Quasi optimal model checking for concurrent systems

NGUYEN Thi Thanh Huyen

LIPN, CNRS UMR 7030, University Paris 13

Jury:

Benoît BARBOT	MCF, LACL, Université Paris Est Créteil	Member
Thomas CHATAIN	Ass. Prof, HdR, LSV, ENS Cachan	Reviewer
Camille COTI	Ass. Prof, LIPN, Université Paris 13	Co-advisor
Fabrice KORDON	Prof, LIP6, Sorbonne Université	Member
Laure PETRUCCI	Prof, LIPN, Université Paris 13	Director
Franck POMMEREAU	Prof, IBISC, Université d'Evry	Reviewer
César RODRÍGUEZ	Ass. Prof, LIPN, Université Paris 13	Co-advisor
Tayssir TOUILI	DR, CNRS, LIPN	President

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

Villetaneuse 2018

Acknowledgements

First of all, I would like to extend my great thanks to Fabrice Kordon, Benoît Barbot, Tayssir Touili for their acceptance to be members of the committee and Thomas Chatain, Franck Pommereau for their reviews of my thesis. I appreciate all their comments to make my thesis better.

I would like to express my great gratitude to my supervisor, Laure Petrucci, for all her guides and supports during the past over three years. From her, I have learned the serious working manner and attitude.

I am particularly grateful to César Rodríguez, my co-advisor, for the patience when he taught me unfolding, model checking and many other things from the scratch. I owe him for his kindness and carefulness to my work and my daily life. He was always a person I could come to when I had problems.

I would like to extend my deepest thanks to Camille Coti, another advisor who gave me plenty of guidances and suggestions to new interesting extent of parallelism. From my first day in France, she has been considerate and willing to help me whenever I need.

My special thanks are extended to Marcelo Sousa and Hoang-Gia Nguyen. Marcelo's cooperation plays an important part on the success of my research. Sharings and advices received from Hoang-Gia, not also a fellow doctoral student but also a brother as well, have been a great help in overcoming many problems during my research. I truly appreciate Gia's friendship and kindness.

Vietnam Ministry of Education and Training who funds my whole the study in France receives from me great thanks. My Ph.D study is also made possible thank to Hanoi University of Education, Vietnam, particularly the Faculty of Information Technology (FIT) where I have been working as a lecturer. I would like to thank my colleagues at FIT for their hard work to offer me the time for my study aboard.

I would like to thank all members of LIPN who have made my time in the laboratory remembering. My sincere thanks are towards to Brigitte Guevenuex who is so kind to help me with all complex administrative procedures from the first day in France and other hard time with accommodation searching. I would like to thank to fellow doctoral students in LIPN whose sharings help me a lot improve my presentation skills.

Outside LIPN, I would like to express my warming thanks to my vietnamese friends in University of Paris 13: Phuong, Viet, Nga, Hoang, Hieu, Thuy, Hoa, Thai, Thi, Tram, Quang and many others that I cannot list all. They really made my time in the university full of joys, funs and warmth. I will remember their considerateness and sharings as well as all travels and parties we made together.

My warmest appreciations goes to my parents, in-law parents, uncles, aunts, brothers and sisters who are always care of me and encourage me along my far-way living.

Finally, from the bottom of my heart, my love and gratitude go to my husband *Le Hoan* who is always by my side, encourages and supports me in the good as well as hard time of my life. My daughter Hoang-Ngan and my little son Hai-Phong are appreciated for their understanding and being good for me to throw myself in work.

Villetaneuse, December 2018

Huyen Nguyen

Abstract

Formal methods have been widely used for design and verification of complex systems. Among the proposals in the field, model checking gains more attractivity as it can perform automatically and give counterexamples when there is any defect. However, by exhaustively exploring all possible behaviours of the system, i.e. generating the state*state space*, model checking has to face the *state space explosion* problem.

Partial Order Reduction (POR) and unfolding are two of approaches that deal with state space explosion. PORs exploit the *commutativity* of concurrent transitions to reduce the state space while unfolding algorithms gain reduction by using a compact graph based on *true concurrency.*

We target the verification of concurrent programs which are a rich source of concurrency. Dynamic partial-order reduction (DPOR), a branch of POR, is a mature approach to mitigate the state explosion problem in stateless model checking of multithreaded programs based on Mazurkiewicz trace theory, whereas unfolding, which is originally for Petri nets, is still at an initial state for targetting programs.

We propose to combine DPOR and unfolding into an algorithm called Unfolding based POR that optimally explores the state space. The exploration is *optimal* when it explores each Mazurkiewicz trace only once. However, in order to obtain optimality, the algorithm is forced to compute sequences of transitions that avoid visiting a previously visited Mazurkiewicz trace known as *alternatives.*

In this thesis, we formally address the *cost of optimality* in a DPOR algorithm. We prove that computing alternatives in optimal DPOR, Unfolding based POR in particular, is an NP-complete problem.

As a trade-off solution between solving this NP-complete problem and potentially explore an exponential number of redundant schedules, we propose a hybrid approach called Quasi-Optimal POR (QPOR). In particular, we provide a new notion of k-partial alternative and a polynomial

algorithm to compute alternative executions that can arbitrarily approximate the optimal solution based on a user-defined constant k.

Finding k-partial alternatives requires decision procedures for the causality and conflict relations in the unfolding. Another main algorithmic contribution is to represent these relations as a set of trees where events are encoded as one or two nodes in two different trees. We show that checking causality and conflict between events amounts to an efficient traversal in one of these trees. We also use a skip list to quickly traverse the tree.

We also implement the algorithm and data structures in a new tool using a specialized data structure.

Besides the algorithmic improvements guaranteed by QPOR, parallelization is another way to speed up the unfolding exploration by taking advantage of multiple processor hardware and parallel models. Therefore, we propose a parallel algorithm based on parallelizing QPOR.

Finally, we conduct experiments on the sequential implementation of QPOR and compare the results with other state-of-art testing and verification tools to evaluate the efficiency of our algorithms. The analysis shows that our tool outperforms them.

Résumé

Les méthodes formelles sont utilisées largement pour la vérification de la conception de systèmes complexes. Parmi les solutions proposées par ce domaine, le *model checking*, ou vérification de modèles, suscite de l'intérêt car il s'effectue de manière automatique et fournit un contre-exemple en cas d'anomalie. Cependant, en effectuant une exploration exhaustive de tous les comportements possibles du système, c'est-à-dire de l'espace d'états, le *model checking* fait face au problème de l'explosion de cet espace d'états.

La réduction d'ordre partiel (POR) et le dépliage (*unfolding*) sont deux approches qui permettent de faire face à l'explosion de l'espace d'états. Les techniques de réduction d'ordre partiel exploitent la commutativité des transitions concurrentes pour réduire l'espace d'états, tandis que les algorithmes de dépliage le réduisent en utilisant un graphe compact basé sur la *vraie concurrence*. Notre but est de vérifier des programmes concurrents qui sont riches en sources de concurrence. La réduction d'ordre partiel dynamique (DPOR), qui est un type de réduction d'ordre partiel, est une approche mature pour réduire le problème de l'explosion de l'espace d'états dans le model checking sans état de programmes multithreadés en se basant sur la théorie des traces de Mazurkiewicz, tandis que les dépliages, conçus à l'origine pour les réseaux de Petri, sont encore une nouveauté pour les programmes. Nous avons proposé de combiner la DPOR et le dépliage dans un algorithme appelé *POR basée sur le dépliage* et qui effectue une exploration optimale de l'espace d'états. L'exploration est optimale quand elle explore chaque trace de Mazurkiewicz exactement une fois. Cependant, afin d'obtenir cette propriété d'optimalité, l'algorithme doit calculer des séquences de transitions qui permettent d'éviter d'explorer une trace de Mazurkiewicz déjà précédemment explorée : c'est ce que l'on appelle les alternatives. Dans cette thèse, nous abordons de manière formelle le coût de l'optimalité d'un algorithme de DPOR. Nous prouvons que le calcul des alternatives dans une DPOR optimale et, en particulier, dans l'algorithme de DPOR basée sur les dépliages, est un problème NP-complet.

En compromis entre la résolution d'un problème NP-complet et l'exploration potentielle d'un nombre exponentiel d'ordonnancements redondants, nous proposons une approche hybride appelée réduction d'ordre partiel quasi-optimale (QPOR). En particulier, nous proposons une nouvelle notion d'alternative k-partielle et un algorithme en temps polynomial qui calcule des exécutions alternatives qui peuvent approcher arbitrairement la solution optimale, paramétré par une constante k définie

par l'utilisateur. Trouver des alternatives k-partielles nécessite des procédures de décision pour les relations de causalité et de conflit dans le dépliage. Une autre contribution algorithmique de cette thèse est la représentation de ces relations comme un ensemble d'arbres dans lequel les événements sont encodés comme un ou deux nœuds dans deux arbres différents. Nous montrons que vérifier la causalité et le conflit entre deux événements revient à une traversée efficace d'un des deux arbres. Nous utilisons également une *skip list* pour traverser rapidement l'arbre. Nous détaillons l'implémentation de l'algorithme et les structures de données utilisées dans un nouvel outil qui utilise une structure de données spécialisée. Outre les améliorations algorithmiques garanties par QPOR, la parallélisation est un autre moyen d'accélérer l'exploration en tirant parti des avantages des architectures matérielles multi-processeurs et multi-cœurs grâce aux modèles de parallélisme. Par conséquent, nous proposons un algorithme de QPOR parallèle.

Enfin, nous présentons des expériences sur l'implémentation séquentielle de QPOR et comparons les résultats avec d'autres outils de test et de vérification à la pointe de l'état de l'art afin d'évaluer l'efficacité de nos algorithmes. L'analyse des résultats montre que notre outil présente de meilleures performances que ceux-ci.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Formal methods 2

1.3 Model checking 3

1.3.1 Model checking process 3

1.3.2 State Space Explosion (SSE) problem 4

1.3.3 Unfolding 5

1.3.4 Partial order reduction 6

1.3.5 Challenges and objectives 8

1.4 Parallel and distributed verification 9

1.4.1 Parallel and distributed hardware and software 9

1.4.2 Parallel and distributed verification 10

1.4.3 Opportunities and objectives 10

1.5 Contributions 11

1.6 Thesis outline 12

2 Preliminaries **13**

2.1 Introduction 13

2.2 Computation model 14

2.2.1 Labelled transition systems 14

2.2.2 Concurrent systems 15

2.3 Dependence and Independence Relation 18

2.3.1 Independence for Petri Nets 19

2.3.2 Independence for concurrent programs 19

2.4 Labelled Prime Event Structures 20

2.4.1 Definition 20

2.4.2 Configurations . . . 21
2.4.3 Extensions . . . 22
2.5 Unfolding semantics of program . . . 22
2.6 Conclusions . . . 25

3 Quasi Optimal Partial Order Reduction **27**
3.1 Introduction . . . 27
3.2 Unfolding-based POR . . . 28
3.2.1 Unfolding exploration algorithm . . . 28
3.2.2 Algorithm correctness . . . 32
3.3 Partial alternatives . . . 35
3.3.1 Complexity of computing alternatives . . . 35
3.3.2 Motivating example . . . 39
3.3.3 k-partial alternatives . . . 41
3.4 Conflicting extensions . . . 43
3.4.1 Conflicting extension algorithm . . . 43
3.4.2 Complexity . . . 44
3.5 Conflict and Causality . . . 47
3.6 Sequential tree . . . 50
3.6.1 Causality and Conflict of nodes . . . 50
3.6.2 Data structure and efficient tree navigation . . . 52
3.6.3 Causality and Conflict for events . . . 52
3.7 Conclusions . . . 53

4 QPOR Parallelization **55**
4.1 Introduction . . . 55
4.2 Motivations . . . 56
4.2.1 Technology for parallelism . . . 56
4.2.2 Challenges and opportunities for QPOR parallelization . . . 56
4.2.3 Our objectives . . . 58
4.3 Parallelization design for QPOR . . . 58
4.3.1 Parallel computing . . . 58
4.3.2 General idea . . . 59
4.3.3 Data structure . . . 59
4.3.4 Overall algorithm . . . 61
4.3.5 Parallel exploration process . . . 62
4.3.6 Avoiding redundant exploration . . . 64

4.3.7 Algorithm termination 64
4.3.8 Synchronization mechanism 65
4.4 Conclusions 65

5 Implementation and experiments **67**
5.1 Introduction 67
5.2 DPU - Dynamic Program Unfolder 68
5.2.1 Front-end 69
5.2.2 Back-end 69
5.3 Sequential implementation 71
5.4 Parallel implementation 72
5.4.1 OpenMP 74
5.4.2 Algorithm implementation 76
5.5 Experiments 77
5.5.1 Comparison to SDPOR 77
5.5.2 Evaluation of the Tree-based Algorithms 79
5.5.3 Evaluation Against the State-of-the-art on System Code . . . 81
5.5.4 Profiling a Stateless POR 83
5.6 Conclusions 83

6 Conclusions and Perspectives **85**
6.1 Conclusions 85
6.2 Perspectives 86

Bibliography **89**

List of Figures

2.1 A Petri net . . . 15
2.2 A multithreaded program . . . 17
2.3 A labelled prime event structure . . . 20

3.1 Unfolding example. (a): a program P; (b): its unfolding semantics $\mathcal{U}_{P,\diamondsuit_P}$. . . . 31
3.2 Example of encoding a 3-SAT formula. . . . 37
3.3 Program unfolding encoding a 3-SAT formula. . . . 39
3.4 Motivating example. (a): Programs; (b): Partially-ordered executions; 40
3.5 Petri Net encoding a 3-SAT formula. . . . 46
3.6 Multiple trees for a process. a^i_j is node indexed j at the depth i . . . 50

4.1 Example: (a) Program; (b) All maximal configurations; (c) Unfolding 57
4.2 Parallel exploration . . . 60

5.1 DPU architecture . . . 68
5.2 Unfolding memory alignment . . . 70
5.3 Dpu class diagram for sequential implementation . . . 71
5.4 OpenMP execution model . . . 74
5.5 Execution of tasks . . . 75
5.6 Part of class diagram for parallel implemention . . . 76

List of Tables

5.1 Comparing QPOR and SDPOR. Machine: Linux, Intel Xeon 2.4GHz. TO: timeout after 8 min. Columns are: Th: nr. of threads; Confs: maximal configurations; Time in seconds, Memory in MB; SSB: Sleep-set blocked executions. N/A: analysis with lower k yielded 0 SSBs. . 78
5.2 (a), (b): depths of variable/thread trees; (c), (d): frequency of depth distances on causality/conflict queries. 80
5.3 Comparing DPU with Maple (same machine). LOC: lines of code; Execs: nr. of executions; R: safe or unsafe. Other columns as before. Timeout: 8 min. 82

Chapter 1

Introduction

Contents

1.1 Introduction **1**
1.2 Formal methods **2**
1.3 Model checking **3**
1.3.1 Model checking process 3
1.3.2 State Space Explosion (SSE) problem 4
1.3.3 Unfolding 5
1.3.4 Partial order reduction 6
1.3.5 Challenges and objectives 8
1.4 Parallel and distributed verification **9**
1.4.1 Parallel and distributed hardware and software 9
1.4.2 Parallel and distributed verification 10
1.4.3 Opportunities and objectives 10
1.5 Contributions **11**
1.6 Thesis outline **12**

1.1 Introduction

Complex systems require intensive verification and validation to assure reliable operations in the real world. Design is a critical part that determines system quality, design verification is hence an important work. Design verification should start early in the development process, thus help reduce significantly the cost and efforts to detect and fix bugs. Many design verification techniques have been proposed such as *testing* [91], *simulation* [5], *deductive verification* [50, 23] and *formal methods* [25, 34, 101]. Simulation works on an abstract model of the design while testing works on the actual

product, but these two conventional techniques have the same paradigm: providing prepared inputs and then observing and analyzing the outputs. They may be cost-efficient but risk to leave some fatal errors unfound. Moreover, working on manually designed and selected test cases and data, testing cannot certify there is no bug or how many bugs still remain in the software. *Deductive verification* [50] exploits the axioms and rules to prove the correctness of system, but it requires mathematicians or logic experts to manually construct proofs, so the verification process cannot be automatic. *Formal methods* are based on mathematical models to verify systems. Contrary to testing and simulations, the answer of formal methods is always sound since it relies on mathematical foundation. When a system model is declared correct by formal verification, it does mean that: the system is proper and there is no bugs left with a certain specification of the expected behaviours. This chapter will give introduction to formal methods and a short survey of techniques that have been proposed over the years in the field.

1.2 Formal methods

Formal methods [101, 103] are based on mathematical analysis to enhance the reliability and robustness of system design. It is described as the application of theoretical computer science fundamentals such as logic calculus [52], formal languages, automata theory [62], etc. in software specification and verification. Among the variety of recent proposed approaches, theorem provers, proof checkers and model checking have attracted many researches.

Theorem provers [70, 40] and *proof checkers* [13] require expert knowledge in mathematics to prove a mathematical model to be correct. Moreover, proof construction must be done by hand and require expertise that cannot be easily found in development team, it is hence time-consuming and effort-costly. Model checking is a good candidate for solving most of the aforementioned problems.

This method is based on *exhaustive exploration* of all possible behaviours of system. Compared with others, it has several strengths:

- *Automatic:* As long as model and specification are provided, the verification process is performed automatically. It is almost a push-button technique where users just provide input data, then get the answer while all the process is a black-box to them. Hence, it is also called *automated verification*.

- *Counterexample given:* When detecting an unsatisfied property, model checkers are able to provide the *counterexample* and even the source of the errors.

How does model checking work? In the context of automatic verification for concurrent programs, we discuss in the following sections the principles of the model checking method and provide a short overview of its various techniques. We also point out some shortcomings of existing model checking approaches and then propose our improvements.

1.3 Model checking

Model checking [25, 34] is automated verification that means the checking process can be done without users' manipulation. The main idea of model checking is to explore a directed graph representing all the possible behaviours of a transition systems, called *state space*, to find out reachability errors, such as deadlock, violations, etc. We witness plenty of model checking tools like SPIN [59], Divine [19], Uppaal [97], ITS-tools [93], etc.

By exhaustive exploration of the state space, model checking insures the complete coverage of system behaviours and produces reliable and sound output, unlike testing. But also arises the problem called *state space explosion* [37], which is the situation when the number of states to be explored increases combinatorially. The question is how to transform an actual system to a transition one, how to explore the state space efficiently and how to mitigate *state space explosion*, etc. This section details the model checking process and a survey of several techniques proposed recently to deal with SSE.

1.3.1 Model checking process

The process of verifying a system with model checking includes three steps:

- Modeling: This is a process of investigation and abstraction from the actual system into an acceptable formalism for model checkers. One of the most popular representations is *transition systems* that will be discussed in detail in Chapter 2. The aim of modeling is trying to reduce the number of computations and limit the computational resources to obtain a model as simple as possible but still representing completely the critical properties of the system.

- Specification: Relevant properties are extracted from system design and described in a suitable logic formalism, usually *temporal logic* [55, 33] (for instance, Linear Temporal Logic, Computation Tree Logic) to keep track of time in system operations. This stage also discards irrelevant details, so that verification determines if the model satisfies the specification but is often impossible to tell if the specification covers all necessary properties or not. It is essential for designer to insure the completeness and soundness of the specification.

- Verification: Verification is exploring algorithmically the system model to check if specified properties are satisfied or not. If it results in an error, counterexamples should be given to trace back to the source, which probably corresponds to bugs in the system itself or in the specification. Sometimes, we possibly get a time-out meaning that there is something wrong in the system or algorithms. It is then necessary to adjust parameters or review model and do experiments again to get a clear answer.

1.3.2 State Space Explosion (SSE) problem

Exploring all reachable states of a system guarantees the full coverage of the system behaviours but makes *state space explosion* [35, 37] happen as well. Assume that we have a system composed of m sequential components, each of which can be in n possible states, so the system reaches $m \times n$ states in total. In another case, if these m components are independent (concurrent and non-communicating), the state space will be calculated by n^m, exponential in m. It is obvious that even a moderate concurrent system can produce an enormous state space, leading to a problem called *state space explosion*, *SSE* for short. As a consequence, naive model checking may take a huge amount of time for small and simple systems, so that it limits the range of systems. Approaches recently proposed to tackle SSE can be divided into three main groups:

- *Reduction techniques* or Partial Order Reduction (POR) family [83, 51, 27, 16]: The main idea of POR methods is trying to reduce the number of states to explore and transitions to execute by exploiting commutativity or independence relations. We can avoid exploring interleavingly independent transitions which leads to an exponential reduction in the state space.

- *Compression* [53]: Compressing methods aim at constructing a compact representation for the reachability graph. Binary Decision Diagrams have been used

in [30, 31, 32] for a clever encoding of state spaces which exhibit some kind of regularity. The compact graph is obtained by removing redundant nodes and merging equal branches and subtrees. Symbolic execution [82, 68] can be seen as a compression approach constructing representation by partitioning the input values into equivalence classes such that some symbolic values can present all possible values. Similarly, unfolding [42, 43, 44, 87] is based on concurrency to construct a symbolic representation of possible interleavings of the original system.

- *Abstraction* [36, 90]: These approaches compute an abstract system that is finite and relatively small but still infer to the properties of the actual system. These involve transition or reactive systems that have data paths. The abstraction is generally specified by mapping a set of actual data values to a smaller set of abstract values, so that we apply it to states and transitions in transition systems to get a much reduced abstract system for more simple verification. Predicate abstraction is a typical instance of abstraction methods.

Each approach has its own pros and cons, we are interested in POR and unfolding. POR methods make use of independence relation to greatly reduce the number of states explored as well as transitions to fire, hence, gain exponentially compact state space while the unfolding method proposes an efficient representation for the state space. Moreover, unfolding is a partial order based technique, then it is convenient to combine it with POR. In the light of these findings, we can combine these two approaches to better tackle the SSE problem, and aim at dealing with a wider range of systems. We will discuss in details these two techniques in the following sections.

1.3.3 Unfolding

The unfolding [58, 45] method is a POR technique dealing with the state space explosion problem and based on the theory of *true concurrency* to replace full labeled transition systems by special partially ordered graphs.

Although their nodes are not actually reachable states themselves, these graphs contain full information about the reachable states of the system through *configuration*, a set of conflict-free and causal-closed events.

