq 32$</td>
<td>SAT</td>
<td>4m 28s</td>
<td>2m 29s</td>
</tr>
<tr>
<td>Fib 2</td>
<td>$i \geq 144 \lor j \geq 144$</td>
<td>SAT</td>
<td>7m 2s</td>
<td>26m 18s</td>
</tr>
<tr>
<td>Fib 2</td>
<td>$i &gt; 144 \lor j &gt; 144$</td>
<td>UNSAT</td>
<td>7m 15s</td>
<td>29m 54s</td>
</tr>
<tr>
<td>Dining 2</td>
<td>deadlock</td>
<td>SAT</td>
<td>0m 17s</td>
<td>0m 19s</td>
</tr>
<tr>
<td>Dining 3</td>
<td>deadlock</td>
<td>SAT</td>
<td>9m 51s</td>
<td>12m 36s</td>
</tr>
<tr>
<td>Locking 2</td>
<td>deadlock</td>
<td>UNSAT</td>
<td>&gt; 30m</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Szymanski</td>
<td>deadlock</td>
<td>UNSAT</td>
<td>0m 49s</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Updater</td>
<td>deadlock</td>
<td>UNSAT</td>
<td>22m 37s</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Updater 2</td>
<td>deadlock</td>
<td>SAT</td>
<td>2m 15s</td>
<td>&gt; 30m</td>
</tr>
<tr>
<td>Synthetic 2</td>
<td>deadlock</td>
<td>SAT</td>
<td>0m 29s</td>
<td>0m 11s</td>
</tr>
</tbody>
</table>
satisfying a given property is reachable by using a SMT translation is feasible, but only for small programs, with the place replication based approach. However, the sizes of the SMT translations grow quickly as the complexity of the program under test grows. This generates more demanding SMT instances for the solvers. When using contextual unfoldings, the runtimes of the SMT solver are more unpredictable when compared to the regular unfolding case to make the approach practical. Note that these experiments were also repeated 10 times and different random seeds were given to the constraint solver. As shown by the experiments, sometimes it can be faster to check a property from a contextual unfolding based translation but sometimes it can be significantly slower. A possible explanation for this is that there are a large number of markings that satisfy the given global property but none of them is reachable from the initial state. In such cases the additional encoding to take contextual cycles into account makes it more demanding for the solver to determine which markings are reachable and which are not.

For checking deadlocks, the SMT translation approach is not practical. In many of the cases the deadlock checks take more time than generating the unfolding and searching all deadlocks by using the optimized backtrack search algorithm described in Chapter 6.3.

Based on the experimental results in the previous section, it is, however, possible to find deadlocks efficiently from an unfolding by taking advantage of the structure of the unfolding to optimize the search algorithm. Therefore it could be possible that also other global properties in addition to deadlocks can also be searched efficiently by using similar techniques. Investigating such search algorithms is left for future work.

8.3.3 Stateful Algorithms

As a final set of experiments, we have tested the benchmarks with the lightweight state capturing based testing algorithms described in Chapter 7. The results of these experiments are shown in Table 8.3. In these experiments the naive stateful testing algorithm has also been combined with sleep sets to further improve the approach. The table also includes the results of the place replication based unfolding algorithm (referred to as stateless unfolding in the table) from Chapter 3 to make it easier to compare the stateless and stateful approaches. The table shows the number of test executions needed to fully test the benchmarks and also the runtimes of the algorithms. As a sanity check for the implementations,
Table 8.3. Experimental Results of Stateful Algorithms