Unfolding was proposed by Nielsen [79] originally applied to Petri nets [4, 39] where a net is unfolded into an infinite acyclic *occurrence net*. It stops when there is no transition to fire. This process basically relies on the dependence relation (called

causality) where a transition can only fire if all its predecessors with respect to causality have already fired. The unfolding procedure starts at the initial state (marking), finds all possible enabled transitions whose inputs places hold tokens, then extends the graph by adding an instance of an enabled transition if it has no conflict in the local configuration (the configuration leads to a transition occurrence). With all these notions, the unfolding of a net is easy to construct but it is still infinite and has a lot of repeated parts. In his doctoral thesis [75], McMillan proposed to construct a finite graph called *prefix*, an initial part of the unfolding that covers all reachable markings of the net. With the introduction of prefix, the size of the graph associated to a Petri net is greatly reduced, thus simplifies the exploration. The unfolding procedure terminates at *cutoff points* that are a set of transitions such that: any configuration containing a cutoff point must be equivalent in term of final state to a configuration containing no cutoff points.

Although the proposal of cutoff points dramatically reduces the size of the unfolding, prefixes still have redundant information and sometimes McMillan's algorithm produces such prefixes that are much larger than necessary. In [45], Ezparza proposes an algorithm that generates a minimal prefix that is usually smaller than that in McMillan's one by using an adequate order. On the other hand, unfolding traditionally developed as partial order semantics for Petri nets has been used for analysis multithreaded programs [90, 87, 66]. However, existing unfolding based technique has also some disadvantages that will be addressed in Section 1.3.5.

1.3.4 Partial order reduction

As mentioned above, concurrency is one of the causes of state space explosion. Partial order reduction [84, 85] are best suited for asynchronous systems where concurrent actions must be interleaved i.e. at a time only one transition is executed. This approach exploits the *commutativity* of concurrent transitions to avoid exploring all orderings of independent statements, so that the state space reduction is achieved. At every state, only a sufficient subset of transitions enabled at that state is executed. To apply POR, firstly, we construct a reduced graph based on the original system. In fact, it is unnecessary to build the full graph of reachable states because the reduced one can represent complete behaviours of system.

The reduced graph is generated by selecting one representative for an equivalence class, so-called a *Mazukiewicz trace*. Two or more *interleaving sequences* that start from a state and return in a single state are considered equivalent and POR only takes one as representative, thus remarkably reduces the state space.

The partial order reduction approach [84, 54] is based on an observation that the arbitrary interleaving of concurrent actions returns the same state. Consequently, the identification of concurrency between two actions a and b, i.e. "a can occur before b and b can occur before a", greatly contributes to solve the state explosion problem. A family of POR methods has been proposed including persistent set, sleep set, unfolding and many other extended POR methods.

Persistent set methods: These methods [54] try to reduce the number of states explored by computing a persistent set at every state. At a state s, the persistent set is a subset T of the transitions enabled at s such that any transition not in T is independent with all transitions in T, i.e. no transition outside T is able to interact or affect those inside. Exploring only transitions in the persistent set of each state is sufficient to detect all reachability errors, such as deadlock and assertion violations [54]. Persistent sets can be implemented by stubborn sets and ample sets.

Stubborn set: This approach basically works on a subset of successors that can fire at a state, exploiting the commutativity between concurrent transitions in a concurrent system [98]. Roughly speaking, a stubborn set is a closed set of transitions with respect to mutual interactions that means all transitions intervening with the transitions in the set must belong to the set. At every state, a stubborn set, a subset of the enabled set, is computed and only those transitions in this set will be chosen to fire for the next steps.

Ample set: At every state, an ample set [54] is computed using static analysis. The most important condition in the method is that if a transition a is dependent on transition b which is in the ample set, a should not be invoked until b fires.

It is sufficient to detect all deadlocks if they exists in the reduced state space.

Sleep set This technique [51] aims at reducing transitions fired by exploiting the dependence between transitions and its *past* of the exploration, based on the idea: if two transitions are independent and their firings in any order lead to the same state, we can sufficiently take one of these sequences. Therefore, at every state s, a subset of the enabled set is computed such that transitions in this subset will not be invoked at s.

Extended partial order reduction methods *Dynamic Partial Order Reduction*, DPOR for short, extends classic POR in the sense that all subsets such as persistent set and sleep set are computed *dynamically* [51]. On the way of exploring a path, the

algorithm simultaneously keeps track of relations between threads to identify backtracking points that might lead to other execution traces that are not "equivalent" to the current one. It is guaranteed that all deadlocks and failures of the system are detected when the exploration finishes. Many extensions of the original DPOR have been proposed including the use of vector clocks [80], Monotonic POR [67], TransDPOR [92], source set DPOR [2] and unfolding based POR [87].

We point out in this thesis two DPORs: source set based DPOR which is considered the state-of-art non-optimal DPOR [2] and unfolding based POR [87, 88] as one of the state-of-art Optimal DPORs. Source set based DPOR presents a new DPOR algorithm which is optimal in the sense that it always explores the minimal number of executions based on a novel class of sets, called source sets. Source set replace the role of persistent sets in previous algorithms and is smaller but sufficient to guarantee exploration of all Mazurkiewicz traces [74]. Moreover, those events in the persistent set but not in the source set will initiate sleep-set blocked explorations in the future which we should avoid as much as possible. They also extend the algorithm with a novel mechanism, called wakeup trees, that allows to achieve optimality.

For optimality, unfolding based POR [87] use the concept *alternatives* to imply the sequences of actions that avoid visiting a previously visited Mazurkiewicz trace. Computing these sequences thus amounts to deciding whether the DPOR needs to visit yet another Mazurkiewicz trace (or all have already been seen).

However, both these DPORs still have algorithmic shortcomings. They have to solve an NP-complete problem when computing wakeup trees or alternative executions (another execution rather than the current one at a node of exploration tree) during the state space exploration. In the meanwhile, for some programs, SDPOR, expected to explore polynomial executions, will explore $\mathcal{O}(2^n)$ while optimal POR explore $\mathcal{O}(n)$ executions.

1.3.5 Challenges and objectives

Despite intensive improvements in model checking methods over the years, both POR and unfolding still suffer from some challenges as follows:

1. Computing alternatives or wakeup tree in optimal DPOR algorithms is a NP-complete problem.

2. Many states are visited repeatedly, making DPOR unoptimal.

3. Both optimal DPOR of [87] and [1] leave the problem of computing alternatives open. Efficient algorithms for deciding causality and conflict which are essential in computing alternatives are also required.

4. Current stateless POR algorithms have not efficiently exploited memory for fast memory accesses.

The aforementioned are challenges but opportunities for computer scientists for a better verification. We see that partial order reduction or unfolding alone has not been able to sufficiently cope with the state space explosion problem. Our objectives in this thesis are to obtain a significant reduction of the state space. To target the objectives, our approaches are as follows:

- Algorithm: We aim to combine partial order reduction and unfolding to achieve an algorithm that explores the state space efficiently.

- Implementation: We aim to find a data structure to organize unfolding in memory in an efficient way. Together with combined algorithm, we aim at implementing a tool available to the public.

1.4 Parallel and distributed verification

Users demand information systems to run faster and faster while the power of hardware is physically limited. Increasing physical power of computers is truly difficult, so instead of making more powerful processors, people try to have more processors in a computer system to increase its computational power. Thanks to multiple processor computers, parallel and distributed computing have gained attention from researchers as well. Parallel/distributed verification is to use parallel/distributed computing to perform the verification. The success of parallel/distributed verification depends on the development of hardware and software engineering as well as the verifying algorithms.

1.4.1 Parallel and distributed hardware and software

The past decades witness a rapid development of hardware and software so that parallel and distributed computers become more and more familiar. Parallel computers are those that have multiple cores (processing units) or multiple functional units (Graphic Processing Unit - GPU) on a single chip. Distributed computers, so-called supercomputers, even have multiple CPUs in different locations and each of them

may have several cores. That provide them extremely huge computational power. Having several independent processors allows new-generation computers to execute their works concurrently. Software programming needs to take advantages of this feature to speed up the execution. Parallel and distributed computing are models that use multiple resources of computer systems to perform a computational work simultaneously to reduce the execution time. To apply parallel/distributed computing, the computation problem must be broken into separate sub-problems that can be executed concurrently and each of the sub-problems is solved by a processing unit. The partition of work should be done in such a way that they are as independent as possible to avoid communication overhead. A communication mechanism, e.g. shared memory or message passing, also needs to be established to control the interaction among processing units.

1.4.2 Parallel and distributed verification

Automated verification (model checking) verifies a system by exploring its state space. For large scale systems, most of which are concurrent as well, the state space is extremely enormous, so the exploration is surely a huge computation. Beside researches conducted to reduce the state space, parallelism is one of the solutions to speed up the exploration using multiple processing units. Distributing the work over several nodes and cores increases the computational power, so that the execution time decreases.

We have witnessed a plenty of researches on parallel model checking algorithms such as parallel nested depth-first search (NDFS) algorithms [69, 47, 71], parallel BFS-based LTL model checking [102], Cartesian POR [57], POR for GPU [94, 7] and distributed work such as distributed POR [29, 105, 89], etc. We also witness some parallelization in the field of unfolding such as [18, 49] These works have obtained significant speedup in state space exploration.

On the other hand, many other works parallelize existing sequential model checkers to achieve parallel or distributed one with better performance. We have seen SPIN [59, 60] and distributed SPIN [21, 61] Helena [46] and multi-core Helena [48], distributed Helena [38], IMITATOR [10] and distributed IMITATOR [11, 12]. These distributed model checkers have obtained better performance than the original ones, so it is reasonable to parallelize a sequential tool.

1.4.3 Opportunities and objectives

In respect to parallelism, we can see opportunities as follows:

- Parallel and distributed computers and programming models are available for developers to exploit to gain speedup.

- As the nature of unfolding, branches are independent i.e. it is possible to explore them concurrently, in other words, an unfolding is naturally suitable to be parallelized.

These opportunities mentioned above urge us to design and implement a parallel/distributed model checker. Our objectives are:

- To algorithm aspect: In addition to the efficiency achieved by the algorithm, we aim at parallelizing the algorithm to enforce its strength in execution time. We would design a parallel algorithm that partitions the exploration into sub-works that can be executed concurrently.

- To implementation aspect: In scope of this research, we target to use a parallel programming model to implement proposed parallel algorithm.

We believe that parallelism at both algorithm and implementation levels will gain expected speedup.

1.5 Contributions

In our thesis, we address the verification of multithreaded C/C++ programs which is practical for software industry but still in initial state of research. Our main contributions are:

- We prove that computing alternative in optimal exploration is an NP-complete problem for both Petri nets and concurrent programs. This is the first formal proof of complexity in the literature of DPOR.

- We propose a trade-off solution called Quasi Optimal Partial Order Reduction (QPOR) with a new notion of partial alternative that can be computed in polynomial time (addressing the challenge 1) and explore less redundant schedules than non-optimal DPOR (addressing the challenge 2). This is the main result in our paper published in 30th International Conference on Computer Aided Verification in 2018 (CAV'18) [77].

- We also propose an efficient data structure to decide causality and conflict (addressing the challenge 3).

- We propose a parallel algorithm for unfolding exploration by parallelizing the sequential one based on the relative independence between branches in an unfolding. That is the prerequisite for parallel implementation.
- We implement the sequential algorithm in a new tool Dpu with a specific data structure addressing challenge 4. Parallel implementation is in experimental level.

Finally, we conducted experiments on a collection of benchmarks using sequential implementation to illustrate the efficiency of our algorithms and tools compared to some other testing and verification tools.

1.6 Thesis outline

The following parts of this thesis are organized as follows:

- Chapter 2 provides preliminary concepts necessary for the following chapters.
- Chapter 3 presents our algorithm QPOR combining partial order reduction and unfolding to achieve a quasi optimal exploration in verifying multithreaded C programs. This chapter also details the new concept of k-partial alternative for computing alternatives and an efficient data structure to compute structural relations between events.
- Chapter 4 discusses the challenges and opportunities for parallelization underlying the nature of the exploration. We present in detail a new parallel algorithm achieved by parallelizing the sequential QPOR.
- Chapter 5 provides details of both sequential and parallel implementation of corresponding algorithms. Experimental results on selected benchmarks are also shown together with evaluations of the efficiency of proposed algorithms against some state-of-art testing and verification tools.
- Finally, we summarize our contributions in the thesis and present works in perspective.

Chapter 2

Preliminaries

Contents

2.1 Introduction **13**
2.2 Computation model **14**
2.2.1 Labelled transition systems 14
2.2.2 Concurrent systems 15
2.3 Dependence and Independence Relation **18**
2.3.1 Independence for Petri Nets 19
2.3.2 Independence for concurrent programs 19
2.4 Labelled Prime Event Structures **20**
2.4.1 Definition 20
2.4.2 Configurations 21
2.4.3 Extensions 22
2.5 Unfolding semantics of program **22**
2.6 Conclusions **25**

2.1 Introduction

This chapter provides some basic notions that are necessary for the rest of the thesis. First, we mention two main models of behaviour, Petri nets and multithreaded programs as concurrent systems, and their suitable representation for formal verification of programs. Finally, the unfolding semantics of concurrent programs is presented.

2.2 Computation model

Model checkers do not take real programs as input but finite state systems, so we need abstraction to provide a suitable notation and divide the state space into manageable segments.

2.2.1 Labelled transition systems

Automated verification for concurrent systems is based on the state space exploration of their associated transition systems. A *transition system* is a graph composed of reachable states of the system as nodes and transitions between states as edges. The original system in terms of Petri nets or concurrent programs needs to be transformed into a transition system whose nodes and edges are labelled with states and actions of the original models.

Generally, a *labelled transition system* (LTS) is a structure $M := \langle \Sigma, \rightarrow, A, s_0 \rangle$, where Σ is the set of *states*, A is the set of *actions*, $\rightarrow \subseteq \Sigma \times A \times \Sigma$ is the transition relation, and s_0 is the *initial state*.

Firing actions and Enabledness If $s \xrightarrow{a} s'$ is a transition, the action a is *enabled* at s, and a can *fire* at s to produce s'. Let $enabl(s)$ denote the set of actions enabled at s. A sequence $\sigma := a_1 \dots a_n \in A^*$ is a *run* when there are states $s_1, \dots, s_n$ satisfying $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$. For such a sequence σ, we define $state(\sigma) := s_n$. We let $runs(M)$ denote the set of all runs of M, and $reach(M) := \{state(\sigma) \in \Sigma \colon \sigma \in runs(M)\}$ the set of all *reachable states* of M.

Determinism An LTS is *deterministic* if at every state s, at a time, only one transition can fire, e.g. a, and there exists a state s' such that $s \xrightarrow{a} s'$, then s' is unique. All actions are also deterministic that means firing an action produces only one state s'.

Local transition When studying the behaviour of the whole system, we are interested in communication actions that affect the state of global variables, e.g. actions on shared memory in multithreaded programs. *Local transitions*, denoted by a type *local*, are those that affect merely local variables, hence do not have any influence on other processes.

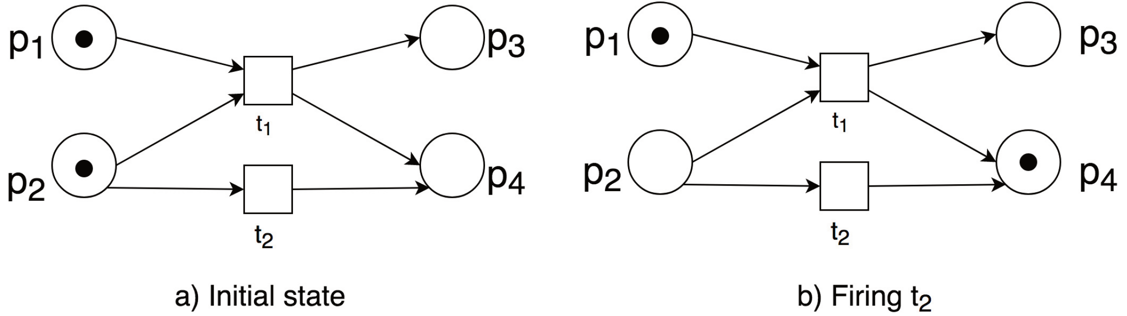


Figure 2.1: A Petri net

2.2.2 Concurrent systems

There are various models of system behaviour, we mention here two of them: Petri nets and concurrent programs.

Petri nets A Petri net [76] is a model of a concurrent system. It is a specification language to model system behaviour based on a strong mathematical foundation. It is well suited to model concurrency and synchronization in distributed systems.

Definition 1. *A* Petri net *is a tuple* $N := \langle P, T, F, m_0 \rangle$*, where* P *and* T *are disjoint finite sets of* places *and* transitions*,* $F \subseteq (P \times T) \cup (T \times P)$ *is the* flow relation*, and* $m_0 \colon P \to \mathbb{N}$ *is the* initial marking.

Roughly speaking, a Petri net is a diagram consisting of places (denoted by circles), transitions (denoted by rectangles) and arcs for connecting places to transitions and vice versa. We illustrate a simple Petri net $N := \langle P, T, F, m_0 \rangle$ in Fig. 2.1 where a set of places $P = \{p_1, p_2, p_3, p_4\}$, set of transitions $T = \{t_1, t_2\}$, the flow relation $F = \{\langle p_1, t_1 \rangle, \langle p_2, t_1 \rangle, \langle t_1, p_3 \rangle, \langle t_1, p_4 \rangle, \langle p_2, t_2 \rangle, \langle t_2, p_4 \rangle\}$ and the initial marking $m_0 = (p_1 = 1, p_2 = 1, p_3 = 0, p_4 = 0)$.

A Petri net N is called *finite* if P and T are finite. The unfolding approach is applicable for finite nets. For $x \in P \cup T$, let ${}^\bullet x := \{y \in P \cup T \colon (y, x) \in F\}$ be the *preset* - the input, and $x^\bullet := \{y \in P \cup T \colon (x, y) \in F\}$ the *postset*, the output, of x. In the example, ${}^\bullet t_1 = \{p_1, p_2\}$ and $t_1^\bullet = \{p_3, p_4\}$. The state of a net is represented by a marking. A *marking* of N is a function $m \colon P \to \mathbb{N}$ that assigns *tokens* to every place. Graphically, places in a Petri net may contain a number of marks called tokens, a unity of some resource of the system. In Fig. 2.1, p_1, p_2 hold one token, p_3, p_4 do not have any token. The maximum number of tokens a place can hold decides the level of safety of a Petri net. For example, a 1-safe Petri net is a net in which each place can only contain 0 or 1 token. The net in Figure 2.1 is 1-safe as places holds at most

1 token. Any distribution of tokens over the places will represent a configuration of the net, a marking which is a state of the Petri net. A transition t is *enabled* at a marking m iff for any $p \in {}^\bullet t$ we have $m(p) \geq 1$.

Firing a transition consists in consuming tokens from its input places and producing tokens in its output places. Hence, a transition can only be fired, called *enabled*, at a marking m if there are enough tokens in its input places. t is *enabled* iff $\forall p \in {}^\bullet t : m(p) \geq 1$ (we work on 1-safe Petri nets). In the example, both t_1 and t_2 are enabled at the initial state (Figure 2.1.a). Firing a transition produces a new reachable marking corresponding to a reachable state of the system. Firing t_1 produces the marking in Figure 2.1.b.

We give semantics to nets using transition systems. We associate N with a transition system $M_N := \langle \Sigma, \rightarrow, A, m_0 \rangle$ where $\Sigma := P \rightarrow \mathbb{N}$ is the set of markings, $A := T$ is the set of transitions, and $\rightarrow \subseteq \Sigma \times A \times \Sigma$ contains a triple $m \xrightarrow{t} m'$ exactly when, for any $p \in {}^\bullet t$ we have $m(p) \geq 1$, and for any $p \in P$ we have $m'(p) = m(p) - |\{p\} \cap {}^\bullet t| + |\{p\} \cap t^\bullet|$. We call N k-safe when for any reachable marking $m \in \mathit{reach}(M_N)$ we have $m(p) \leq k$, for $p \in P$.

Concurrent programs Concurrent programs are programs composed of a number of threads or processes that are simultaneously executable. Process and threads in a concurrent program communicate with each other via message passing, shared memory or both. Each thread is a finite sequence of program statements. Statements touching shared variables are *visible*, otherwise they are considered to be *invisible*. Concurrent programs are often non-deterministic because at an execution step, there might be multiple concurrent actions that can take place at the next step. We are interested in data deterministic (in the sense that with certain input data, it produces a unique output) and shared memory concurrent programs. Moreover, threads are assumed data race free implying that no action or thread in a single process accesses the shared memory concurrently and all threads are synchronized using mutexes.

Definition 2. *A concurrent program is a structure $P := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0 \rangle$, where $\mathcal{M}$ is the set of* memory states *(values of program variables, including instruction pointers), $\mathcal{L}$ is the set of* mutexes, *m_0 is the* initial state of memory, *l_0 is the* initial state of mutexes *and T is the set of* thread states*.*

We define a corresponding structure $P_1 := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0 \rangle$ for the program in Figure 2.2 as follows:

- Set of memory (variables) $\mathcal{M} = \{x, y, z\}$

```
|Thread0       Thread1     Thread2   |
|-----------------------------------|
| x := 0       lock(m)     lock(m')  |
| lock(m)      y := 1      z := 3    |
| if (y == 0) unlock(m)   unlock(m')|
| unlock(m)                          |
| else                               |
| lock(m')                           |
| z := 2                             |
|                                    |
```

Figure 2.2: A multithreaded program

- Set of mutexes $\mathcal{L} = \{m, m'\}$
- Initial state of memory $m_0 = \{x = 0, y = 0, z = 0\}$
- Initial state of mutexes $l_0 = \{m = 0, m' = 0\}$
- Pool of threads $T = \{Thread0, Thread1, Thread2\}$

We now define a *LTS* $M_P := \langle \mathcal{S}, \rightarrow, A, s_0 \rangle$ of given program P. Let $\mathcal{S} := \mathcal{M} \times (\mathcal{L} \rightarrow \{0, 1\})$ be the set of *states* of program P, where each *state* $s \in S$ is a pair of the form $\langle m, v \rangle$ where m is the state of the memory and v indicates when a mutex is locked (1) or unlocked (0). Actions in $A \subseteq \mathbb{N} \times \Lambda$ are pairs $\langle p, b \rangle$ where p is the identifier of the thread that executes some statement and b is the effect of the statement.

The relation $\rightarrow$ contains a triple $\langle m, v \rangle \xrightarrow{\langle p,b \rangle} \langle m', v' \rangle$ exactly when there is some thread statement $\langle p, f \rangle \in T$ such that $f(m) = \langle m', b \rangle$ and either

1. $b = \texttt{local}$ and $v' = v$, or
2. $b = \langle \texttt{lock}, x \rangle$ and $v(x) = 0$ and $v' = v_{|x \mapsto 1}$, or
3. $b = \langle \texttt{unlock}, x \rangle$ and $v' = v_{|x \mapsto 0}$.