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Stateless Unfolding</th>
<th>Stateful Naive</th>
<th>Stateful Unfolding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Time</td>
<td>Tests</td>
</tr>
<tr>
<td>Filesystem(16)</td>
<td>3</td>
<td>0m 0s</td>
<td>-</td>
</tr>
<tr>
<td>Filesystem(18)</td>
<td>3</td>
<td>0m 0s</td>
<td>-</td>
</tr>
<tr>
<td>Parallel Pi 1</td>
<td>720</td>
<td>0m 2s</td>
<td>720</td>
</tr>
<tr>
<td>Parallel Pi 2</td>
<td>5040</td>
<td>0m 19s</td>
<td>-</td>
</tr>
<tr>
<td>Fib 1</td>
<td>19605</td>
<td>0m 17s</td>
<td>5746</td>
</tr>
<tr>
<td>Fib 2</td>
<td>218243</td>
<td>4m 18s</td>
<td>53478</td>
</tr>
<tr>
<td>Dining 1</td>
<td>798</td>
<td>0m 3s</td>
<td>3</td>
</tr>
<tr>
<td>Dining 2</td>
<td>5746</td>
<td>0m 14s</td>
<td>3</td>
</tr>
<tr>
<td>Dining 3</td>
<td>36095</td>
<td>1m 29s</td>
<td>2</td>
</tr>
<tr>
<td>Dining 4</td>
<td>205161</td>
<td>12m 55s</td>
<td>-</td>
</tr>
<tr>
<td>Szymanski 1</td>
<td>65138</td>
<td>2m 3s</td>
<td>50264</td>
</tr>
<tr>
<td>Szymanski 2</td>
<td>82008</td>
<td>2m 40s</td>
<td>64843</td>
</tr>
<tr>
<td>Szymanski 3</td>
<td>132532</td>
<td>6m 20s</td>
<td>105270</td>
</tr>
<tr>
<td>Pairs</td>
<td>8</td>
<td>0m 0s</td>
<td>9</td>
</tr>
<tr>
<td>Locking 1</td>
<td>2520</td>
<td>0m 4s</td>
<td>20</td>
</tr>
<tr>
<td>Locking 2</td>
<td>22680</td>
<td>0m 49s</td>
<td>29</td>
</tr>
<tr>
<td>Locking 3</td>
<td>113400</td>
<td>4m 14s</td>
<td>39</td>
</tr>
<tr>
<td>Locking 4</td>
<td>-</td>
<td>(&gt; 30m)</td>
<td>115</td>
</tr>
<tr>
<td>Updater 1</td>
<td>33269</td>
<td>2m 22s</td>
<td>13586</td>
</tr>
<tr>
<td>Updater 2</td>
<td>33504</td>
<td>2m 21s</td>
<td>13627</td>
</tr>
<tr>
<td>Writes</td>
<td>-</td>
<td>(&gt; 30m)</td>
<td>1</td>
</tr>
<tr>
<td>Synthetic 1</td>
<td>915</td>
<td>0m 2s</td>
<td>68</td>
</tr>
<tr>
<td>Synthetic 2</td>
<td>8205</td>
<td>0m 38s</td>
<td>123</td>
</tr>
<tr>
<td>Synthetic 3</td>
<td>11462</td>
<td>1m 6s</td>
<td>326</td>
</tr>
</tbody>
</table>

it was checked that both the naive and the stateful unfolding algorithms generates models of the program (i.e., Petri nets) that have the same size.

From the results it can be seen that the new stateful algorithms can in some cases greatly outperform the stateless testing approaches. There are two main reasons for this. The first one is that the Petri net model allows the testing algorithms in some cases to explore execution paths without performing concrete test executions. In other words, exploring some execution path by a concrete execution can be avoided altogether by exploring the execution path in the model instead. For the naive algorithm this is especially crucial as it explores interleavings of also independent operations. If one execution belonging to a specific Mazurkiewicz trace has been explored and therefore modeled in the Petri net, then all other linearizations of the same Mazurkiewicz trace can be explored from the model instead of concretely executing them. Furthermore, it is sometimes
possible to predict the states resulting from different interleavings of even
dependent operations from the model as described in Chapter 7.3. In the
experiments the algorithms benefit from this especially in the Dining and
Writes benchmarks.

The second reason why the new stateful algorithm can outperform the
stateless ones is naturally that they can sometimes avoid exploring the
same part of the state space multiple times by detecting that some execution
paths lead to the same state. In other words, the naive algorithm can
avoid exploring potentially huge subtrees of the symbolic execution tree
and similarly the stateful unfolding algorithm can generate a smaller pre-
fixes of unfoldings.

There are, however, cases where the stateless algorithms can outper-
form the stateful ones. For the naive algorithm this is especially true
for programs that contain a large number of threads. For example, the
Filesysten benchmarks contain close to 20 threads. In such programs
there are simply too many interleavings for the naive algorithm to ef-
efficiently explore even though it can explore most of them in the model
instead of using concrete test executions. This means that the naive algo-
ithm is suitable only for programs that contain a small number of con-
currently enabled threads.

The stateful unfolding algorithm does not have to explore all interleav-
ings of independent operations and therefore it scales better for programs
with a large number of threads. However, it is computationally more de-
manding than the stateless unfolding algorithm as it has to maintain the
Petri net model, store markings and sort the possible extensions accord-
ing the ERV-adequate order. This means that in cases where lightweight
state matching does not provide a significant reduction to the number
of test executions, the statefull unfolding algorithm is slower than the
stateless one. This can be well seen in the Parallel Pi and Szymanski
benchmarks. The difference, however, is not usually significant.

One disadvantage of the stateful approaches is that they require more
memory as the model and the visited markings need to be stored. The
amount of memory required to store an abstract state (i.e., a marking of
the Petri net model) depends directly on the number of shared variables
in the program under test. This is because there is a place for each share
variable in any reachable marking of the model. Therefore if the shared
state of the program is large, the new stateless approaches can require
considerable amounts of memory. Therefore the approaches are primar-
ily suitable for programs where the number of shared variables is small. Note also that when compared to storing full state information, storing the markings of the Petri net model can require considerably less memory in cases where most of the state of the program is in the local states of threads.
9. Conclusions

9.1 Summary

In this thesis we have studied the problem of how to automatically and systematically test multithreaded programs. One popular approach to automate the generation of test inputs for single threaded programs is to use dynamic symbolic execution (DSE). DSE can be seen as an approach that systematically explores different execution paths of a sequential program by covering its symbolic execution tree. There also exists earlier work where DSE has been used as a part of testing multithreaded programs. In these approaches DSE is used to limit the number of different input values that are used in the test executions and different partial order reduction methods are used to limit the number of thread interleavings that are explored. The earlier approaches combining DSE and partial order reductions have concentrated on testing all Mazurkiewicz traces of a given program. This allows errors such as assertion violations and deadlocks to be detected.

Explicitly covering all Mazurkiewicz traces, however, is not always necessary in order to detect specific types of errors. In this thesis we have investigated methods based on net unfoldings that allow us in some cases to reduce the number of test executions below the number of Mazurkiewicz traces and still be able to detect assertion violations and deadlocks. The reason for this is that unfoldings can succinctly represent all interleavings of threads in such a way that the unfolding can in some cases be covered with exponentially less test executions than there are Mazurkiewicz traces. This succinctness, however, comes with a price. It can be computationally expensive to compute from an unfolding which execution paths still need to be tested so that no errors are missed. Computing such infor-
The unfolding based testing algorithms developed in this thesis can avoid exploring all Mazurkiewicz traces in such programs where there are reachable states after which some threads become independent. This can in the best case lead to even an exponential reduction to the number of needed test executions when compared to the number of Mazurkiewicz traces. We have also developed an unfolding based testing algorithm that uses contextual unfoldings to succinctly represent concurrent reads and writes of shared variables. This allows contextual unfolding based testing algorithms to avoid a potentially large number of irrelevant test executions in programs with data races as contextual unfoldings can be exponentially more succinct than place replication based ones for such programs. Furthermore, the possible extensions computation is polynomial to the size of the unfolding even in the worst case with contextual unfoldings.

For the new testing algorithms we have shown that they cover all feasible control flow paths of individual threads. Therefore they can be used to check whether assertions can be violated in the program. We have also shown how deadlocks can be efficiently detected from the unfoldings generated by the algorithm. It is also possible to check for other global properties in addition to deadlocks from an unfolding of a program. In this thesis we have investigated the possibility of using SMT solvers as the search engine for finding states that satisfy a given property. Based on the experiments, such an approach, however, is feasible only for small programs.

To evaluate the feasibility of the new unfolding based testing approaches, we have compared them against an algorithm that combines DSE with DPOR. Such an algorithm covers all Mazurkiewicz traces in the program under test. The experiments show that using unfoldings in testing is a...
viable approach. More specifically, the new algorithms are competitive against DPOR based approaches which suggest that the possible extensions computation used in the algorithms is not too expensive for the unfolding based approaches to be practical. Furthermore, the new algorithms never explore the same Mazurkiewicz trace more than once and in cases where they are able to take advantage of independence between threads, they are considerably faster than approaches based on covering all Mazurkiewicz traces.

To further reduce the number of redundant test executions that the testing algorithms perform, we have investigated how abstract state information can be used to detect cases where even executions that belong to different Mazurkiewicz traces reach the same global state. As fully capturing and storing all visited states is often infeasible due to the amount of memory required for doing so, we have developed a more lightweight approach that in limited cases can determine that two non-equivalent executions reach the same state. Our approach also does not depend on potentially expensive constraint solver calls to determine if symbolic states subsume one another. Based on the experiments, even limited state information can sometimes greatly reduce the number of unnecessary test executions.

### 9.2 Future Work

There are multiple possible directions for future work. One direction is to investigate whether multithreaded programs can be represented even more succinctly without increasing the computational cost per test execution excessively. One potential approach for this would be to consider merged processes [KKKV05, RSK13].

Another direction is to investigate how unfolding based testing could be combined with verification methods that can prove that a state in the program is not reachable. For example, the algorithms developed in the Yogi-project [GHK+06, BNRS08, GNRT10] by Microsoft Research are promising approaches that combine DSE and abstraction refinement based techniques. It would be interesting to see if such approaches could be extended to multithreaded programs.

One possible way to speed up the testing of multithreaded programs is to parallelize the testing process. Based on our earlier findings [KLS+11, KSH13] and based on experiments on a distributed variant of DPOR pre-
Conclusions

Presented in [YCGK10], parallelization is a viable option for DSE and DPOR based algorithms. For unfolding based approaches, however, distributing the computation is more challenging as the unfolding is kept in memory. There exists earlier work on parallelization of Petri net unfolding algorithms [HKK02] but further research would be needed to determine to which extend unfolding based testing approaches could benefit from parallelization.

As already discussed in Chapter 7, the Petri net based state capturing approach could be extended in many different ways. One approach would be to take the valuations of shared variables into account. This potentially adds more opportunities to limit the number of test executions. Another approach would be to attempt to detect when a thread returns to an already explored local state. One way to do this is to consider how the local states change between transitions and capture the deltas as discussed in [YCGK08]. The state capturing could also be combined with different partial order reduction algorithms. Good candidates for this are the DPOR algorithm and contextual unfoldings with cut-offs [BBC+12].


[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In


[RZ08] Barbara G. Ryder and Andreas Zeller, editors. Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and


Automated Systematic Testing Methods for Multithreaded Programs

Kari Kähkönen