The initial state is $s_0 := \langle m_0, l_0 \rangle$. We use the function $p \colon A \rightarrow \mathbb{N}$ to retrieve the thread identifier associated with an action.

The previous conditions ensure that a `local` transition does not change the mutexes values and model the behaviour of the lock and unlock operations. Note that they also ensure that the set of enabled transitions cannot contain two transitions

whose effect is a `lock` and a `unlock` over the same mutex. Also note that it is possible for two actions enabled at a state to be labelled with the same function. We assume the function f is computable in polynomial time.

We define a LTS $M_{P_1} := \langle \mathcal{S}, \to, A, s_0 \rangle$ for program P as follows:

- $\mathcal{S} = \{\langle x = y = z = 0, m = 0, m' = 0 \rangle, \langle x = 0, y = 1, z = 3, m = 1, m' = 1 \rangle, ..\}$
- $A = \{\langle 0, local \rangle, \langle 1, lock_m \rangle, \langle 2, lock_{m'} \rangle, ..\}$
- $\to = \{\langle x = y = z = 0, m = m' = 0 \rangle \xrightarrow{\langle 0, lock_m \rangle} \langle x = y = z = 0, m = m' = 0 \rangle, ..\}$
- $s_0 = \langle m_0, l_0 \rangle = \langle x = 0, y = 0, z = 0, m = 0, m' = 0 \rangle$

Furthermore, if $s \xrightarrow{a} s'$ is a transition, the action a is *enabled* at s. Let $enabl(s)$ denote the set of actions enabled at s. A sequence $\sigma := a_1 \dots a_n \in A^*$ is a *run* when there are states $s_1, \dots, s_n$ satisfying $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$. We define $state(\sigma) := s_n$. We let $runs(M_P)$ denote the set of all runs and $reach(M_P) := \{state(\sigma) \in \mathcal{S} \colon \sigma \in runs(M_P)\}$ the set of all *reachable states*.

2.3 Dependence and Independence Relation

Partial-order reduction methods are described as model checking selecting representatives from equivalence class of behaviours. Commutativity is exploited to identify these equivalence classes, and then eliminate unnecessary interleavings as much as possible. We recall a standard notion of commutativity [54].

Two actions a, a' of a transition system M are considered to *commute* at a state s iff

- if $a \in enabl(s)$ and $s \xrightarrow{a} s'$, then $a' \in enabl(s)$ iff $a' \in enabl(s')$; and
- if $a, a' \in enabl(s)$, then there is a state s' such that $s \xrightarrow{a.a'} s'$ and $s \xrightarrow{a'.a} s'$.

where *enabl(s)* is the set of transitions enabled at state s and $s \xrightarrow{\sigma} s'$ is firing a sequence of transitions σ at state s to get to the next state s'.

Independence between transitions is an underapproximation of commutativity. A relation $\diamondsuit \subseteq A \times A$ is an *independence* for M if it is symmetric, irreflexive, and satisfies that every $(a, a') \in \diamondsuit$ commutes at every reachable state of M. In general, M has multiple independence relations, forming a set I; clearly $\emptyset$ is always one of them. If $(a, a') \notin I$, they are *dependent*.

2.3.1 Independence for Petri Nets

We specify the independence relation for a Petri Net via the dependence among transitions. For a Petri net N, the dependence relation of N, denoted $\otimes_N$, is defined as follows:

Given two transitions t and t',

$$t \otimes_N t' \text{ iff } (t^\bullet \cap {}^\bullet t' \neq \emptyset) \ \text{ or } \ (t'^\bullet \cap {}^\bullet t \neq \emptyset) \ \text{ or } \ ({}^\bullet t' \cap {}^\bullet t \neq \emptyset),$$

where ${}^\bullet t$ and $t^\bullet$ are respectively the *preset* and *postset* of t.

The dependence relation in Petri nets is fixed, hence, the independence relation, denoted $\diamondsuit_N$, given by the complement of dependence relation $\otimes_N$ defined above is also unique and fixed.

In the example in Figure 2.1, $T = \{t_1, t_2\}$ but ${}^\bullet t_1 \cap {}^\bullet t_2 = p_2 \neq \emptyset$, so $t_1 \otimes t_2$, that means $\diamondsuit_N = \emptyset$.

2.3.2 Independence for concurrent programs

In contrast to classic unfolding of Petri nets, the unfolding for a program is parametric which means it depends on the independence relation defined by model checkers. Given a program $P := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0 \rangle$ and its associated transition system $M_P := \langle \mathcal{S}, \rightarrow, A, s_0 \rangle$, we now define the independence relation $\diamondsuit_P$ on the set of actions A. First, let a thread transition $t := \langle p, f \rangle$ be a pair where $p \in \mathbb{N}$ is the *thread identifier* associated with the thread and $f : \mathcal{M} \rightarrow (\mathcal{M} \times \Lambda)$ is a *partial* function that models the transformation of the memory as well as the *effect* $\Lambda := \{\texttt{local}\} \cup (\{\texttt{lock}, \texttt{unlock}\} \times \mathcal{L})$ of the state with respect to thread synchronisation. `local` models code of a program thread that cannot potentially affect the enabledness of actions in another thread with respect to mutexes. $\langle \texttt{lock}, x \rangle$ or $\langle \texttt{unlock}, x \rangle$ model lock and unlock of a mutex x. We assume the function f to be decidable in polynomial time. The independence relation $\diamondsuit_P$ is defined as follows.

Definition 3. *The relation $\diamondsuit_P$ is the largest irreflexive, symmetric relation where for two arbitrary actions $a := \langle p, b \rangle$ and $a' := \langle p', b' \rangle$ in A, $a \diamondsuit_P a'$ if $p \neq p'$ and*

1. *$b =$ `local` or*
2. *$b =$ `lock` (x) and $b' \notin \{$`lock` (x), `unlock` $(x)\}$.*

$\diamondsuit_P$ identifies a set of actions whose complement is the dependence relation. In the example with program P_1, $(\langle 0, \texttt{lock}\ m \rangle, \langle 1, \texttt{lock}\ m' \rangle)$ or $(\langle 0, \mathit{local} \rangle, \langle 1, \texttt{lock}\ m' \rangle)$ are part of independence relation.

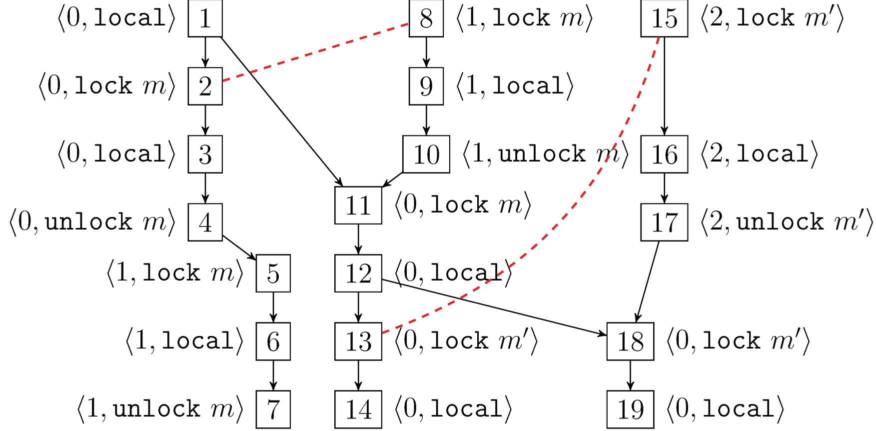


Figure 2.3: A labelled prime event structure

Definition 4. *A program P is* well-formed *when relation $\diamondsuit_P$ is an independence relation on M_P.*

In this thesis, we assume that programs are *well-formed*, i.e. the modeling of program statements via the statement functions is consistent with the labels generated by the function.

2.4 Labelled Prime Event Structures

Standard dynamic partial order reduction methods are based on the interleaving semantics of *LTS* M_P, i.e. they operate over executions of the program. We will operate over a non-interleaving, partial order based semantics known as labelled prime event structure. Intuitively, a prime event structure is a structure that compactly represents a set of partial orders.

2.4.1 Definition

Let X be a set. An *X-labelled prime event structure* [78] (X-LPES, or PES in short) is a tuple $\mathcal{E} := \langle E, <, \#, h \rangle$ where $< \subseteq E \times E$ is a strict partial order, $\# \subseteq E \times E$ is a symmetric, irreflexive relation, and $h \colon E \to X$ is a labelling function satisfying:

- for all $e \in E$, $\{e' \in E \colon e' < e\}$ is finite, and (1)
- for all $e, e', e'' \in E$, if $e \mathrel{\#} e'$ and $e' < e''$, then $e \mathrel{\#} e''$. (2)

We refer to elements of E as *events*, $<$ as *causality* and $\#$ as *conflict*. The conflict respects the causality, i.e. if $e_1 \mathrel{\#} e_2$ and $e_2 < e_3$, then $e_1 \mathrel{\#} e_3$. Figure 2.3

illustrates a PES where events are boxes with numbers which are their identifiers, causality is denoted by arrows and conflict is denoted by red dashed lines. Set of events $E = \{1, 2.., 19\}$, labeling function h is such that $h(6) = \langle 1, local \rangle$. It is obvious that $2 < 3$ and $8 \#^i 2$, then $3 \# 8$.

History The *history* of an event $e \in E$ is the unique set $\lceil e \rceil := \{e' \in E \colon e' < e\}$ representing the set of events that must not be fired before firing e. For example, $\lceil 11 \rceil = \{1, 8, 9, 10\}$ and $\lceil 5 \rceil = \{1, 2, 3, 4\}$.

2.4.2 Configurations

The notion of state in a PES $\mathcal{E}$ is captured by the concept of *configuration*. A configuration of $\mathcal{E}$ is any finite set $C \subseteq E$ satisfying:

- (causally closed): for all $e \in C$ we have $\lceil e \rceil \subseteq C$; (3)
- (conflict free) for all $e, e' \in C$, it holds that $\neg(e \# e')$. (4)

Let $conf(\mathcal{E})$ denote the set of all configurations of $\mathcal{E}$. Given a configuration C, we define the *interleavings* of C as $inter(C) := \{h(e_1), \ldots, h(e_n)\}$ such that $\forall e_i, e_j \in C : e_i < e_j \implies i < j$. An interleaving corresponds to the sequence labelling any topological sorting (sequentialisation) of the events in the configuration. We say that $\mathcal{E}$ is finite iff E is finite. $state(C)$ denotes state reached by executing any interleaving of C.

Maximal configuration: A maximal configuration is a configuration where no transition is enabled at its state. Each maximal configuration corresponds to a run or an execution of a program. A PES has a fixed number of maximal configurations. In Figure 2.3, the set $\{1, 3, 4, 5, 6, 7\}$ is an $\subseteq$-maximal configuration.

Local configuration: A local configuration is a configuration associated with an event which implies the *history* of the event including itself. For any $e \in E$, the *local configuration* of e is defined as $[e] := \lceil e \rceil \cup \{e\}$. In Figure 2.3, the local configuration of 4 is $\{1, 2, 3, 4\}$.

Immediate conflict: Two events e, e' are in *immediate conflict*, $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup [e']$ and $[e] \cup \lceil e' \rceil$ are configurations. In the example in Figure 2.3, we can see $2 \#^i 8$ and $13 \#^i 15$ depicted by a dashed line. For each event e in the PES, we also define a set of direct conflicting events $\#^i(e) := \{e' \in E \colon e \#^i e'\}$.

2.4.3 Extensions

Given a configuration C, the *extensions* of C, written $ex(C)$, are all those events outside C whose causes are included in C. Formally, we define:

$$ex(C) := \{e \in E \colon e \notin C \wedge \lceil e \rceil \subseteq C\}$$

We let $en(C)$ denote the set of events *enabled* by C, which intuitively correspond to the transitions enabled at $state(C)$, defined as:

$$en(C) := \{e \in ex(C) \colon C \cup \{e\} \in conf(\mathcal{E})\}$$

Conflicting extensions $cex(C)$ are events that are enabled at some subconfiguration $C' \subset C$ but conflicts with at least one event in $C \setminus C'$. Formally, we define:

$$cex(C) := \{e \in ex(C) \colon \exists e' \in C, e \#^i e'\}$$

Clearly, all events in $ex(C)$ which are not in $en(C)$ are in $cex(C)$ and $en(C)$ and $cex(C)$ partition the set $ex(C)$. For example, in Figure 2.3, $C = \{1, 2\}$ then $ex(C) = \{3, 8\}$ where $en(C) = \{3\}$ and $cex(C) = \{8\}$ as $2 \#^i 8$.

Event structures are naturally (partially) ordered by a *prefix* relation $\trianglelefteq$. Given two PESs $\mathcal{E} := \langle E, <, \#, h\rangle$ and $\mathcal{E}' := \langle E', <', \#', h'\rangle$, we say that $\mathcal{E}$ is a *prefix* of $\mathcal{E}'$, written $\mathcal{E} \trianglelefteq \mathcal{E}'$, when $E \subseteq E'$, $<$ and $\#$ are the projections of $<'$ and $\#'$ on E, and $E \supseteq \{e' \in E' \colon e' < e \wedge e \in E\}$. The set of prefixes of a given PES $\mathcal{E}$ equipped with $\trianglelefteq$ forms a complete lattice.

2.5 Unfolding semantics of program

Unfolding is originally a partial order semantics for a Petri net constructed by exploiting concurrency. Thanks to the similarity between programs and Petri nets in terms of concurrency, we can give semantics to concurrent programs using labelled transition systems. On the other hand, as a partial order technique, the unfolding technique first needs to define the independence relation over the program.

Given a program $P := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0\rangle$, we first identify a LTS $M_P := \langle \mathcal{S}, \rightarrow, A, s_0\rangle$ and the independence relation as in Definition 3. Assume that a well-defined relation $\diamondsuit_P$ is an independence relation on M_P, i.e. the modeling of program statements via the statement functions is consistent with the labels generated by the function.

We use, in this thesis, labelled prime event structures (LPES or PES for short) to represent unfolding semantics. Building the unfolding $\mathcal{U}_{M,\diamondsuit_P}$ for model M based on independence relation $\diamondsuit_P$ is a process of identifying events whose canonical name

of the form $e := \langle a, H \rangle$, where $a \in A$ is an action of M and H is a configuration of $\mathcal{U}_{M,\diamondsuit}$. Intuitively, e represents the action a after the *history* (or the causes) $H := \lceil e \rceil$.

Definition 5 (Histories of an action)**.** *The set $\mathcal{H}_{\mathcal{E},\diamondsuit,t}$ of candidate* histories *for an action a in a* PES *is the maximal set of configurations H of $\mathcal{E}$ such that:*

- *action a is enabled at state(H), and*
- *either $H = \{\bot\}$ or all $<$-maximal events e in H satisfy that $h(e) \otimes a$,*

where h is the labeling function in $\mathcal{E}$.

Once an event e has been identified, its associated transition $h(e)$ may be dependent with $h(e')$ for some e' already present and outside the history of e. Since the order of occurrence of e and e' matters, we prevent their occurrence within the same configuration by introducing a conflict between e and e'.

Definition 6 (Conflicting set)**.** *The set $\mathcal{K}_{\mathcal{E},\diamondsuit,e}$ of* events conflicting *with $e := \langle a, H \rangle$ is the maximal set of events e' in $\mathcal{E}$ such that $e' \notin [e]$ and $e \notin [e']$ and $a \otimes h(e')$.*

The definition of the unfolding $\mathcal{U}_{M,\diamondsuit}$ is inductive. The base case inserts into the unfolding a special *bottom event* $\bot$ on which every event causally depends. The inductive case extends the unfolding with one event. The unfolding is the $\trianglelefteq$-maximal element in the collection of unfolding prefixes generated by the inductive procedure of Definition 7. Therefore, the process of defining unfolding includes two steps: firstly, constructing the set of unfolding prefixes and then, defining the unfolding is the maximal prefix with respect to the $\trianglelefteq$ relation.

Definition 7 (Finite unfolding prefixes)**.** *The set of* finite unfolding prefixes *of M under the independence relation $\diamondsuit$ is the smallest set of PESs that satisfies the following conditions:*

1. *The PES having exactly one event $\bot$, empty causality and conflict relations, and $h(\bot) := \varepsilon$ is an unfolding prefix.*
2. *Let $\mathcal{E}$ be an unfolding prefix containing a history $H \in \mathcal{H}_{\mathcal{E},\diamondsuit,t}$ for some action $a \in A$. The extension of $\mathcal{E}$ with a new event $e := \langle a, H \rangle$ is the LES $\langle E, <, \#, h \rangle$ satisfying:*
    - *for all $e' \in H$, we have $e' < e$;*
    - *for all $e' \in \mathcal{K}_{\mathcal{E},\diamondsuit,e}$, we have $e \# e'$; and $h(e) := a$.*

Intuitively, each unfolding prefix contains the dependence graph (configuration) of one or more executions of M (of finite length). The unfolding starts from $\bot$, the "root" of the tree, and then iteratively adds events enabled by some configuration until saturation, i.e. when no more events can be added. Observe that the number of unfolding prefixes is finite if and only if all runs of M terminate.

The actual unfolding is infinite, but it is sufficient to consider it equivalent to a finite unfolding prefix that covers all reachable states of the system. We define the unfolding $\mathcal{U}_{M,\diamondsuit}$ as in Definition 8.

Definition 8. *The* unfolding $\mathcal{U}_{M,\diamondsuit}$ *is the* unique $\trianglelefteq$*-maximal element in the set of unfolding prefixes of* M *under* $\diamondsuit$*.*

Theorem 1. *The unfolding* $\mathcal{U}_{M,\diamondsuit}$ *exists and is unique. Furthermore, for any non-empty run* σ *of* M*, there exists a unique configuration* C *of* $\mathcal{U}_{M,\diamondsuit}$*, such that* $\sigma \in inter(C)$*.*

Proof. Let F be the set of all, finite or infinite, unfolding prefixes of $\mathcal{U}_{M,\diamondsuit}$. We have that $\mathcal{U}_{M,\diamondsuit} = union(F)$ is an unfolding prefix, $\trianglelefteq$-maximal and unique. Observe that for a run that fires no transition, i.e. $\sigma = \varepsilon \in T^*$ we may find the empty configuration $\varnothing$ or the configuration $\{\bot\}$, and in both cases σ is an interleaving of the configuration. Hence the restriction to non-empty runs. Assume that σ fires at least one transition. The proof is by induction on the length $|\sigma|$ of the run.

Base Case: If σ fires a transition t, then t is enabled at s, the initial state of M. Then, $\{\bot\}$ is a history for t, as necessarily $state(\{\bot\})$ enables t. This means that $e := \langle t, \{\bot\}\rangle$ is an event of $\mathcal{U}_{M,\diamondsuit}$ and clearly $\sigma \in inter(\{\bot, e\})$. It is easy to see that no other event e' different than e but such that $h(e) = h(e')$ can exist in $\mathcal{U}_{M,\diamondsuit}$ and satisfy that the history $[e']$ of e' equals the singleton $\{\bot\}$. The representative configuration for σ is therefore unique.

Inductive Step: Consider $\sigma = \sigma'.t_{k+1}$ with $\sigma' = t_1.t_2..t_k$. By the induction hypothesis, we assume that there exists a unique configuration C' such that $\sigma' \in inter(C')$. All runs in $inter(C')$ reach the same state s and σ' is such a run, t_{k+1} is hence enabled at state s. If there is a $<$-maximal event $e \in max(C')$ such that $h(e)$ interferes with t_{k+1}, then C' is a valid configuration H and by construction (second condition of Definition 5) there is a configuration $C = C' \cup \{e'\}$ with $e' = \langle t_{k+1}, H\rangle$. Otherwise, we construct a valid H by considering subconfigurations of C' removing a maximal event $e \in max(C')$ such that $h(e)$ does not interfere with t_{k+1}. We always reach a valid H since C' is a finite set and $\{\bot\}$ is always a valid H. Considering $C = H \cup \{e'\}$ with $e = \langle t, H\rangle$, by construction (second condition of Definition 5) we have that

$\forall e_H \in H : \neg (e \# e_H)$ and $\forall e_{H'} \in C' \setminus H : \neg (e \# e_{H'})$ (otherwise these events would be in H). Hence, $C \cup \{e\}$ is a configuration. □

2.6 Conclusions

In this chapter, we have provided notions and definitions necessary to transform a concurrent program into a labelled transition system, a representation suitable for exploration by a POR algorithm. Having all these premilinaries, the next chapter will detail our main contribution that is an algorithm for constructing as well as exploring unfolding of concurrent programs called Quasi Optimal Partial Order Reduction.

Chapter 3

Quasi Optimal Partial Order Reduction

Contents

3.1 Introduction **27**

3.2 Unfolding-based POR **28**

3.2.1 Unfolding exploration algorithm 28

3.2.2 Algorithm correctness 32

3.3 Partial alternatives **35**

3.3.1 Complexity of computing alternatives 35

3.3.2 Motivating example 39

3.3.3 k-partial alternatives 41

3.4 Conflicting extensions **43**

3.4.1 Conflicting extension algorithm 43

3.4.2 Complexity 44

3.5 Conflict and Causality **47**

3.6 Sequential tree **50**

3.6.1 Causality and Conflict of nodes 50

3.6.2 Data structure and efficient tree navigation 52

3.6.3 Causality and Conflict for events 52

3.7 Conclusions **53**

3.1 Introduction

POR techniques have gained great success in coping with SSE by exploiting *commutativity* and/or *independence relation* to reduce the state space to explore. On

the other hand, with labelled event structures, the unfolding technique provides a compact acyclic representation to program executions. Therefore, combining these two techniques results in an optimal algorithm that explores exactly each execution of the program once. Despite of the optimality, unfolding based POR still faces the challenge of solving a hard problem of computing alternatives. Several proposals including new notions, data structures are introduced to improve this algorithm. In this chapter, we focus on our contributions as follows:

- We detail in Section 3.2 the unfolding based algorithm to verify a concurrent program and provide our proofs for the algorithm soundness and termination.
- We also prove that computing alternatives in optimal exploration for both Petri nets and concurrent programs is NP-complete.
- We present a new concept of *partial alternative* and additional data structures as a trade-off solution between optimal and non-optimal DPORs in Section 3.3.
- Procedures and data structures to efficiently compute structural relations, causality and conflict, are also provided in Sections 3.5 and 3.6.

3.2 Unfolding-based POR

3.2.1 Unfolding exploration algorithm

In this section, we present a DPOR algorithm that does not explore sequential executions in PES. Instead, it considers only one single configuration at a time and organises the exploration into a binary tree where each deadlocking execution corresponds to a path from the root of the tree to some leaf [87]. The tree is explored using a depth-first search. During the exploration, when it is unable to move on the tree, the algorithm backtracks to a node n and it needs to decide whether another execution stemming from n shall be explored. We call such an execution an *alternative*, and refer to the decision procedure for finding whether such an alternative execution exists as *computing alternatives*.

The POR algorithm to explore the unfolding is described in Algorithm 1.

`Explore`(C, D, A) uses a set U that represents all events already discovered throughout the exploration. The parameters passed to the function include:

- a configuration C which has already been explored.

Algorithm 1: An unfolding-based POR exploration algorithm.

```
1  Initially, set U := {⊥}, and call Explore({⊥}, ∅, ∅).
2  Procedure Explore(C, D, A)
3      Add ex(C) to U
4      if en(C) = ∅  return
5      if ena(C) = ∅  return
6      if A = ∅
7          Choose e from ena(C)
8      else
9          Choose e from A ∩ ena(C)
10     Explore(C ∪ {e}, D, A \ {e})
11     if there exists a J ∈ Alt(C, D ∪ {e})
12         Explore(C, D ∪ {e}, J \ C)
```

- a set of events D (for *disabled*) that represents the conflicts which can be used to avoid repeated explorations in `Explore()`.
- a set of events A (for *add*) which is used to partially navigate the exploration in order to justify an event in D.

The key intuition in Algorithm 1 is to visit all maximal configurations of $\mathcal{U}$ which contain C and do not contain D; and it should always include A.

In detail, the algorithm first updates U with all extensions (including enabled and conflicting extensions) of C (Line 3). If C is a maximal configuration, i.e. there is no event enabled, then there is nothing to do and it backtracks (Line 4). Otherwise, it picks an enabled event that is available $\texttt{ena}(C) := en(C) \setminus D$. If there is no available event, the algorithm cannot progress in the exploration, then it has to backtrack (Line 5). This situation is known in the literature of PORs (that use sleep sets) as a *sleep-set blocked exploration* (SSB). Otherwise, if A is empty, any enabled event available can be taken (Line 7). If not, A needs to be explored and e must come from the intersection (Line 9). Then, it makes a recursive call (left subtree), where it explores *all* configurations containing all events in $C \cup \{e\}$ and no event from D. Since $\texttt{Explore}(C, D, A)$ visited all maximal configurations containing C and e, it remains to visit those containing C but not e. Thus, we determine whether $\mathcal{U}$ has a maximal configuration that contains C and does not contain $D \cup \{e\}$.

At Line 11 we determine if any such configuration exists. Function `Alt` returns a set of configurations, so-called *clues*. A clue is a witness that a $\subseteq$-maximal configuration exists in $\mathcal{U}_{P,\diamondsuit}$ which contains C and not $D \cup \{e\}$.

Definition 9 (Clue)**.** *Let D and U be sets of events, and C a configuration such that $C \cap D = \emptyset$. A* clue *to D after C in U is a configuration $J \subseteq U$ such that $C \cup J$ is a configuration and $D \cap J = \emptyset$.*

Definition 10 (`Alt` function)**.** *Function `Alt` denotes* any *function such that `Alt`(B, F) returns a set of clues to F after B in U, and the set is non-empty if $\mathcal{U}_{P,\diamond}$ has at least one maximal configuration C where $B \subseteq C$ and $C \cap F = \emptyset$.*

When `Alt` returns a clue J, the clue is passed in the second recursive call (Line 12) to "mark the way" (using set A) in the subsequent recursive calls at Line 10, and guide the exploration towards the maximal configuration that J witnesses. Definition 10 does not identify a concrete implementation of `Alt`. It rather indicates how to implement `Alt` so that Algorithm 1 terminates and is complete (see below). Different PORs in the literature can be reframed in terms of Algorithm 1. SDPOR [2] uses clues that mark the way with only one event ahead ($|J \setminus C| = 1$) and can hit SSBs. Optimal DPORs [2, 87] use size-varying clues that guide the exploration provably guaranteeing that any SSB will be avoided.

Algorithm 1 is *optimal* when it does not explore a SSB. To make Algorithm 1 optimal `Alt` needs to return clues that are *alternatives* [87], which satisfy stronger constraints. When that happens, Algorithm 1 is equivalent to the DPOR in [87] and becomes optimal (see [88] for a proof).

Definition 11 (Alternative [87])**.** *Let D and U be sets of events and C a configuration such that $C \cap D = \emptyset$. An* alternative *to D after C in U is a clue J to D after C in U such that $\forall e \in D : \exists e' \in J$, $e \# e'$.*

The simple example in Figure 3.1.(a) shows a concurrent program $P := \langle \mathcal{M}, \mathcal{L}, T, m_0, l_0 \rangle$ where the set of mutexes $\mathcal{L} = \{m, m'\}$ are initially *unlocked*. T is the set of three threads $Thread_0$, $Thread_1$ and $Thread_2$ communicating via two mutexes m and m'. The associated LTS with $\mathcal{P}$ is $M_P := \langle \mathcal{S}, \rightarrow, \mathcal{A}, s_0 \rangle$ with set of actions $\mathcal{A}$:

$\mathcal{A} = \{\langle 0, \texttt{lock}\ m \rangle, \langle 0, \texttt{unlock}\ m \rangle, \langle 0, \texttt{lock}\ m' \rangle, \langle 1, \texttt{lock}\ m \rangle, \langle 1, \texttt{unlock}\ m \rangle,$
$\langle 2, \texttt{lock}\ m' \rangle, \langle 2, \texttt{unlock}\ m \rangle, \langle 0, \texttt{local} \rangle, \langle 1, \texttt{local} \rangle, \langle 2, \texttt{local} \rangle\}$

and the transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$. At the *initial state* s_0, all variables are set to 0 and mutexes are *unlocked*.

Based on the LTS, we have set of independence relation set I:

$I = \{(\langle 0, \texttt{lock}\ m \rangle, \langle 1, \texttt{lock}\ m' \rangle), (\langle 0, \texttt{lock}\ m \rangle, \langle 2, \texttt{lock}\ m' \rangle)),$
$(\langle 0, \texttt{unlock}\ m \rangle, \langle 2, \texttt{lock}\ m' \rangle)), (\langle 1, \texttt{lock}\ m \rangle, \cdots\}$

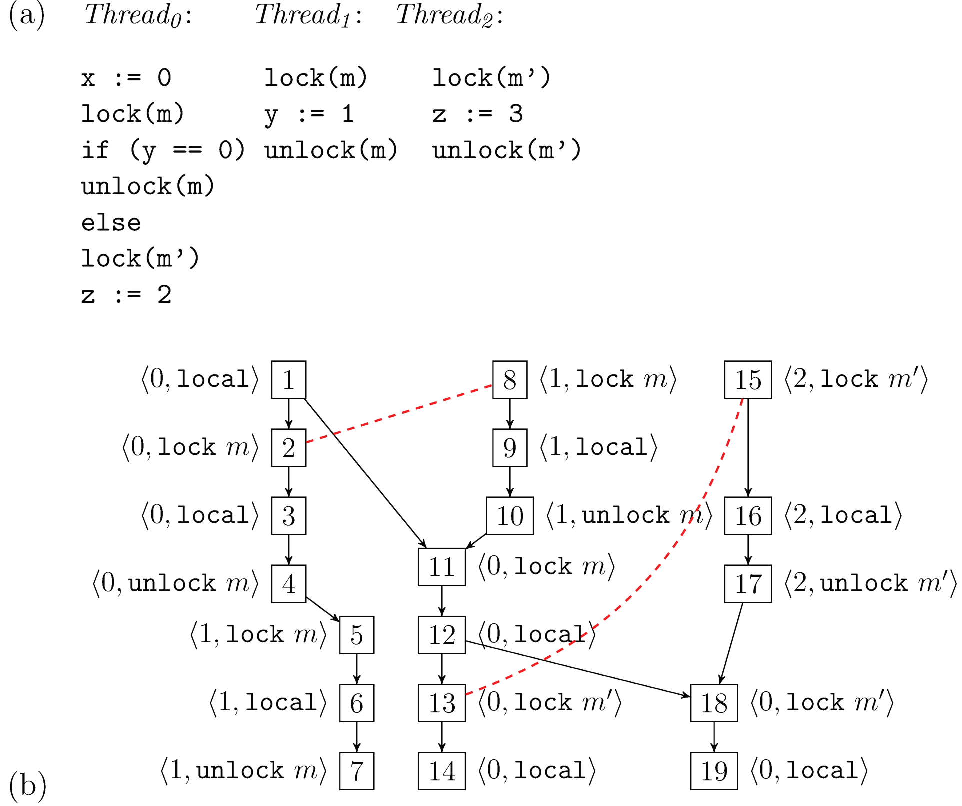


Figure 3.1: Unfolding example. (a): a program P; (b): its unfolding semantics $\mathcal{U}_{P,\diamondsuit_P}$.

where lock and unlock events touching different variables are independent. The dependence set $\mathcal{D} = \mathcal{A} \times \mathcal{A} \setminus I$.

The PES construction procedure shown in Definition 7 leads to the unfolding in Figure 3.1.(b). The unfolding events are named by numbers which are actually the visiting order and labeled with corresponding actions in $\mathcal{A}$. Conflicts and causalities are denoted by dashed red and solid black lines respectively. A Mazurkiewicz trace of execution is represented by a maximal configuration. There are a total of 4 maximal configurations in the unfolding of the program. The run $\langle 0, \texttt{local}\rangle, \langle 0, \texttt{lock}\ m\rangle, \langle 0, \texttt{local}\rangle$, $\langle 0, \texttt{unlock}\ m\rangle, \langle 1, \texttt{lock}\ m\rangle, \langle 1, \texttt{local}\rangle, \langle 1, \texttt{unlock}\ m\rangle$ yields a maximal configuration $\{1, 2, 3, 4, 5, 6, 7\}$. Similarly, the run $\langle 0, \texttt{local}\rangle, \langle 1, \texttt{lock}\ m\rangle, \langle 1, \texttt{local}\rangle, \langle 1, \texttt{unlock}\ m\rangle$, $\langle 0, \texttt{lock}\ m\rangle, \langle 0, \texttt{local}\rangle, \langle 0, \texttt{lock}\ m'\rangle, \langle 0, \texttt{local}\rangle$ yields $\{1, 8, 9, 10, 11, 12, 13, 14\}$. Let $\mathcal{P}$ be an unfolding prefix $\{\bot\}$. We can extend it with possible events $1, 8$ or 15 by Definition 8. Assume that we extend the unfolding prefix with 1 whose canonical name is $\langle\langle 0, \texttt{local}\rangle, \{\bot\}\rangle$, then produce the new prefix $\{\bot, 1\}$. To continue, let us consider the action $\langle 0, \texttt{lock}\ m\rangle$ and an extension event $2 = \langle\langle 0, \texttt{lock}\ m\rangle, \{\bot, 1\}\rangle$. Two $\bot$ and 1 satisfy $\bot < 2$ and $1 < 2$. Let us take $\langle 0, \texttt{lock}\ m\rangle$ to produce 2 with cononical

name $\langle\langle 0, \texttt{lock}\ m\rangle, \{\bot, 1\}\rangle$ in the prefix. Since $(\langle 0, \texttt{lock}\ m\rangle \diamondtimes \langle 1, \texttt{lock}\ m\rangle)$, $\mathcal{K}_{\mathcal{P},\diamond,2}$ results in event 8 with the canonical name $\langle\langle 1, \texttt{lock}\ m\rangle, \{\bot\}\rangle$ in the unfolding prefix. When continuing the process with other actions, we obtain the unfolding shown in Figure 3.1.(b).

3.2.2 Algorithm correctness

We now discuss the correctness of this algorithm. First we introduce some general lemmas as a basis for correctness theorems.

Lemma 1. *Let $\langle C, D, A, e\rangle$ is a node of the call tree. We have the following:*

- $D \cap A = \emptyset$ *(1)*
- $e \in \texttt{ena}(C)$ *(2)*
- *C is a configuration* *(3)*
- *$C \cup A$ is configuration and $C \cap A = \emptyset$* *(4)*

Proof.

(3.1): Base case: $D = A = \emptyset$, so clearly $D \cap A = \emptyset$

Step: Assume D. By Definition 9, $D \cap J = \emptyset$ and $A = J \setminus C$, then $A \subset J$, clearly $A \cap D = \emptyset$.

(3.2): Observe that e is picked from $\texttt{ena}(C)$ at Line 7 or $\texttt{ena}(C) \cap A$ at Line 9, it is definitely the case that $e \in \texttt{ena}(C)$.

(3.3) Base case: $n = 0$ and $C = \{\bot\}$. The set $\{\bot\}$ is a configuration. Assume C_{n-1} is a configuration and $b_{n-1} \triangleright b_n$ (either left or right subtree), then $C = C_{n-1} \cup \{e\}$. As stated in (3.2), $e \in en(C')$, so by definition of the enable set, $C' \cup \{e\}$ is a configuration, equivalently C is a configuration.

(3.4) Base case: $n = 0$, $C = \{\bot\}$, and $A = \emptyset$, so clearly $C \cup A$ is a configuration and $C \cap A = \emptyset$.

Step: Now assume $C_{n-1} \cup A_{n-1}$ is a configuration and $C_{n-1} \cap A_{n-1} = \emptyset$. There are two cases:

- $b_{n-1} \triangleright_l b_n$: If A_{n-1} is empty and then A is also empty. Certainly, $C \cup A$ is a configuration and $C \cap A = \emptyset$. If A_{n-1} is not empty, then $C = C_{n-1} \cup \{e\}$ where e is taken from $\texttt{ena}(C_{n-1}) \cap A_{n-1})$ and $A = A_{n-1} \setminus \{e\}$. We have $C \cup A = (C_{n-1} \cup \{e\}) \cup (A_{n-1} \setminus \{e\}) = C_{n-1} \cup A_{n-1}$. Since $C_{n-1} \cup A_{n-1}$ is a configuration, so is definitely $C \cup A$. Similarily, $C \cap A = C_{n-1} \cap A_{n-1} = \emptyset$.
- $b_{n-1} \triangleright_r b_n$: $C = C_{n-1}$ and $A = J \setminus C_{n-1}$ for some $J \in \texttt{Alt}(C_{n-1}, D \cup \{e\})$. We have:

  $C \cup A = C_{n-1} \cup J \setminus C_{n-1} = C_{n-1} \cup J$

  By definition of clue Definition 9, $C_{n-1} \cup J$ is a configuration, so $C \cup J$ is also configuration. On the other hand, by construction of A, it is clear that $C \cap A = \emptyset$.

□

Lemma 2. *If $C \subseteq C'$ are two finite configurations, then $en(C) \cap (C' \setminus C) = \varnothing$ iff $C' \setminus C = \varnothing$.*

Proof. If there is some $e \in en(C) \cap (C' \setminus C)$, then $e \in C'$ and $e \notin C$, so $C' \setminus C$ is not empty. If there is some $e' \in C' \setminus C$, then there is some event e'' that is $<$-minimal in $C' \setminus C$. As a result, $\lceil e'' \rceil \subseteq C$. Since $e'' \notin C$ and $C \cup \{e\}$ is a configuration (as $C \cup \{e\} \subseteq C$), we have that $e'' \in en(C)$, then $en(C) \cap (C' \setminus C)$ is not empty. □

Lemma 3. *Let $b = \langle C, D, A, e \rangle$ and $b' = \langle C', D', A', e' \rangle$ be two nodes in the call tree such that $b \triangleright b'$, then: $C \subseteq C'$ and $D \subseteq D'$ and*

- *if $b \vartriangle_l b'$ then $C \subsetneq C'$*
- *if $b \vartriangle_r b'$ then $D \subsetneq D'$*

Proof. If $b \vartriangle_l b'$ then `Explore` (C,D,A) is called at Line 10, then $C' = C \cup \{e\}$ and $D' = D$, so clearly $C \subsetneq C'$.

If If $b \vartriangle_r b'$ then `Explore` (C,D,A) is called at Line 12, then $C' = C$ and $D' = D \cup \{e\}$, so clearly $D \subsetneq D'$. □

Theorem 2 (Termination)**.** *Let U be the set of events, $C \subseteq U$ a configuration of $\mathcal{U}_{M,\diamond}$ and two sets of events $D, A \subseteq U$, a recursive call of the function* `Explore` *(C,D,A) always terminates in finite time.*

Proof. By contradiction. Assume that there is an infinite path (infinite recursive call of `Explore` (C,D,A)) $b_0 \rhd b_1 \rhd ..b_{n-1} \rhd b_n$ in the call tree. Let $b_i = \langle C_i, D_i, A_i, e_i \rangle$ with $i \geq 0$. Since $\mathcal{U}$ has finitely many events, all configurations in $\mathcal{U}$ are finite. It is observed that C_i and C_{i+1} are related finite times because each time `Explore` (C,D,A) is called at Line 7 and Line 9, only one event is added to C_i. Formally, $L := \{i \in \mathcal{N} : C_i \vartriangle_l C_{i+1}\}$ is a finite set. Assume $k = max(L) + 1$ to be the index of the path such that $\forall i \geq k : C_i \rhd_r C_{i+1}$. We then have $C_i = C_k$ with $\forall i \geq k$ and $D_i \subset ex(C_k)$. Recall that $ex(C_k)$ is finite and by Lemma 3, so we have: $D_k \subsetneq D_{k+1} \subsetneq D_{k+2} \subsetneq \ldots$

This is a contradiction as we can have D_{k+j} larger than $ex(C_k)$ with some large enough j. □

Lemma 4. *Let $b := \langle C, D, A, e \rangle \in B$ be a node in the call tree and $\hat{C}$ an arbitrary maximal configuration of $\mathcal{U}$ such that $C \subset \hat{C}$ and $D \cap \hat{C} = \emptyset$, then one of the following statements holds:*

- *C is a maximal configuration of $\mathcal{U}$ or*
- *$e \in \hat{C}$ and b has a left subtree or*
- *$e \notin \hat{C}$ and b has a right subtree.*

Proof. If C is maximal, the first statement holds, so we are done. So assume that C is not maximal, since $C \subset \hat{C}$, then b has at least one left child. If $e \in \hat{C}$, then we are done, as the second statement holds. Now assume that $e \notin \hat{C}$, we need to show that the third statement holds, i.e. that b has right child. In particular we show that there exists some clue $\hat{J} \subseteq \hat{C}$ such that $\hat{J} \in \texttt{Alt}(C, D \cup \{e\})$. By definition, if such $\hat{J}$ exists, then $\hat{J} \cap (D \cup \{e\}) = \emptyset$ and $\hat{J} \cup C$ is a configuration. We define now the set:

$F := \{e_1, .., e_n\} := (\hat{C} \setminus C) \cap (\texttt{ena}(C) \setminus \{e\}$

By Lemma 2, it is obvious that $F \neq \emptyset$ unless C is maximal. By definition, $F \subset \hat{C}$ and $\hat{C} \cap D = \emptyset$, so $F \cap D = \emptyset$. Plus, $e \notin \hat{C}$ and $F \subset \hat{C}$, so $e \notin F$. Thus, $F \cap (D \cup \{e\}) = \emptyset$. On the other hand, we observe that $\texttt{ena}(C) \setminus \{e\} \neq \emptyset$, because $C \subset \hat{C}$ and $e \notin \hat{C}$. Therefore, there is some $e' \in \texttt{ena}(C)$ such that $\{e'\} \cup C$ is a configuration (by definition of $\texttt{ena}(C)$ and $en(C)$). Therefore, it is possible to find a configuration $\hat{J} \subseteq F$ (at least $\hat{J} = \{e'\}$) such that $J \cup C$ is a configuration. Summarily, with $e \notin \hat{C}$, there exists a configuration J such that $J \in \texttt{Alt}(C, D \cup \{e\})$ which means the third statement holds. □

Lemma 5. *For any node $b = \langle C, D, A, e\rangle \in B$ in the call tree and any maximal configuration $\hat{C}$ of $\mathcal{U}$, if $C \subseteq \hat{C}$ and $D \cap \hat{C} = \emptyset$ and Lemma 4 holds for all nodes of subtrees rooted from b, there is a node $b' = \langle C', D', A', .\rangle \in B$ such that $b \vartriangle^* b'$ and $C' = \hat{C}$.*

Proof. Assume that Lemma 4 holds for all nodes b'' in subtrees of b. Since $C \subset \hat{C}$ and $D \cap \hat{C} = \emptyset$, we can apply Lemma 4 to b and $\hat{C}$. If C is maximal, then clearly $C = \hat{C}$. If not, by Lemma 4, if $e \in \hat{C}$ then b has a left child b_1, otherwise, $e \notin \hat{C}$, b has a right child b_1 with $b_1 = \langle C_1, D_1, A_1, e_1\rangle$. In any case, we observe that $C_1 \subseteq \hat{C}$ and $D_1 \cap \hat{C} = \emptyset$.

If C_1 is a maximal configuration, then necessarily $C_1 = \hat{C}$, so $b' := b_1$ is what we need. If not, continue to apply Lemma 4 to b_1 or maybe its children until we reach a node $b_n = \langle C_n, ., ., .\rangle$ where $C_n = \hat{C}$ is a maximal configuration. Because all paths in the call tree are finite, so this repeat process terminates. We can take $b' := b_n$. ☐

Theorem 3 (Soundness)**.** *Let $\hat{C}$ be a $\subseteq$-maximal configuration of $\mathcal{U}_{M,\diamond}$. Algorithm 1 calls the function* `Explore(`C, D, A`)` *at least once with $C = \hat{C}$.*

Proof. It is necessary to prove that for each maximal configuration $\hat{C}$, we can find a node $b = \langle \hat{C}, ., ., .\rangle$. This is an immediate result of Lemma 5. Take the root node $b_0 = \langle C, D, A, \bot\rangle$ with $C = \{\bot\}$ and $D = A = \varnothing$. Clearly $C \subset \hat{C}$ and $D \cap \hat{C} = \emptyset$ and Lemma 4 holds for all nodes of the call tree. Therefore, apply Lemma 5 to $\hat{C}$ and b_0, which insures such a node b exists. ☐

3.3 Partial alternatives

In the algorithm described in Algorithm 1, an alternative is computed by $Alt(C, D)$ which is intuitively defined but not instantiated. In our research, we aim to find an efficient algorithm to implement it. We prove first that the problem of computing an alternative is NP-complete.

3.3.1 Complexity of computing alternatives

We prove this complexity bound for a general model of computation (Petri nets), and show that it remains NP-hard for a more restrictive model of computation (multi-threaded programs with mutexes).

Theorem 4. *Let $\mathcal{E} := \langle E, <, \#, h\rangle$ be a finite PES, $C \subseteq E$ a configuration, and $D \subseteq ex(C)$ a set of events. Deciding whether there is an alternative to D after C in E is NP-complete.*

Proof. We first prove that the problem is in NP. Let us non-deterministically choose a configuration $J \subseteq E$. We then check that J is an alternative to D after C:

- $J \cup C$ is a configuration can be checked in linear time: The first condition for $J \cup C$ to be a configuration is that $\forall e \in J \cup C : \lceil e \rceil \subseteq J \cup C$. Since J is a configuration, this condition holds for all $e \in J$. Similarly, as C is a configuration, it also holds for all $e \in C$. The second condition is that $\forall e_1, e_2 \in J \cup C : \neg(e_1 \# e_2)$. This is true for $e_1, e_2 \in J$ and $e_1, e_2 \in C$. If $e_1 \in J \wedge e_2 \in C$ (or the converse), we have to effectively check that $\neg(e_1 \# e_2)$. Checking if two events e_1 and e_2 are in conflict is linear on the size of $[e_1] \cup [e_2]$.

- Every event $e_1 \in D$ must be in immediate conflict with an event $e_2 \in J$. Thus, there are at most $|D| \cdot |J|$ checks to perform, each in linear time on the size of $[e_1] \cup [e_2]$. Hence, this is in $O(n^2)$.

We now prove that the problem is NP-hard, by reduction from the 3-SAT problem. Let $\{v_1, \ldots, v_n\}$ be a set of Boolean variables. Let $\phi := c_1 \wedge \ldots \wedge c_m$ be a 3-SAT formula, where each clause $c_i := l_i \vee l'_i \vee l''_i$ comprises three literals. A literal is either a Boolean variable v_i or its negation $\overline{v_i}$.

Formula ϕ can be modelled by a PES $\mathcal{E}_\phi := \langle E, <, \#, h \rangle$ constructed as follows:

- For each variable v_i we create two events t_i and f_i in E, and put them in immediate conflict, as they correspond to the satisfaction of v_i and $\overline{v_i}$, respectively.

- The set D of events to disable contains one event d_j per clause c_j. Such a d_j has to be in immediate conflict with the events modelling the literals in clause c_j. Hence it is in conflict with 1, 2, or 3 t or f events.

- There is no causality: $< := \emptyset$.

- The labelling function shows the correspondence between the events and the elements of formula ϕ, i.e. $\forall t_i \in E : h(t_i) = v_i$, $\forall f_i \in E : h(f_i) = \overline{v_i}$ and $\forall d_j \in E : h(d_j) = c_j$.

We now show that ϕ is satisfiable iff there exists an alternative J to D after $C := \emptyset$ in E. This alternative is constructed by selecting for each event $d_j \in D$ and event e in immediate conflict. By construction of $\mathcal{E}_\phi$, $h(e)$ is a literal in clause $h(d_j) = c_j$. Moreover, $C \cup J = J$ must be a configuration. The causal closure is trivially satisfied since $< := \emptyset$. The conflict-freeness implies that if $t_i \in J$ then $f_i \notin J$ and vice-versa. Therefore, formula ϕ is satisfiable iff an alternative J to D exists.

The construction of $\mathcal{E}_\phi$ is illustrated in Figure 3.2 for:

$$\phi := \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{c_1} \wedge \underbrace{(\overline{x_1} \vee \overline{x_2})}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_3})}_{c_3}$$

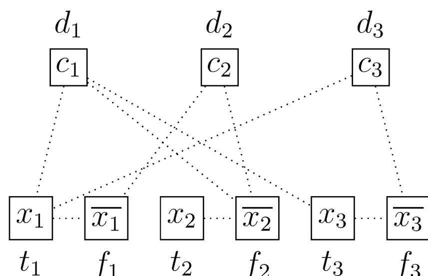


Figure 3.2: Example of encoding a 3-SAT formula.

□

Theorem 4 assumes that the input PES is an arbitrary PES, rather than the unfolding semantics of a program. One may ask if the cost of computing alternatives is reduced by assuming that the PES is the unfoling of a program. The answer is negative:

Theorem 5. *Let P be a program, $\diamondsuit$ an independence relation on M_P, and U a causally-closed set of events from $\mathcal{U}_{M_P,\diamondsuit}$. Let $C \subseteq U$ be a configuration and $D \subseteq ex(C)$ a set of events. Deciding whether there is an alternative to D after C in U is NP-complete.*

Proof. Observe that the only difference between the statement of this theorem and that of Theorem 4 is that here we assume the PES to be the unfolding of a given program P.

As a result, the problem is obviously in NP, as restricting the class of PESs that we have as input cannot make the problem more complex.

However, showing that the problem is NP-hard requires a new encoding, as the (simple) encoding given for Theorem 4 generates PESs that may not be the unfolding of any program. Recall that two events in the unfolding of a program are in immediate conflict only if they are lock statements on the same variable. So, in Figure 3.2, for instance, since $t_1 \# f_1$ and $f_1 \# d_2$, then necessarily we should have $t_1 \# d_2$, as all the three events should be locks to the same variable.

For this reason we give a new encoding of the 3-SAT problem into our problem. As before, let $V = \{v_1, \ldots, v_n\}$ be a set of Boolean variables. Let $\phi := c_1 \wedge \ldots \wedge c_m$

be a 3-SAT formula, where each clause $c_i := l_i \vee l'_i \vee l''_i$ comprises three literals. A literal is either a Boolean variable v_i or its negation $\overline{v_i}$. As before, for a variable v, let $pos(v)$ denote the set of clauses where v appears positively and $neg(v)$ the set of clauses where it appears negated. We assume that every variable either appears positively or negatively in a clause (or does not appear at all), as clauses where a variable happens both positively and negatively can be removed from ϕ. As a result $pos(v) \cap neg(v) = \emptyset$ for every variable v.

Let us define a program P_ϕ as follows:

- For each Boolean variable v_i we have two threads in P, t_i corresponding to v_i (true), and f_i corresponding to $\overline{v_i}$ (false). We also have one lock l_{v_i}.
- Immediately after starting, both threads t_i and f_i lock on l_{v_i}. This scheme corresponds to choosing a Boolean value for variable v_i: the thread that locks first chooses the value of v_i.
- For each clause $c_j \in \phi$, we have a thread d_j and a lock l_{c_j}. The thread contains only one statement which is locking l_{c_j}.
- For each clause $c_j \in pos(v_i) \cup neg(v_i)$, the program contains one thread $r_{\langle v_i, c_j \rangle}$ (run for variable v_i in clause c_j). This thread contains only one statement which is locking l_{c_j}.
- After locking on l_{v_i}, thread t_i starts in a loop all threads $r_{\langle v_i, c_j \rangle}$, for $c_j \in pos(v_i)$.
- Similarly, after locking on l_{v_i}, thread f_i starts in a loop all threads $r_{\langle v_i, c_j \rangle}$, for $c_j \in neg(v_i)$.

When P_ϕ is unfolded, each statement of the program gives rise to exactly one event in the unfolding. Indeed, by construction, each t_i or f_i thread starts by a lock event and then causally leads to one r event per clause the variable v_i appears in. Any two of them concern different clauses and thus different locks, and they are independent.

Let $C := \emptyset$ be an empty configuration, $D := \{d_1, \ldots, d_m\}$, and U the set of all events in the unfolding of the program.

We now show that ϕ is satisfiable iff there exists an alternative J to D after $C := \emptyset$ in $\mathcal{U}_{P_\phi}$. This alternative is constructed by selecting for each event $d_j \in D$ and event e in immediate conflict. By construction of P_ϕ, it is a $r_{\langle v_i, c_j \rangle}$ where v_i is a literal in clause $h(d_j) = c_j$. Moreover, $C \cup J = J$ must be a configuration. In order to satisfy the causal closure, since $< := \{\langle t_i, r_{\langle v_i, c_j \rangle} \rangle : c_j \in pos(v_i)\} \cup \{\langle f_i, r_{\langle v_i, c_j \rangle} \rangle : c_j \in neg(v_i)\}$,

J must also contain the t_i or f_i preceding $r_{\langle v_i, c_j \rangle}$. The conflict-freeness implies that if $t_i \in J$ then $f_i \notin J$ and vice-versa. Therefore, formula ϕ is satisfiable iff an alternative J to D exists.

There are at most $2|V| + |\phi|(|V| + 1)$ events, so the construction can be achieved in polynomial time.

Therefore our problem is NP-hard.

The construction of $\mathcal{U}_{P_\phi}$ is illustrated in Figure 3.3 for:

$$\phi := \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{c_1} \wedge \underbrace{(\overline{x_1} \vee \overline{x_2})}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_3})}_{c_3}$$

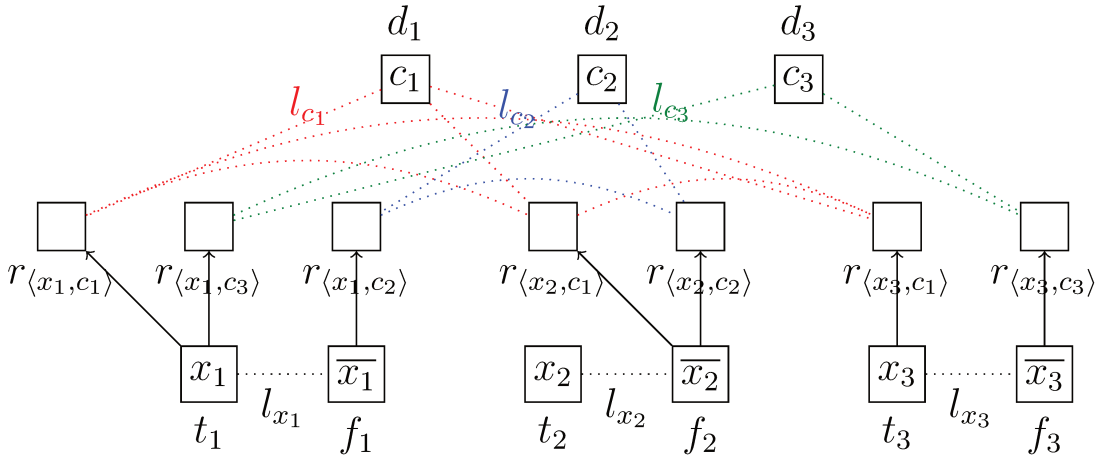


Figure 3.3: Program unfolding encoding a 3-SAT formula.

□

These complexity results lead us to consider new approaches that avoid the combinatorial explosion of a NP-complete problem.

3.3.2 Motivating example

In the previous section, we have proved that computing alternatives in an optimal DPOR is an NP-complete problem. However, on the other hand, the non-optimal, state-of-the-art SDPOR algorithm [2] suffers from exploring exponential number of redundant executions. The program shown in Figure 3.4 (a) illustrates a practical consequence of this result: SDPOR can explore here $\mathcal{O}(2^n)$ interleavings but the program has only $\mathcal{O}(n)$ Mazurkiewicz traces.

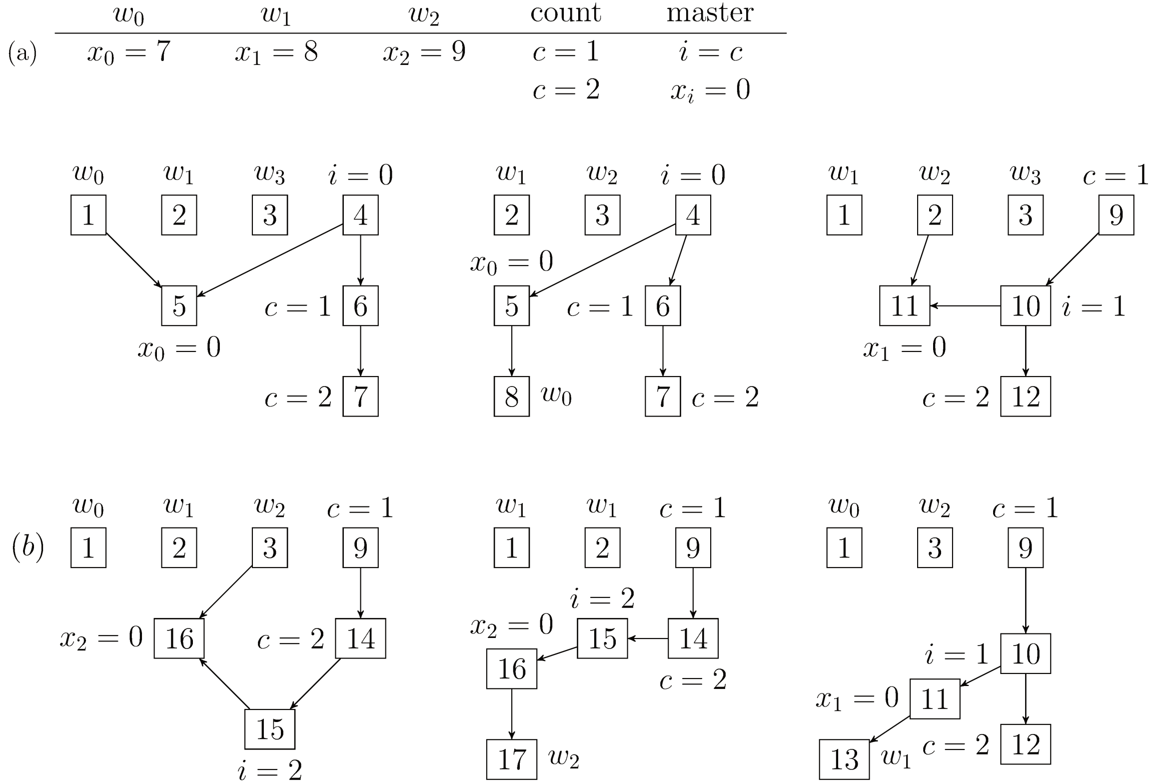


Figure 3.4: Motivating example. (a): Programs; (b): Partially-ordered executions;

The program contains $n := 3$ *writer* threads w_0, w_1, w_2, each writing to a different variable. The thread *count* increments $n - 1$ times a zero-initialized counter c. Thread *master* reads c into variable i and writes to x_i.

The statements $x_0 = 7$ and $x_1 = 8$ are independent because they produce the same state regardless of their execution order. Statements $i = c$ and any statement in the *count* thread are dependent or *interfering*: their execution orders result in different states. Similarly, $x_i = 0$ interferes with exactly one *writer* thread, depending on the value of i.

Using this independence relation, the set of executions of this program can be partitioned into six Mazurkiewicz traces, corresponding to the six partial orders shown in Figure 3.4 (b). Thus, an optimal DPOR explores six executions ($2n$-executions for n *writers*). We now show why SDPOR explores $\mathcal{O}(2^n)$ in the general case. Conceptually, SDPOR is a loop that (1) runs the program, (2) identifies two dependent statements that can be swapped, and (3) reverses them and re-executes the program. It terminates when no more dependent statements can be swapped.

Consider the interference on the counter variable c between the *master* and the *count*

thread. Their execution order determines which *writer* thread interferes with the *master* statement $x_i = 0$. If $c = 1$ is executed just before $i = c$, then $x_i = 0$ interferes with w_1. However, if $i = c$ is executed before, then $x_i = 0$ interferes with w_0. Since SDPOR does not track relations between dependent statements, it will naively try to reverse the race between $x_i = 0$ and *all writer threads*, which results in exploring $\mathcal{O}(2^n)$ executions. In this program, exploring only six traces requires understanding the entanglement between both interferences as the order in which the first is reversed determines the second.

In the light of these findings, it becomes clear that existing unoptimal PORs avoid the NP-hard combinatorial explosion when computing alternatives using an inexpensive procedure whose actual price is to explore up to exponentially redundant executions. On the other hand, optimal PORs are necessarily subject to the potential combinatorial explosion arising from having to solve a NP-hard problem on every node of the tree. These two observations motivate the need for specialised algorithms and data structures for computing alternatives. What we ideally need here is a solution in the middle: a polynomial algorithm for computing unoptimal alternatives (i.e., the resulting POR will be unoptimal) which can approximate the result of an optimal algorithm.

3.3.3 k-partial alternatives

In this section, we propose a polynomial time algorithm $\texttt{Alt}_k$ to compute *clues* considering a subset of D of size k. A *k-partial alternative* may not be an alternative but a clue that might lead us to an actual alternative. Thus, sometimes we explore a fault alternative but even in that case, we can backtrack and select another clue to continue. We here trade some redundant explorations for complexity reduction. It is clear that there is a certain threshold k_0 in which $\texttt{Alt}_{\geq k_0}$ produces alternatives and when k_0 holds, we are finding the alternative for the full size of D which can be called *optimal alternative.*

The main contribution here is the formalisation of the polynomial time algorithm of clues. This algorithm and the computation of extensions of a configuration fundamentally rely on efficient procedures to check causality and conflict. Thus, a second contribution of this section are specialised algorithms for checking general causality and conflict between events.

The algorithm $\texttt{Alt}_k$ computes $k-$partial alternatives.

Definition 12 (k-partial alternative)**.** *Let U be a set of events, $C \subseteq U$ a configuration, $D \subseteq U$ a set of events, and $k \in \mathbb{N}$ a non-negative integer. A configuration J is a k-*partial alternative *to D after C if there is some $\hat{D} \subseteq D$ such that $|\hat{D}| = k$ and J is a alternative to $\hat{D}$ after C.*

A k-partial alternative is a straightforward restriction of alternatives that considers a subset of D of size k. We compute k-partial alternatives using a data structure that we call *comb*. Intuitively, the *comb* is the space of combinations for candidate k-partial alternatives.

Definition 13 (Comb)**.** *Let A be a set. An* A-comb *c of size $n \in \mathbb{N}$ is an ordered collection of* spikes *$\langle s_1, \ldots, s_n \rangle$, where $s_i \in A^*$ is a sequence of elements over A. Furthermore, a* combination *over c is any tuple $\langle a_1, \ldots, a_n \rangle$ where $a_i \in s_i$ is an element of the spike.*

The problem of finding a k-partial alternative J in the resulting comb amounts to find a combination of events that is *conflict free* and *causally closed*. Hence, it is possible to compute k-partial alternatives (and by extension optimal alternatives) to D after C in U using a comb, as follows:

1. Select k (or $|D|$, whatever is smaller) arbitrary events $e_1, \ldots, e_k$ from D.
2. Build a U-comb $\langle s_1, \ldots, s_k \rangle$ of size k, where spike s_i contains all events in U in (immediate) conflict with e_i.
3. Remove from s_i any event $\hat{e}$ such that either $[\hat{e}] \cup C$ is not a configuration or $[\hat{e}] \cap D \neq \emptyset$.
4. Find some combination $\langle e'_1, \ldots, e'_k \rangle$ in the comb satisfying that $\neg(e'_i \# e'_j)$ for $i \neq j$.
5. For any such combination the set $J := [e'_1] \cup \ldots \cup [e'_k]$ is a k-partial alternative.

Step 3 guarantees that the set J is a clue. Steps 1 and 2 guarantee that it will conflict with at least k events from D, ensuring that it will be a k-partial alternative.

Steps 2, 3, and 4 require to decide whether a given pair of events is in conflict. Computing k-partial alternatives thus reduces to computing conflicts between events. The key problem remaining in computing a *k-partial alternative* is then how to efficiently check conflict and causality among events.

3.4 Conflicting extensions

Another important function in Algorithm 1 is $ex(C)$ at Line 3, which computes all events that are enabled at some subset of C. $ex(C)$ is clearly partitioned into $en(C)$ and $cex(C)$. While $en(C)$ is trivially computed, $cex(C)$ is unfortunately much more complicated. We recall that *conflicting extensions* is a set of events enabled at some sub-configuration of C but conflicts with at least one event in C. We discuss the algorithm of computing conflicting extensions of a configuration and its complexity.

3.4.1 Conflicting extension algorithm

Conflicting extension algorithm $cex(C)$ returns a set of events that are enable at C but conflict with one of events in C. To compute conflicting extension $cex(C)$ for a configuration C, we compute all conflicting events resulted by $cex(e)$ for each event e in C. We exploit that fact that in concurrent programs, only mutex locks touch shared variables, so we necessarily compute conflicting events for *lock* events with the algorithm described in Algorithm 2 and ignore the others:

Algorithm 2: Compute conflicting events for a mutex lock

```
1  Given an event e with h(e) = t, a mutex transition
2  Set ep := pp(e), em := pm(e).
3  Procedure cex(e)
4      while ¬(em ≤ ep) do
5          Set em = pm(pm(em))
6          Create (or retrieve) an event ex such that:
7              h(ex) := t
8              pp(ex) := ep
9              pm(ex) := em
10         Add ex to cex(C)
11     end
12     return
```

In Algorithm 2, $pp(e)$ returns the maximal event for the thread of e in e's local configuration: $\forall e \in E : pp(e) = e' \in E$ such as $p(e) = p(e')$ and $e' < e$ and $\nexists e'' : p(e'') = p(e')$ and $e' < e''$ where $p(e)$ returns the thread of e. $pm(e)$ returns the maximal event related to mutex in e's local configuration: $\forall e \in E : pm(e) = e' \in E$ such as $v(e) = v(e')$ and $e' < e$ and $\nexists e'' : e' < e''$ and $v(e'') = v(e')$ where $v(e)$ returns the mutex which is modified by e.

New conflict events ex are generated by combining the labelling transition t (Line 7), thread maximal predecessor ep (Line 8) and one of the lock events found in

the trace of the variable predecessor from the maximal one to the root identified by $pm(pm(em))$ in Line 5. The procedure possibly terminates before reaching the root when we reach the predecessor that is also predecessor of ep (Line 4).

This algorithm guarantees that we find out all possible conflicting events for all lock events in the configuration. It is also sufficient to apply the algorithm to maximal configurations C, since it assures that the resulting conflicting extension covers all subsets for all sub-configuration of C.

3.4.2 Complexity

The computation of conflicts is NP-complete for an arbitrary independence relation in a Petri net.

Theorem 6. *Let N be a Petri net, t a transition of N, $\diamondsuit$ an independence relation on N, and $C \in conf(\mathcal{U}_{M_N,\diamondsuit})$ a configuration. Deciding whether $h^{-1}(t) \cap cex(C) = \emptyset$ is NP-complete.*

Proof. We first prove that the problem is in NP. This is achieved using a *guess and check* non-deterministic algorithm to decide the problem. Let us non-deterministically choose a configuration $C' \subseteq C$, in linear time on the input. A linearisation of C' is chosen and used to compute the marking m reached. We check that m enables t and that for any $<$-maximal event e of C, $h(e) \not\diamondsuit\ t$ holds. Both tests can be done in polynomial time. If both tests succeed then we answer *yes*, otherwise we answer *no*.

We now prove that the problem is NP-hard, by reduction from the 3-SAT problem. Let $V = \{v_1, \ldots, v_n\}$ be a set of Boolean variables. Let $\phi := c_1 \wedge \ldots \wedge c_m$ be a 3-SAT formula, where each clause $c_i := l_i \vee l'_i \vee l''_i$ comprises three literals. A literal is either a Boolean variable v_i or its negation $\overline{v_i}$. For a variable v, $pos(v)$ denotes the set of clauses where v appears positively and $neg(v)$ the set of clauses where it appears negated.

Given ϕ, we construct a 3-safe net N_ϕ, an independence relation $\diamondsuit$, a configuration $C \in conf(\mathcal{U}_{M_{N_\phi},\diamondsuit})$, and a transition t from N_ϕ such that ϕ is satisfiable iff some event in $ex(C)$ is labelled by t :

- The net contains one place d_i per clause c_i, initially empty.
- For each variable v_i are two places s_i and s'_i. Places s_i initially contain 1 token while places s'_i are empty.

- For each variable v_i, a transition p_i takes into account positive values of the variable. It takes a token from s_i, puts one in s'_i (to move on to the other possibility for this variable) and puts one token in all places associated with clauses $c_j \in pos(v_i)$. This transition mimics the validation of clauses where the variable appears as positive.

- For each variable v_i, a transition n_i takes into account negative values of the variable. It takes a token from s'_i and puts one token in all places associated with clauses $c_j \in neg(v_i)$. It also removes one token from all places associated with clauses $c_j \in pos(v_i)$, that have been marked by some p_k transition. This transition n_i mimics the validation of clauses where the variable appears as negative.

- Finally, a transition t is added that takes a token from all d_i. Thus, it can only be fired when all clauses are satisfied, i.e. formula ϕ is satisfied.

The independence relation $\diamondsuit$ is the smallest binary, symmetric, irreflexive relation such that $p_i \diamondsuit p_j$ exactly when $i \neq j$ and $p_i \diamondsuit n_j$ exactly when $i \neq j$. Recall that p_i, n_i correspond to respectively to the positive and negative valuations of variable v_i. In other words, $\otimes$ is the reflexive closure of the set

$$\{\langle p_i, n_i \rangle : 1 \leq i \leq n\} \cup \{\langle t, p_i \rangle : 1 \leq i \leq n\} \cup \{\langle t, n_i \rangle : 1 \leq i \leq n\}$$

Relation $\diamondsuit$ is an independence relation because:

- $\forall i \neq j$, transitions p_i and p_j do not share any input place ;

- $\forall i \neq j$, the intersection between $p_i^\bullet$ and ${}^\bullet n_j$ might not be empty, but n_j is always preceded by (and thus enabled after) p_j (and not p_i). So firing p_i cannot enable, nor disable, p_j, and firing p_i and n_j in any order reaches the same state.

Finally, configuration C contains exactly one event per p_i and one per n_i, hence $2|V|$ events. This is because transition n_i is dependent *only* with p_i, and independent (thus concurrent) to any other transition in C. Thus formula ϕ has a model iff there is an event $e \in en(C)$ labelled by t. Indeed, initially only positive transitions p_i are enabled that assign a positive value to their corresponding variable v_i. They add a token in all places d_j such that $c_j \in pos(v_i)$. Then, when a negative transition n_i fires, it deletes the tokens from these d_j that had been created by p_i since the variable cannot allow for validating these clauses anymore. It also adds tokens in the d_k such that $c_k \in neg(v_i)$ since the clauses involving $\overline{v_i}$ now hold. Therefore, the number of

tokens in a place d_j is the number of variables (or their negation) that validate the associated clause. Formula ϕ is satisfied when all clauses hold at the same time, i.e. each clause is validated by at least one variable. Thus all places d must contain at least one token (and enable t) for ϕ satisfaction.

The construction of N_ϕ is illustrated in Figure 3.5 for:

$$\phi := \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{c_1} \wedge \underbrace{(\overline{x_1} \vee \overline{x_2})}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_3})}_{c_3}$$

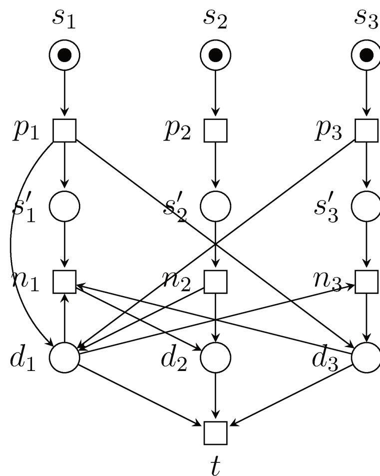


Figure 3.5: Petri Net encoding a 3-SAT formula.

□

This might be alleviated in the case of specific independence relations. Fortunately, for multithreaded programs, $cex(C)$ can be computed linearly. We observe that programs are deterministic i.e. at a time, in a thread, only one statement can be executed, and mutexes prevent from outside all conflicts with their inside statements. Therefore, only those in PES touching mutex variables might have conflict events. Consequently, in computing $cex(C)$, instead of finding possible conflict events for all events in C, we simply compute conflicting events for *lock* ones. Unlock events are also excluded since they must always follow the lock of the same variable in the same thread, hence do not have any conflicting events. The $cex(C)$ computation is linear to the number of lock events in C and now depends on the complexity of $Cex(e)$ algorithm (Algorithm 2) to find all possible conflicting events for an lock event, which is also $\mathcal{O}(n)$ where n is the number of lock events on the same mutex as e in C. Its efficiency now mainly depends on the causality deciding algorithm between ep and em in Line 4 which will be discussed in Section 3.5.

3.5 Conflict and Causality

Causality and conflict deciding queries are intensively requested in algorithms to compute alternatives (Algorithm 1) and conflicting extensions (Algorithm 2), so it is critical to solve these efficiently. We now propose a new data structure and algorithms to efficiently answer these queries when the underlying PES is the semantics of a program with locks.

Given two events in a PES, they are definitely in one of the relations: concurrent, causality or conflict. In Figure 3.1 (c), we can see $7 \# 3$, $5 \parallel 3$ and $7 < 10$. However, consider events in a single thread, they can not be concurrent, but either in causality or conflict. Take $Thread_0$ as an example, its four events $\{1, 5, 8, 9\}$ are all in conflict pairwise (this is rather a special example where there is only one statement per thread, so we cannot see the causality). It is similarly obvious for events that lock/unlock the same mutex variable.

In the light of these findings, representing events in a thread or touching a single mutex variable in a *sequential tree* explicitly preserves the structural relation. Events in the same *branch* are in causality and otherwise, those in different branches are in conflict. We store in memory a sequential tree per thread/variable. Consequently, an event in the unfolding belongs to one or two sequential trees corresponding to its thread and/or mutex. Using tree structures to represent causality and conflict follows the perspective that the unfolding is regarded as a synchronization of trees. We will discuss this structure in detail in Section 3.6.

Come back now to the causality and conflict relation between events in PES. Given two arbitrary events in the unfolding, they fall in one of the following cases:

- They are in the same thread if they modify the same mutex. (5)
- They modify the same mutex if they are in the same thread. (6)
- They neither are in the same thread nor modify the same mutex. (7)

An event in the unfolding belongs to one or two sequential trees in respect of its thread and mutex, so we store for each event one or two parents depending on its type where the second is only relevant for variable related ones.

As mentioned above, events in the same thread are definitely in the same sequential tree and so are those touching the same mutex. Assuming that an efficient algorithm is available for answering causality/conflict queries in a sequential tree, the cases (3.5) and (3.6) are simple to solve. In case (3.7), events are not in the same thread or

related to the same mutex, they necessarily do not belong to the same tree. We now need to store more data in order to apply the sequential tree algorithm. Observe that every event depends primarily on its local configuration, for example, if there exists any two conflicting events in local configurations of e and e', e surely conflicts with e'. Therefore, we annotate every event e with two sets of events, denoted by $pcut(e)$ and $vcut(e)$.

The set $pcut(e)$ stores, for every thread of the program, the only $<$-maximal event in $[e]$ of that thread. Formally, $pcut(e) := \{e_1, .., e_n\}$ where $\forall i \in [1, n] : e_i \in [e]$ and $p(e_i) = p_i$ and $\nexists e' \in [e] : e_i < e'$ and $p(e') = p_i$ with $p : U \to P$ indicating the process identifier the event belongs to. Let $vcut(e)$ denote the map from mutex to the maximal event associated with that mutex in the local configuration of e. More formally, $vcut(e) := \{e_1, .., e_k\}$ where $k \leq m$ and $\forall i \in [1, k] : e_i \in [e]$ and $v(e_i) = v_i$ and $\nexists e' \in [e] : e_i < e'$ and $v(e') = v_i$ with $v : U \to V \cup \{null\}$ mapping the event to the variable it touches or $null$ otherwise.

An event in PES belongs to a thread and/or locks a mutex and in no other case, so these two sets assure full coverage of events in one's local configuration. With these two sets, we have the following propositions:

Proposition 1 (Causality)**.** *Let e, e' be two different events, then $e < e'$ iff there is some $e'' \in pcut(e')$ such that $p(e'') = p(e)$ and ($e = e''$ or $e < e''$).*

Proof. We prove the proposition by proving the necessary and sufficient conditions as follows:

($\Rightarrow$) Assume that e, e' are two different events, $e < e'$ then there is some $e'' \in pcut(e')$ such that $p(e'') = p(e)$ and $e = e''$ or $e < e''$.

Let $e < e'$, it means that $e \in \lceil e' \rceil$ and there must exist e" $\in pcut(e')$ such that $pe(e'') = pe(e)$. Since e'' is the maximal event of process $p(e)$ in $[e']$, it is definite that $e'' = e$ or $e < e''$.

($\Leftarrow$) Assume there are two events e and e' and there exists $e'' \in pcut(e')$ such that $pe(e'') = pe(e)$ and ($e = e''$) or ($e < e''$), then $e < e'$

Since $e'' \in pcut(e')$, it means $e'' \in [e']$, equivalently $e'' < e'$. As in the assumption we already have: $e < e''$ or $e = e''$, so $e < e'' < e'$ that transitively implies that $e < e'$.

$\square$

Using Proposition 1, given two events, deciding whether $e < e'$ reduces to finding certain event e'' in $pcut(e')$ in the same thread of e and checking if $e < e''$, which is a much simpler task because e and e'' belong to the same thread.

We can prove a similar results for checking conflicts:

Proposition 2 (Conflict)**.** *Let e, e' be two different events. Then $e \# e'$ iff there is some $e_1 \in vcut(e)$ and some $e_2 \in vcut(e')$ such that $v(e_1) = v(e_2)$ and $e_1 \# e_2$.*

Proof. We prove the proposition by proving the necessity and sufficiency as follows:

($\Rightarrow$) Having $e \# e'$, there exists $e'_1 \in [e']$ and $e'_2 \in [e]$ such that $e'_1 \#^i e'_2$. Because of the fact that only mutex events touching the same variable are able to be in immediate conflict, we obviously know that $v(e'_1) = v(e'_2)$. Assume $e_1 \in vcut(e)$ such that $v(e_1) = v(e'_1)$ and $e_2 \in vcut(e') : v(e_2) = v(e'_2)$, then $v(e_1) = v(e'_1) = v(e'_2) = v(e_2)$.

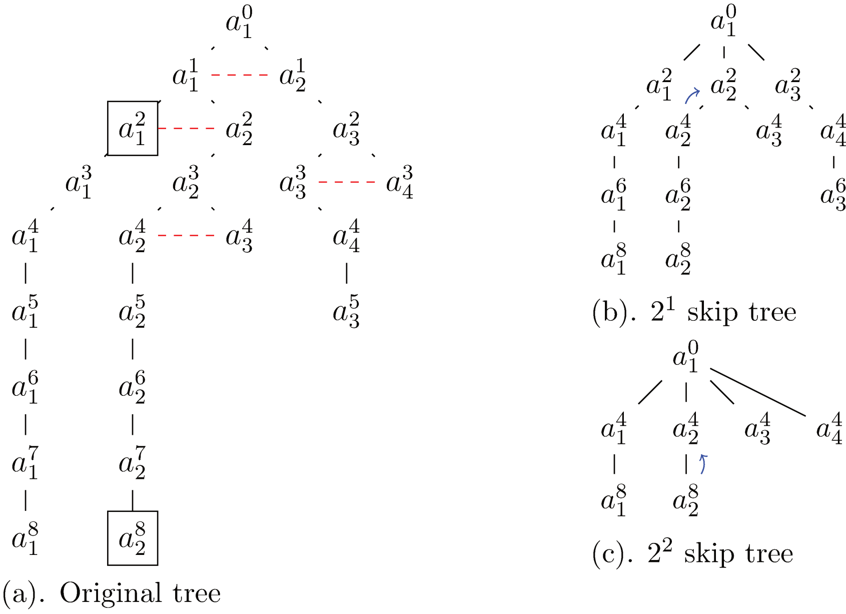
Moreover, $e'_1 < e_1$ or $e_1 = e'_1$ and similarly, $e_2 < e'_2$ or $e_2 = e'_2$ while the conflict relation is inherited, so $e_1 \# e_2$. Summarily, we have $v(e_1) = v(e_2)$ and $e_1 \# e_2$

($\Leftarrow$) Assume $e_1 \in vcut(e)$ and $\exists e_2 \in vcut(e')$ such that $ve(e_1) = ve(e_2)$ and $e_1 \# e_2$. Since $e_1 \in vcut(e)$, then it is obvious that $e_1 \in [e]$, implying that $e_1 < e$. Similarly, $e_2 \in [e']$ i.e. $e_2 < e'$. The conflict is inherited and $e_1 \# e_2$, so it is definite $e \# e'$.

$\square$

Thanks to Proposition 2, deciding whether $e \# e'$ holds reduces to finding events in $vcut(e)$ and $vcut(e')$ that lock or unlock on the same variable and are in conflict. Events e and e' are in conflict iff such events can be found. As before, this reduces the checking two events that modify the same mutex variable, which is much simpler than for arbitrary events.

Using the sets *pcut*, *vcut* and the two propositions above, deciding conflict and causality reduces to checking structural relations of two *nodes* in a *sequential tree*. Hence, we denote event's possible appearances in two sequential tree by two nodes $n_0 := \langle d_0, p_0, S_0 \rangle$ and $n_1 := \langle d_1, p_1, S_1 \rangle$ representing those in thread and mutex tree respectively. Sequential tree and structural relations between nodes are discussed in detail in Section 3.6.



(a). Original tree (b). 2^1 skip tree (c). 2^2 skip tree

Figure 3.6: Multiple trees for a process. a_j^i is node indexed j at the depth i

3.6 Sequential tree

We know that an event in the unfolding always belongs to at least one tree and at most two trees. The causality and conflict between events are decided based on their positions in these trees. We exploit here a tree structure to present events that are process or variable related.

3.6.1 Causality and Conflict of nodes

We let a partial order set $T := (N, prt)$ denote the sequential tree where N is a set of nodes, $prt \subseteq N \times N$ its *parent* relation where $a = prt(b)$ if there is an edge from a to b.

A *node* $n \in N$ is a tuple $n := < d, p, S >$ where d is the distance from the root, $p = prt(n)$ and $S := \{\forall s \in N : n = prt(s)\}$ indicates the set of right-after coming nodes, called *successors*.

Definition 14 (Ancestor relation)**.** *Given two nodes n_1, n_2, then n_1 is an ancestor of n_2, denoted $n_1 < n_2$, if there exists a sequence of nodes $S := (s_1, .., s_n)$ with $n \geq 1$ such that $n_1 = prt(s_1), s_1 = prt(s_2), .., s_n = prt(n_2)$.*

Consequently, it is obvious that n_1 is an ancestor of n_2, then its depth must be smaller than that of n_2. In the example of Figure 3.6 (a), node a_2^7 is an ancestor

(actually immediate ancestor or parent) of a_2^8. Also, a_2^2 is an ancestor of a_2^8 because there is a sequence of nodes $(a_2^2, a_3^3, a_3^4, a_2^5, a_2^6, a_2^7)$ connecting them together.

Definition 15 (Conflict)**.** *Two nodes n_1 and n_2 are conflict related iff neither "n_1 is ancestor of n_2" nor "n_2 is ancestor of n_1" holds.*

We recall that nodes at the same depth and having the same parent are definitely in immediate conflict. The conflict is inherited in the sense that if two events are in conflict, all successors of one are in conflict with those of the other. Consequently, two events belonging to different branches are surely in conflict.

Based on these observations, we propose an algorithm to check *conflict* and *ancestor relations* between two nodes in a sequential tree as follows:

Given two nodes n and n' having depths d and d' respectively, in a sequential tree, either process or variable one. Without loss of generality, assume that we need to decide if n is an ancestor of n' or not. It is definite that if $depth(n') < depth(n)$, n will never be ancestor of n'. Otherwise, use a *skip list* to go from n' to its predecessor at the depth $depth(n)$ (Algorithm 4.), so-called n''. If $n \equiv n''$, we have n is ancestor of n', otherwise they are in conflict (Algorithm 3).

Algorithm 3: Deciding conflict in a sequential tree

Input: n_1, n_2 at the depths d_1 and d_2 respectively
Output: *true* if $n_1 \# n_2$, otherwise *false*

```
1  if ((n1 ≠ n2) ∧ (d1 = d2))
2  |  return true;
3  else
4  |  if d1 < d2
5  |  |  find n', an ancestor of n2 at depth d1
6  |  else
7  |  |  find n', an ancestor of n1 at depth d2
8  if (n' ≡ n1) ∨ (n' ≡ n2)
9  |  return false ;
10 else
11 |  return true;
```

Backtracking through a tree can be simply performed by tracing back by its *parent* one by one which is linear to the depth of the node but it is not efficient enough when we get a deep tree of the depth of millions. We are ambitious to find out an algorithm to explore the ancestor with a tiny complexity, so the new data structure called *skip list* will be applied.

3.6.2 Data structure and efficient tree navigation

Skip list $(n_1, .., n_m)$ for a node n is a sequence of references to its ancestors at different depths based on *skip step* ss where ss is given by developers indicating the number of ancestor to skip over and n_i is an ancestor of n at the depth ss^i such that $ss^i < depth(n)$. In our very simple example in Figure 3.6, $ss = 2$ and event a_2^8 has a skip list $(a_2^6, a_5^4, 0)$.

As stated in Algorithm 4, to find an ancestor of node n at a specific depth d, we try to find the best skip step bss to jump as far as possible. $bss := max\,\{j \in \mathbb{N} : ss^j < dis\}$ where $dis = depth(n) - d$. Repeat this procedure until the best ancestor has depth d. In the example shown in Figure 3.6, we need to decide the relation between nodes a_2^8 (at depth 8) and a_1^2 (at depth 2). Skip step $ss = 2$, event a_2^8 has a skip list (a_1^0, a_2^2, a_2^4) while that of a_2^4 is (a_1^0, a_2^2). With these, we find a sequence of best ancestors to reach depth 2 from a_2^8: $a_2^8 \rightarrow a_2^4 \rightarrow a_2^2$ where we encounter a_2^2 at depth 2. Since $a_2^2 \neq a_1^2$, it is necessary that a_1^2 conflicts with a_2^8.

The complexity of the algorithm now is $\mathcal{O}(\log N)$, much better than $\mathcal{O}(N)$ if we explore the tree sequentially through nodes' parents.

Algorithm 4: Find ancestor at defined depth

Input: a node n, an depth d
Output: only one node n': $depth(n) = d$ and n' is an ancestor of n

```
1 Initially, let p point to e;
2 while (depth(p) > d) do
3 |   dis = depth(p) − d
4 |   Find the best skip step bss = max {i ∈ ℕ : dis mod ss^i = 0}
5 |   p = skiptab[ms](p)
6 end
7 return p
```

3.6.3 Causality and Conflict for events

Algorithm 3 decides causality and conflict between two nodes in a tree. How about the problem with events in the unfolding? Back to Section 3.5, two arbitrary events in unfolding exist in three situations: in the same thread tree, in the same variable tree or neither. Each event associates with two nodes corresponding its appearances in two trees. In cases (3.5) and (3.6), their relation is the same as that between their corresponding nodes. In case (3.7), based on Proposition 1 and Proposition 2, we decide the relation between corresponding nodes of maximal events where there

exists any two nodes are in conflict or causality, these events have the corresponding relation.

3.7 Conclusions

This chapter introduced our new algorithm QPOR where we propose the new concept of k-partial alternative, a quasi-optimal solution that computes alternatives in polynomial time and reduces redundant executions as well. We also used the data structure *sequential tree* to efficiently check conflict and causality between events which are required in computing alternatives.

In the next chapter, we will present our work on parallelization to take advantage of available multi-core and multi-CPU computers. We will detail the parallel algorithm achieved by partitioning the unfolding exploration into sub-works that explore maximal configurations independently.

Chapter 4

QPOR Parallelization

Contents

4.1 Introduction **55**
4.2 Motivations **56**
4.2.1 Technology for parallelism 56
4.2.2 Challenges and opportunities for QPOR parallelization . . . 56
4.2.3 Our objectives 58
4.3 Parallelization design for QPOR **58**
4.3.1 Parallel computing 58
4.3.2 General idea 59
4.3.3 Data structure 59
4.3.4 Overall algorithm 61
4.3.5 Parallel exploration process 62
4.3.6 Avoiding redundant exploration 64
4.3.7 Algorithm termination 64
4.3.8 Synchronization mechanism 65
4.4 Conclusions **65**

4.1 Introduction

Although QPOR described in Chapter 3 has proposed algorithms and data structures to efficiently explore the execution tree, we can exploit its natural structure to take advantage of current hardware features to execute quickly. Therefore, parallelization is a promising solution for QPOR to speed up the exploration.

In this chapter, we detail how the algorithm can be parallelized including work partition, parallel exploration process and synchronization mechanism.

4.2 Motivations

The parallelization of our algorithm QPOR is motivated by the natural parallelism underlying the unfolding exploration, the availability of multiple processor computers and existing works on parallelization.

4.2.1 Technology for parallelism

Most of nowadays computers have multi-core processors (several cores integrated in a single chip) [100] and supercomputers [96] even have multiple CPUs (several multi-core CPUs in multiple chips) on the same machine where each core or CPU handles a separate computation task participating to a global computation. Thanks to these features, computational work can be done by several cores in parallel to get a good performance. This hardware requires software to recognize and take advantage of these parallel processing capabilities, in other words, programs need to be parallel.

Parallel programming implements software by breaking its work into multiple chunks of work and assigning them to two or more cores (*processing units*) or several nodes. Many parallel programming models are also introduced: Bulk Synchronous Parallel (BSP) [6], directive models[81] or task models [64, 65, 63].

Multi-core features are exploited together with parallel programming to distribute many model checkers such as Divine [22, 20], SPIN [21, 61], multi-core Helena [48], distributed Helena [38] and distributed IMITATOR [11, 12].

4.2.2 Challenges and opportunities for QPOR parallelization

Using depth first search to explore the execution tree, QPOR might suffer a reduction in performance when the tree is deep since they have to sink down the deepest path and backtrack event by event to search for another branch, i.e. computing alternatives. The deeper the tree is, the more nodes there are to compute alternatives.

Moreover, computing alternatives in Algorithm 1, either k-partial alternative or alternative in an optimal algorithm, is time-consuming when there is a large number of conflicts between events. This is because the comb defined in Definition 13 built in the alternative computing procedure composes of spikes that is the set of immediate conflicting events of an event. The greater the size of `comb` (the number and the length of spikes) is, the harder enumerating combinations is. Therefore, for those that have many direct conflicting events, it is complex to compute alternatives. We are hence motivated to speed up the alternative computation.

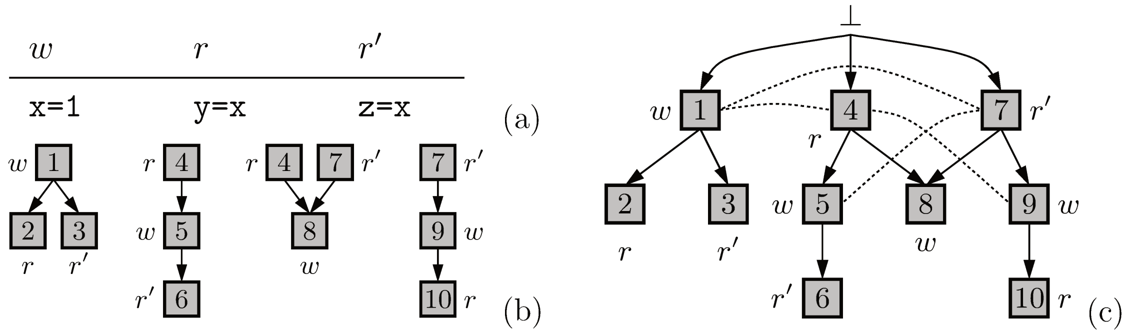


Figure 4.1: Example: (a) Program; (b) All maximal configurations; (c) Unfolding

On the other hand, we witness plenty of works that distribute POR such as POR for GPU [94, 7], Cartesian POR [57] and distributed POR [105, 89, 29, 104] and distribute unfolding such as [18, 49].

Yang et al [105] parallelize DPOR by assigning different works of exploration to different workers to execute concurrently. Each work is a depth first search in the execution tree so that the entire execution tree is organized as a collection of depth first searches. However, this algorithm faces two problems: (1) different workers may explore the same part, and (2) it has to deal with load-balancing when the size of the execution tree is big. They addressed (1) by adding nodes to the exploration frontier eagerly and (2) by simulating a central load-balancer to keep track of workers and distribute work load among them.

Simsa et al [89] propose a design called *n-partitioned depth first search* that partitions the exploration into n *fragments* with an user-defined constant n. Each of these segments is separately explored by the depth first search algorithm. The strong point of this work is that the number of fragments is a constant, thus it is possible to apply the algorithm to a large scale system.

These works on DPOR provide motivations as well as a base for us to parallelize our algorithm.

To parallelize QPOR, we aim to decompose the exploration into a collection of independent chunks of work. We observe that the exploration is organized in a tree-like structure (see Figure 4.1) where two arbitrary branches are either totally separate from each other such as $\{7, 9, 10\}$ and $\{4, 5, 6\}$ or share a prefix and start a new independent branch at the point an alternative is found, like $\{4, 5, 6\}$ and $\{4, 7, 8\}$. Despite some overlap that can be explored twice, two branches can still be explored separately without problems. Two branches $\{4, 5, 6\}$ and $\{4, 7, 8\}$, for instance, share

4. As 5 is in immediate conflict with 7 ($5 \#^i 7$), all successors of these events are totally different or in other words, their successors have no relation with each other. Therefore, these two branches can be explored in any order despite the fact that they explore 4 twice.

In another case, let us assume that the branch (configuration) $\{4, 7, 8\}$ is first explored, backtracking along this branch finds $\{4, 5\}$ at 7 and $\{7, 9\}$ at 4 as two possible alternatives. These two alternatives lead to configurations $\{4, 5, 6\}$ and $\{7, 9, 10\}$ respectively. Since all their events are totally different and have only conflicts, they can be built from the root independently from each other. This observation shows that it is possible to subdivide the exploration into multiple work units, each of which is responsible of exploring one branch corresponding to a maximal configuration of the unfolding and the most important is that they can be executed in parallel.

Based on the natural concurrency in the unfolding exploration and plenty of parallel programming models aforementioned in Section 4.2.1, it is feasible to achieve a parallelization of QPOR.

4.2.3 Our objectives

Our goals are to speed up the unfolding exploration using parallelism in a suitable parallel programming model among available models and then parallelize our sequential algorithm to obtain a parallel QPOR. The following part describes in details our parallelization.

4.3 Parallelization design for QPOR

This section first discusses briefly parallel computing as the context of our parallelization. Next, we detail the algorithm and data structure of parallel QPOR.

4.3.1 Parallel computing

Parallel computing is to use computer's resources e.g. processors, memory simultaneously to execute a computational work. The work is subdivided into independent parts that can be executed concurrently called *processing units*. In general, a *processing unit* is a set of instructions to perform a part of the overall work to achieve a subgoal (a part of the overall goal). They can be implemented as *thread* or *process*. Processing units in a program have to co-operate to produce its expected output.

Therefore, coordinating mechanisms need to be established to guarantee all processing units interact in proper manner.

Among available parallel models are shared memory and message passing. We choose shared memory for our parallelization since it is facilitated by operating systems. If the program works on shared memory in a node, then data in shared memory are directly accessible to all processing units of the system. Otherwise, if on a distributed memory system, processing units need to exchange data by sending and receiving messages over the network. It is obvious that message exchanges take more time than access to shared memory. Benchmarks MAGI [73] show a persuasive evidence where the latency on shared memory is only 0.7s while over Ethernet, that of message passing is 26.5s, even with wide bandwidth network, it is still over double of that on shared memory. Therefore, at the moment, we would use shared memory.

4.3.2 General idea

We parallelize the sequential algorithm by partitioning the unfolding exploration into subworks. Intuitively, a *subwork* is a set of instructions to explore a maximal configuration. As previously mentioned in Section 4.2.2, maximal configurations in the unfolding can be executed separately and in any order, such decomposition of exploration allows multiple subworks to be performed concurrently.

During the exploration of a branch, corresponding to a maximal configuration, a subwork also searches for alternatives to switch the exploration to new branches, so it might produce new subworks. Unlike the sequential Algorithm 1 which switches to a new branch immediately once it finds an alternative and stops backtracking the current branch, the parallel algorithm creates a new subwork to explore a branch whenever it finds an alternative and continues backtracking to search for others until hitting the root. As a result, a subwork might spawn many other subworks and these subworks can be executed by other processing units if any of them is idle. Thus, subworks are generated and removed continuously, so we need to manage them efficiently. The exploration terminates when there is no more subwork to execute.

In the next part, we detail data structures and mechanisms to manage subworks efficiently.

4.3.3 Data structure

In addition to data structures introduced in the sequential algorithm in Chapter 3, for parallelization we first introduce *subwork* structure.

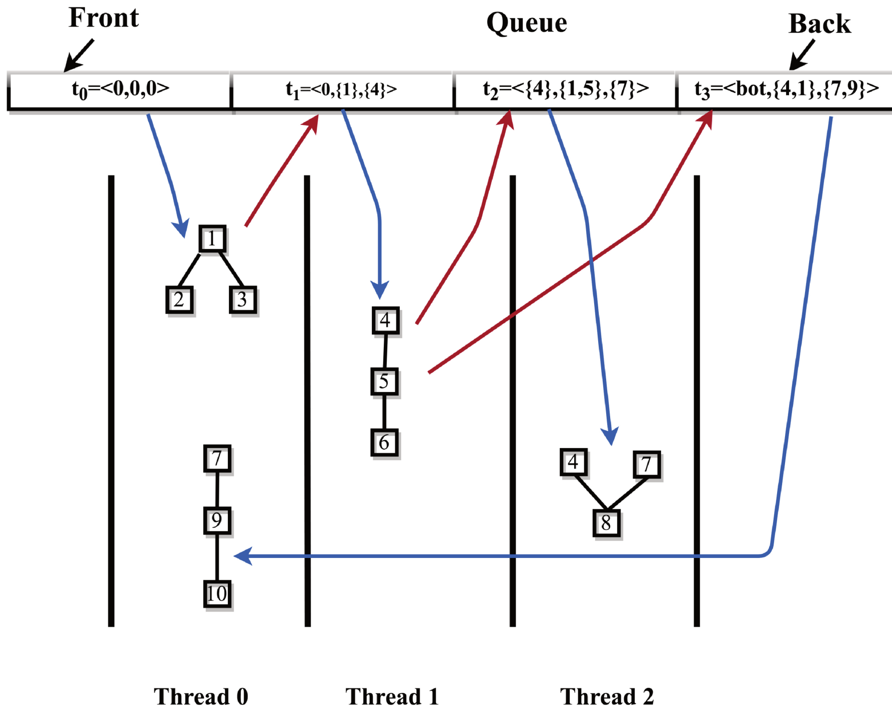


Figure 4.2: Parallel exploration

Subwork A *subwork* is a set of data necessary to produce a maximal configuration. It mainly contains the following:

- *Configuration* C: a set of events as the prefix of maximal configuration that has been explored.
- *Disable set* D: a set of events should be excluded from the execution.
- *To-add set* A: Set of events that should be included in the new configuration.

The tuple composed of C, D and A is sufficient to explore a new maximal configuration as described in Algorithm 1. Beside these data, a subwork contains other information that is useful to its execution such as trail (the sequence in which events are added to the configuration), replay (set of events that is included in the maximal configuration), etc. Subworks are generated and removed continuously during the exploration, so we use another data struture *queue of subworks* to manage them efficiently. Tuples t_i with $i \in [0..3]$ in Figure 4.2 are some examples of subworks.

Queue of subworks The collection of subworks of exploration needs to be managed in such a way that processing units can easily take out a subwork to perform or push a

new one in. Among various data structures available such as stack, linked list, queue, etc., we choose to use *queue*, the container of objects conforming to first-in-first-out (FIFO) principle.

As defined previously, data enclosed in a subwork are sufficient to generate a maximal configuration, that means a subwork does not need any information from other peers for its execution. Therefore, one subwork can be taken and executed before, after or even at the same time as others, that means the order of executing subworks does not matter in the parallel algorithm. That also means that a processing unit can take an arbitrary subwork in the collection to execute. Linked lists allow random access but each node of the list has to maintain links to its neighbours which is unnecessary.

On the other hand, both queue and stack are sequential access structures, the principle of FIFO of queue, i.e. inserting at the front and removing at the back, helps reduce access conflicts between processing units. The front and the back only unify when the queue has a unique element or is empty which are easy to handle cases.

With all above reasons, we have chosen a *queue* to maintain the collection of subworks of the exploration in our algorithm. In the example in Figure 4.2, the queue has four subworks in total: t_0, t_1, t_2 and t_3.

4.3.4 Overall algorithm

Algorithm 5: General parallel unfolding-based POR exploration

```
1  Create N processing units with an user-defined constant N.
2  Q := {t0}
3  Procedure Explore()
4    while Q is not empty or there is at least one working thread do
5      Find an available processing unit thd
6      if Q is not empty
7        tsk ← front(Q)
8        pop(Q)
9        Assign tsk to thd
10       thd calls ExploreOneMC(tsk)
11     end
12   end
```

Algorithm 5 presents the main procedure to explore the execution tree in a parallel manner. At the beginning, a number of processing units are spawned and queue Q is initialized with subwork $t_0 = \langle \bot, \emptyset, \emptyset \rangle$. While there is at least one subwork in the

queue, an idle processing unit takes the first element in the queue (Line 7) and calls function `ExploreOneMC()` to explore one maximal configuration (Line 10). To be sure that while one processing unit is taking, other processing units do not take the last element in the queue, the emptiness needs to be checked again (Line 6). The exploration terminates when there is no subwork anymore. Checking whether the queue is empty or not is not enough. If the queue is empty but some processing unit is working, potential subworks are probably generated. Therefore, to assure there is no subwork, the algorithm verifies that the queue is empty and all processing units are idle (Line 4).

The worker function `ExploreOneMC()` in Algorithm 6 explores one maximal configuration and generates new subworks if an alternative exists. It takes parameters C, D, A from the *subwork* to produce the configuration. First, it adds all extensions to the unfolding (Line 4). Then, it chooses an event (Line 9 or Line 7) to extend the configuration by a recursive call to `ExploreOneMC` (Line 11) until there is none enabled. It then backtracks along the trail of events by returning to the calling function (Line 5) to search for alternatives by `Alt(`C,D`)` (Line 12). We recall that only lock events could have conflicting events, so it is uniquely possible to find an alternative at a lock event. At the event e, the function `Alt(`$C,D \cup \{e\}$`)` uses either k-partial alternatives defined in Definition 12 or alternatives defined in Definition 11. At most one alternative is chosen although there are several potential ones. Any alternative found leads to another maximal configuration, so a new subwork is generated (Line 13) and pushed to the shared queue of waiting subworks (Line 14).

4.3.5 Parallel exploration process

The parallel unfolding exploration is executed as follows:

1. Initialize the queue Q with the first subwork: $t_0 = \langle \bot, \emptyset, \emptyset \rangle$.

2. t at the front of Q is taken by a thread, executes the worker function `ExploreOneMC()`. `ExploreOneMC()` finds a enable event to extend. When there is no more enabled events, it stops. This produces the first maximal configuration.

3. Backtrack along the trail of the current maximal configuration: At each event e_i in the trail, if an alternative is found, $A_i =$ `Alt(`$C_i, D_i \cup \{e_i\}$`)`, a subwork $t_i = \langle C_i, D_i, A_i \rangle$ is created and pushed to the queue Q.

4. The backtracking stops when it meets the root.

Algorithm 6: One maximal configuration exploration

```
1  Input: A subwork = ⟨C, D, A⟩ with C: current configuration, D: disable set,
     A: add set, U: overall unfolding
2  Output: One maximal configuration explored. A subwork is probably
     generated and added to Q.
3  Procedure ExploreOneMC(C, D, A)
4      Add ex(C) to U
5      if ena(C) = ∅  return
6      if A = ∅
7          Choose e from ena(C)
8      else
9          Choose e from A ∩ ena(C)
10     end
11     ExploreOneMC(C ∪ {e}, D, A \ {e})
12     if ∃J ∈ Alt(C, D ∪ {e})
13         Create new subwork t_i = ⟨C, D ∪ {e}, J \ C⟩
14         Push t_i to Q
```

5. In the meantime, another thread looks for a subwork in Q if it is idle. If there exists one subwork, it takes it for processing.

6. The whole exploration stops when Q is empty and all the threads are idle.

Let us take the program in Figure 4.1 to illustrate the process. Its parallel exploration is shown in Figure 4.2. Assume that the program has a pool of three threads. The queue of subworks is initialized by $t_0 = \langle \bot, \emptyset, \emptyset \rangle$. At the beginning, all threads are idle, so assume $Thread0$ takes t_0 and discovers the first maximal configuration $\{1, 2, 3\}$. Backtracking this configuration, a new alternative $\{4\}$ is found by $\texttt{Alt}(\bot, \{1\})$, so a subwork $t_1 = \langle \bot, \{1\}, \{4\} \rangle$ is spawned and pushed to the queue. t_1 is then taken by $Thread1$ to produce the configuration $\{4, 5, 6\}$. From $\{4, 5, 6\}$, two alternatives are found at 5 and 4, generating two subworks $t_2 = \langle \{4\}, \{1, 5\}, \{7\} \rangle$ and $t_3 = \langle \bot, \{1, 4\}, \{7, 9\} \rangle$ respectively. These subworks are concurrently taken by $Thread2$ and $Thread0$ ($Thread0$ has finished working on subwork t_0 earlier and now is idle). Executing t_2 and t_3 produces corresponding two maximal configurations $\{4, 7, 8\}$ and $\{7, 9, 10\}$. Backtracking these two configurations finds no alternative, which means there is no new subwork. By the time $Thread0$ and $Thread2$ finish executing configurations, all the threads are idle and the queue is also empty, so the exploration terminates.

It is obvious that the first subwork (t_0 in this example) must be executed first by one thread to produce the first configuration. The exploration is only able to continue if backtracking this configuration results at least one alternative. Otherwise, there is nothing to continue. We observe that producing the first maximal configuration (done by font-end Steroids) and backtracking it to find alternatives are sequential work that programmers cannot do anything to speed up (certainly, except some algorithmic improvements). According to Amdhal's law [9], the overall execution time cannot be less than the amount spent doing these works. We might gain speedup in exploring the next configurations. Amdhal's law also states that the more cores we have, the less time is spent on doing parallel works. But in our case, the parallelization only makes sense when backtracking configurations results in more than one alternative, i.e. there is more than one subwork in the queue. If we have fewer subworks than cores, the speed up is not as much as expected. In the above example, we only have threads working concurrently when the queue has two subworks t_2 and t_3. Before that, as there is only one subwork in the queue at a time, it works sequentially.

4.3.6 Avoiding redundant exploration

In the example of Figure 4.1, two branches $\{4, 5, 6\}$ and $\{4, 7, 8\}$ share a prefix $\{4\}$, so the two corresponding subworks search for alternatives at 4 two times, which means they risk to find the same alternative twice. To prevent duplication, we propose to use `replay`. A `replay` is a set of references to events in C and A of a subwork that represents the prefix of the branch to be explored. If two subworks tsk_1 and tsk_2 have two prefixes rpl_1 and rpl_2 respectively, such that $rpl_1 = rpl_2$, they definitely explore the same branch. We maintain a set `replays` of all replays of the exploration that is referred whenever there are new alternatives. Once an alternative is found, a replay is computed based on its C and A. This replay is then checked for existence by comparing it to all existing replays in the set `replays`. If the replay already existed, no new subwork is generated. The function continues to search for other subworks.

This set should be shared among all threads, so it is necessary to install a lock for this data structure.

4.3.7 Algorithm termination

The exploration terminates if the number of subworks is finite and there is no deadlock, e.g. a task executes a subwork infinitely while the queue is empty. As we assume

that all input programs are terminating, all executions of the program must terminate. Each subwork corresponds to one execution of program, so it is finite. Thus, we can be sure that there is no deadlock caused by infinite subworks. Moreover, we observe that the number of mutex locks in a multithreaded program is finite, so is the number of mutex locks in a maximal configuration, which implies that the process of backtracking terminates and produces a finite number of new subworks. If no maximal configuration is duplicated, the whole exploration terminates. The duplication checking algorithm in Section 4.3.6 guarantees the termination.

4.3.8 Synchronization mechanism

Synchronization is essential for parallel programs that make concurrent accesses to shared memory to avoid *race condition*. Race condition happens when more than one thread (one of them is a write operation) is accessing the same memory location at a particular point of time. In our algorithm, there are two shared segments of data: the queue of subworks Q and the unfolding U.

Q is retrieved when a processing unit requires a subwork (pop out the queue) or creates a new subwork (push back to the queue). The easiest way to control the access to Q is to implement a lock although some lockless queues exist and can also be chosen.

The shared unfolding is continuously accessed by tasks to add events and update their data. Events in the unfolding are grouped into processes and amending an event to a process has no dependence to any others, so it is sufficient to use a lock for each process. This lock is a read-write lock that allows read operations access the process simultaneously and excludes all others if the operation is a write.

4.4 Conclusions

In this chapter, we have specified the concurrency underlying the execution tree that is the independence among branches. This property allows us to parallelize our sequential algorithm QPOR by exploring maximal configurations concurrently. The multi-core and multiple CPUs features in computers and the availability of parallel programming models facilitates the parallelization. We presented as the main part our parallel algorithm based on a shared memory model which partitions the unfolding exploration into subworks. Each subwork is responsible for exploring different maximal configurations concurrently to others. An algorithm of checking duplication are also designed to assure the termination of the parallel algorithm.

The next chapter will describe how we implement both of our sequential and parallel algorithms and experiments that have been conducted to evaluate the performance of the algorithm as well as our new tool.

Chapter 5

Implementation and experiments

Contents

5.1 Introduction **67**

5.2 DPU - Dynamic Program Unfolder **68**

5.2.1 Front-end 69

5.2.2 Back-end 69

5.3 Sequential implementation **71**

5.4 Parallel implementation **72**

5.4.1 OpenMP 74

5.4.2 Algorithm implementation 76

5.5 Experiments **77**

5.5.1 Comparison to SDPOR 77

5.5.2 Evaluation of the Tree-based Algorithms 79

5.5.3 Evaluation Against the State-of-the-art on System Code . . 81

5.5.4 Profiling a Stateless POR 83

5.6 Conclusions **83**

5.1 Introduction

The goal of this chapter is to provide implementation details for sequential and parallel QPOR algorithms. Our main contributions are as follows:

- We implement sequential QPOR described in Chapter 3 into a tool called DPU using specialized data structures.
- We implement the parallel algorithm described in Chapter 4 as a parallel version of DPU using OpenMP API.

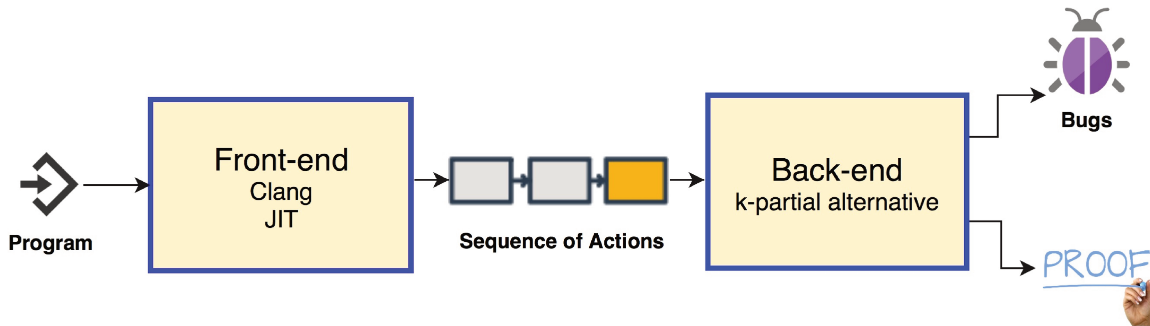


Figure 5.1: DPU architecture

- We also conduct experiments on selected benchmarks to evaluate the efficiency of the approaches and compare them with some testing and verification tools.

5.2 DPU - Dynamic Program Unfolder

We introduce in this section the general structure our new tool called DPU [1] (Dynamic Program Unfolder).

The input of DPU is multi-threaded C programs that are assumed to be data-deterministic. Data-deterministic programs are those that always produce the same output from given input and resources. That means the only source of non-determinism in the program's execution is the order in which concurrent thread statements are interleaved. As a result, all sources of non-deterministic execution (e.g., command-line arguments, input files) need to be fixed before running the tool.

We also assume that all input programs are data race free. For fair comparison against other tools that take programs with data race, we implemented a module that detects data races before being processed by DPU. That is out of scope of this thesis.

To pre-process input programs, we aim to make use of existing compiler frameworks to get an optimized and more easy-to-analyze representation, so LLVM is used.

As shown in Figure 5.1, our tool is composed of two parts: Front-end called *Steroids* and Back-end. Steroids JIT-compiles the input program and collects its information to produce a stream of actions to pass to the back-end. The back-end in turn converts that stream to events, then performs unfolding exploration to detect any defect if it exists.

[1] https://github.com/cesaro/dpu

5.2.1 Front-end

The front end, called Steroids [2], is a library for dynamic analysis of POSIX C programs. It takes LLVM compilation (IR) produced by JIT compiler and optimizer as input, then analyzes it to get information about program execution to produce a stream of actions that describe that execution. The actions involved are thread operations such as thread creation, lock and unlock mutex, assertion violations and calls to abort. An action of the stream is described as follows:

type: Type of operations executed by the program such as:

- THCREAT: a call to `pthread_create`
- THJOIN: a call to `pthread_join`
- THEXIT: a call to `pthread_exit`
- MTXLOCK: a call to `pthread_mutex_lock`
- MTXUNLK: a call to `pthread_mutex_unlock`
- THSTART: the beginning of a thread execution.

Addr: The address of memory allocated to store its values, usually called *variable.*

Val: The values stored in *addr* at a time.

Actions with these information are passed to the back-end for more operations.

5.2.2 Back-end

DPU back-end is the model checker that explores the unfolding to detect any existing defects. It provides options corresponding to the alternative algorithms based on user-provided constant k where:

- $k = 0$: optimal algorithm that implements the algorithm in Rodríguez et al [87].
- $k = -1$: SDPOR that implements the non-optimal algorithm in Abdulla et al [2].
- $k = 1$: 1-partial alternative algorithm that finds a clue of alternative to the last event in the disable set D.
- $k \geq 2$: k-partial alternative algorithm (Definition 12) in Algorithm 1 that finds a clue of alternative to k events in disable set D.

[2] https://github.com/cesaro/steroids

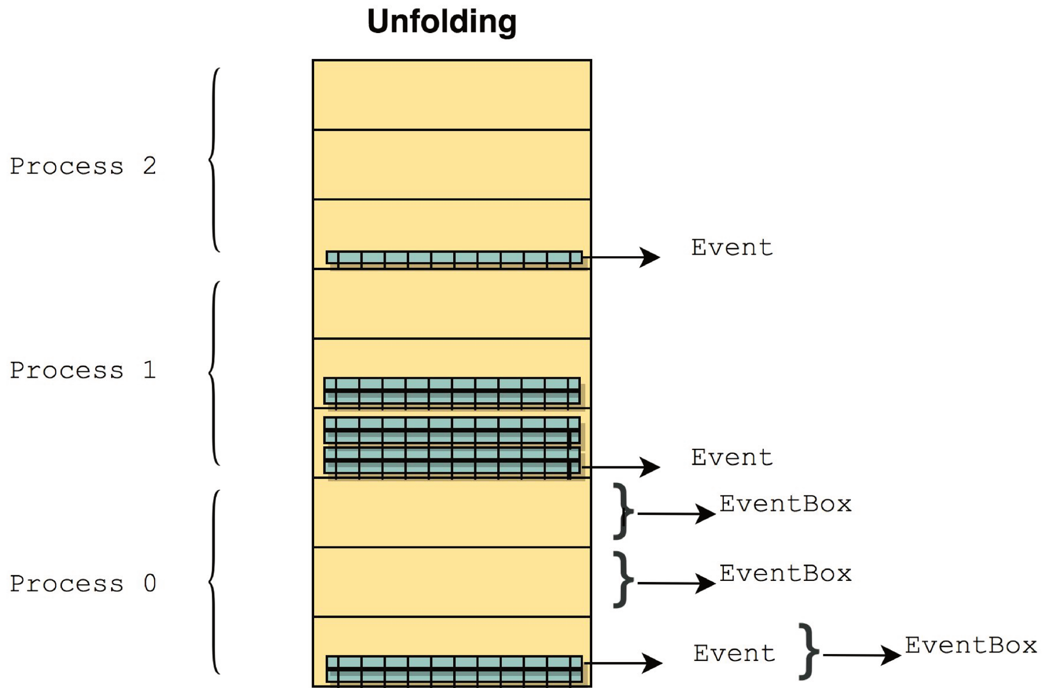


Figure 5.2: Unfolding memory alignment

Data structure An *Unfolding* is a set of *events* explored during the exploration, each of which belongs to a *process*. The number of events of an unfolding is potentially huge while accesses to events are frequent, so we propose a specialized alignment of memory to organize events in an unfolding for easy retrievals.

As illustrated in Figure 5.2, the address space allocated to an unfolding is divided into same-size segments (size of *proc_size*) based on a given maximal number of process. Each segment is for a `Process` which also subdivided into boxes called `EventBox`. An `EventBox` is a container that has the same fixed size and event boxes in a process are filled with events one by one from the starting address of the process. This division is based on process and event size. Events have a pointer linking to the next one in the `EventBox` and the last event in a box points to the first in the next `EventBox`. The address of a process or `EventBox` is the location of its first event.

With this memory alignment, each `Process`, `EventBox` or `Event` has a unique address so that it is easily located in the memory. For example, given the address of the allocated space for the unfolding is X, the size of `Process` is PS, of `EventBox` is EBS, of `Event` is ES, event e_{ij}, event j of process i, has the address $X+j*ES+i*PS$. That means we can trivially retrieve any event in any process.

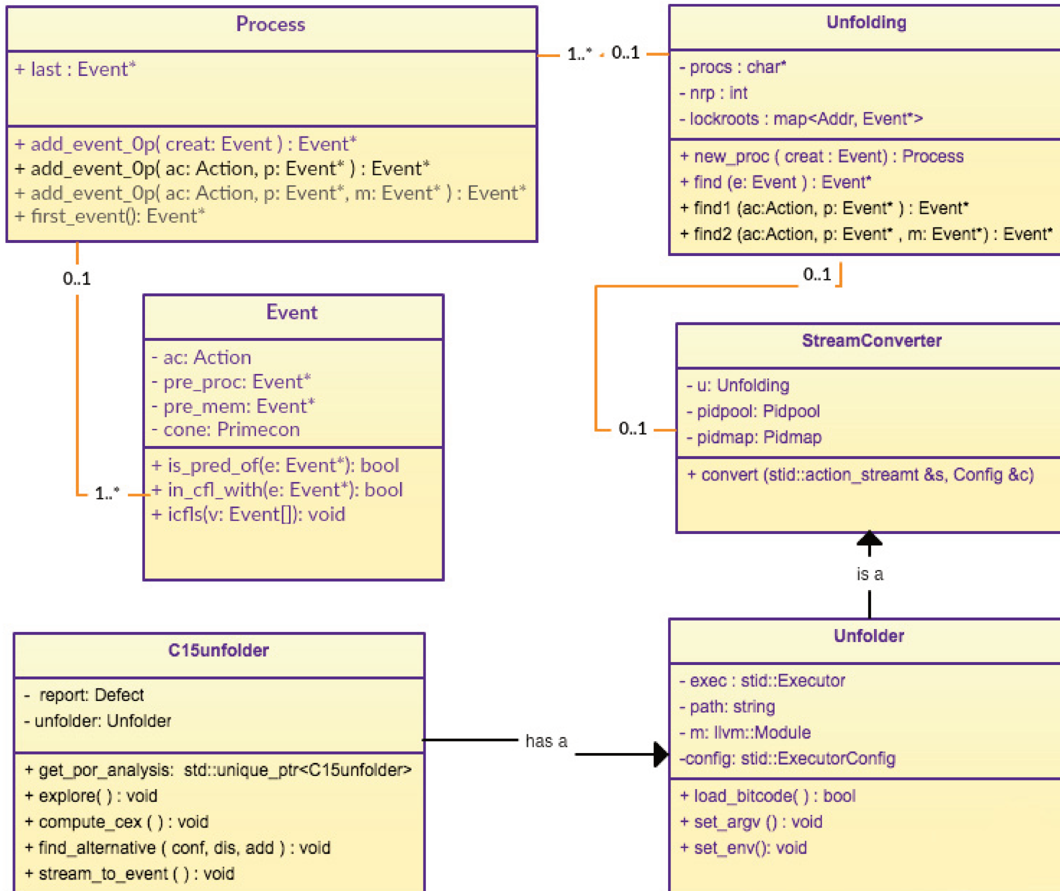


Figure 5.3: DPU class diagram for sequential implementation

5.3 Sequential implementation

We have implemented the sequential algorithm in C and C++ using the specialized data structure described in Section 5.2.2 and released a tool available on github [3].

Main algorithm As shown in Section 5.3, the exploration algorithm in Algorithm 1 is mainly implemented in an object `C15unfolder`. This uses an `Unfolder` that has an executor from the front-end Steroids to produce program executions. By generating an execution, the Unfolder produces a complete maximal configuration, so it is unnecessary to compute the enable set ($en(C)$ in Algorithm 1). Having a maximal configuration, conflicting extensions and alternatives are computed. Both of these computations requires backtracking. To easily backtrack along the configuration, we store a trail that is the sequence of events added to the configuration in a way that

[3] https://github.com/cesaro/dpu

the last event added to the configuration is at the top of the trail. The recursive calls of `Explore()` in Algorithm 1 are realized by a loop that stops when no alternative is found.

Computing alternatives We implemented both optimal and k-partial alternative algorithms. With $k = 1$, the algorithm computes the 1-partial alternative in Definition 12 to the last event added to the disable set, instead of choosing randomly one event in D. $k = 0$ is for the optimal algorithm that computes alternative as in Definition 11. A `Comb` (Definition 13) is simply implemented as a set of spikes which is also a set of events. Finding alternatives is enumerating over spikes to find a conflict-free and causal-closed combination. If such a combination exists, the resulting alternative is a merge of configuration C and J where J is an union of local configurations of all events in the combination.

Conflicting extensions Our tool computes conflicting extensions by finding events in immediate conflict with each event in the maximal configuration. Thanks to the fact that only mutex lock events are in conflict with each other (when they request a lock), we apply Algorithm 2 to only mutex events. To compute conflicting events for a lock event e, we use a while loop to visit all lock and unlock events in e's local configuration to find a predecessor that is able to combine with e's predecessor in the process to form a new event. The new event shares the predecessor in the process with e, so it is in conflict with e.

Sequential tree The data structure *sequential tree* described in Section 3.6 is implemented in MultiNode, an array of two Nodes corresponding to references to appearances of an event in thread and mutex trees. Since only lock/unlock events (MTXLOCK and MTXUNLK) involved in mutexes, only lock/unlock events have two nodes, others have only one Node for their thread reference. Each Node stores a reference to its immediate ancestor and its position in the tree (the depth) to compute the ancestor relation and conflict.

The skip list used to jump over the tree is implemented as a list of Nodes at a skip step based distance. The size of this list depends on the depth of node.

5.4 Parallel implementation

As discussed in Chapter 4, we chose the shared memory model for our parallel implementation. A variety of shared memory parallel programming languages or libraries

are available such as Pthread (POSIX thread) [86], OpenMP [81] or Cilk++ [64] and TBB [64] .

Pthread [86] or POSIX thread library is an thread API for C/C++. It provides operations to create, destroy, schedule threads and data management and synchronization. A mechanism to manage a pool of threads must be implemented by programmers.

OpenMP [81] is an API for multithreaded applications with a set of library routines, compiler directives and environment variables that support shared memory parallel programming in Fortran, C and C++. Interactions between threads are realized via write/read operations to the shared address space.

Intel Cilk [64] is an extension of C/C++ language supporting data and task parallelism. It provides automatic management of parallel execution such as load balancing by its own runtime environment.

TBB [65] is a data and task based shared memory programming library for C++ applications. It provides generic parallel algorithms, concurrent containers, local storage, synchronization primitives, etc. Thanks to generic parallel algorithms, developers avoid starting parallel application from scratch.

StarPU [15] is a middleware to schedule tasks over multi-core GPUs and distributed systems. It infers the dependency graphs between the tasks statically or dynamically.

We chose to use OpenMP for the following features:

- The language used in our sequential tool DPU is C/C++, so the selected library or API should be compatible with this language so that we can modify the existing code rather than write it from scratch. All of the libraries and API support C/C++, StarPU allows modification of sequential code but it would require significant modification and restructuration to define codelets. With OpenMP, it takes less effort to create a parallel program from a sequential one by embedding OpenMP directives in C/C++ code.

- The interface provided by OpenMP is richer than the one provided by Cilk which is simple and less flexible.

- OpenMP is highly portable as it is supported by almost all compilers, for C/C++ compilers such as GCC, Clang, etc.

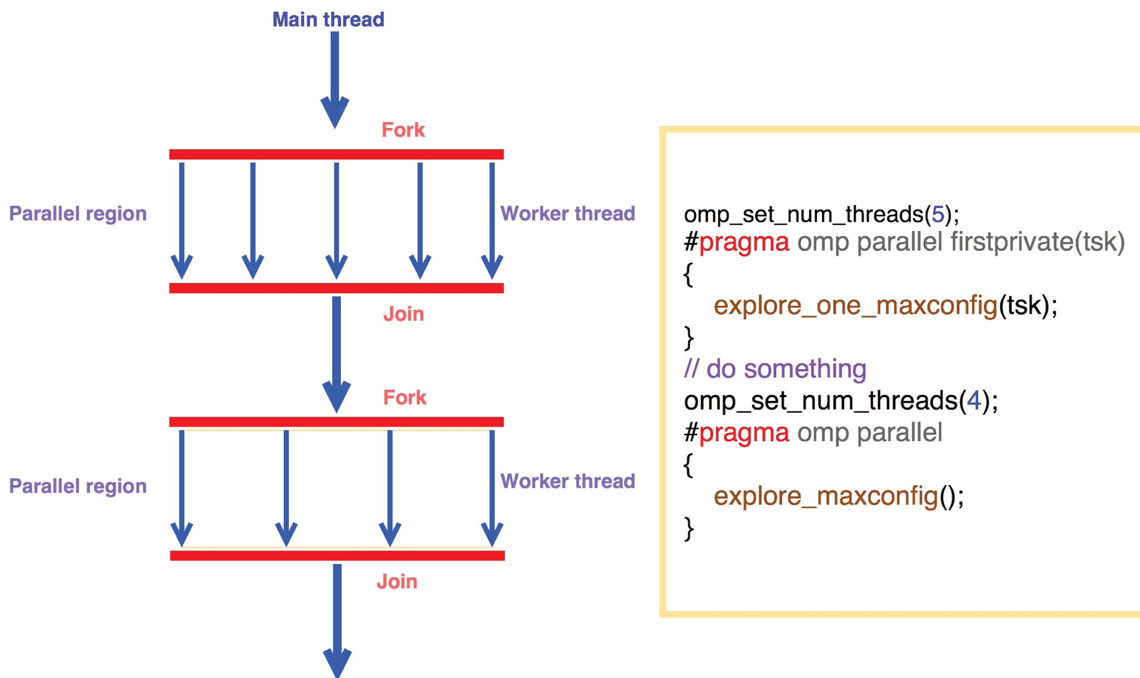


Figure 5.4: OpenMP execution model

- OpenMP is also flexible since its compiler's #pragma can be ignored if OpenMP is not supported by the compiler and then the program still behaves correctly, certainly without any parallelism.

The following part discusses more details about OpenMP and its features we are going to exploit in our parallel tool.

5.4.1 OpenMP

A program with OpenMP code works in `fork-and-join` model since it has interleaving sequential and parallel sequences of instructions. An OpenMP [99] program starts as a single thread, so-called *master thread*. The master thread executes sequentially until it encounters the first *parallel region*.

A *parallel region* is a block of code starting by parallel directive `#pragma omp parallel`. The code in this region can be executed simultaneously by multiple threads. It operates in a mechanism of `fork-and-join` as demonstrated in Figure 5.4. A thread pool is created at the beginning of the program, or just before the parallel region. At `fork`, all statements enclosed by the parallel region are executed by these threads in parallel. `join` is considered a implicit barrier where all threads finish their executions

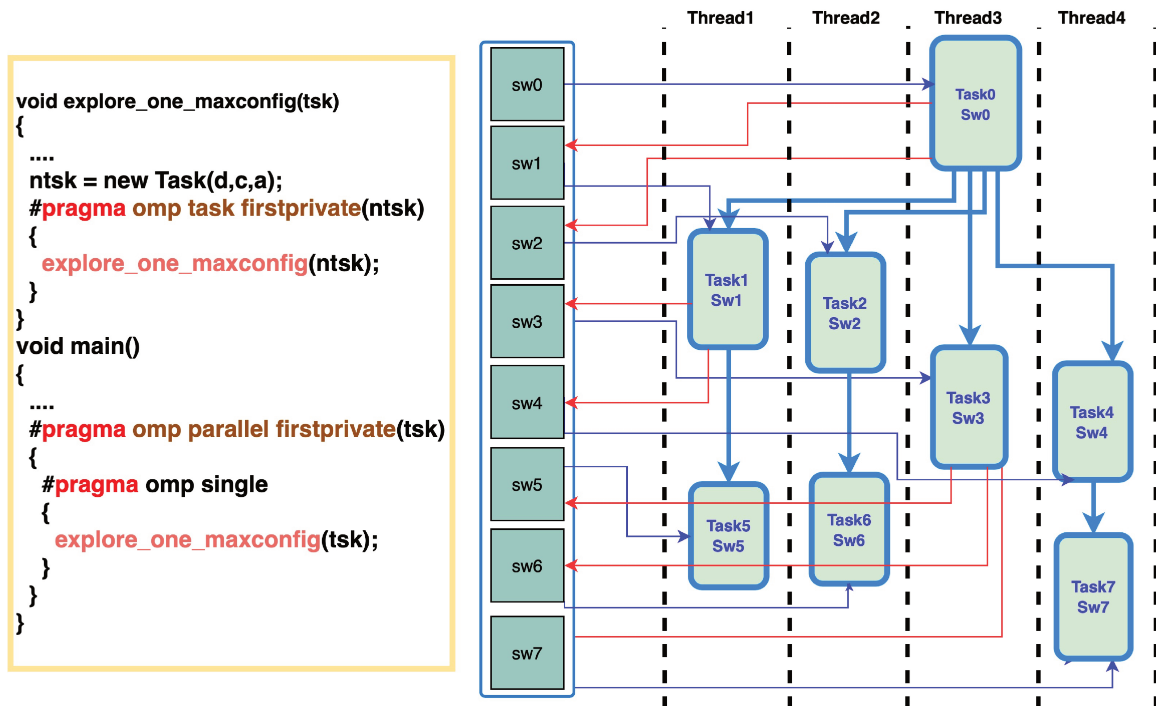


Figure 5.5: Execution of tasks

of the statements in the region, then synchronize and terminate. The execution comes back to the master thread.

This execution model allows us to easily embed parallel regions to the source code of the sequential implementation to achieve a parallel one.

As pointed out in Algorithm 5, threads in our program execute the whole work (a block of code) multiple times, rather than execute different parts of a block concurrently. They also access a shared data frequently. Therefore, low level synchronization using locks is suitable for our implementation.

OpenMP Task construct `Task` based parallelism has been in since OpenMP 3.0 to allow programmers to parallelize irregular problems such as unbounded loops, recursive algorithms, etc. A task is a structured block of code along with instructions for threads to allocate data as they encounter a task. Encapsulating both instructions and data, tasks are independent units of work, so their executions are possibly in parallel.

The OpenMP runtime system provides a mechanism to automatically schedule the set of tasks. They can be executed immediately after being generated or be deferred until later. A task being executed by a thread can be suspended and continued later by the same or a different thread.

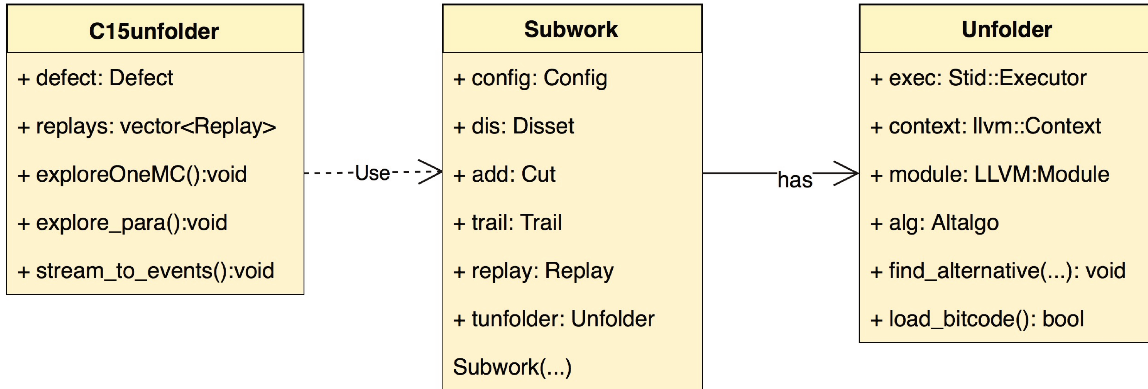


Figure 5.6: Part of class diagram for parallel implemention

We use OpenMP task to execute *subwork* of exploring a maximal configuration in Algorithm 5. Accordingly, the function `ExploreOneMC()` can be seen as a worker function while `Explore()` is the main function. We do not need to handle tasks management since OpenMP provides an automatic task scheduling mechanism.

The following section shows our parallel implementation details using OpemMP.

5.4.2 Algorithm implementation

This part details the implementation with OpenMP embedded in C/C++ source code of our parallel algorithm.

We keep the main class `C15unfolder` from the sequential implementation which takes the responsibility of getting input and outputing the statistic data to users. Unfolder is not a part of C15unfolder any more, that means threads do not share one Unfolder. Instead, each task is assigned an unfolder that generates an execution based on the $\langle C, D, A \rangle$ of the subwork it receives. This is the point that makes our implementation parallel. Other works such as finding alternatives, computing conflicting extensions also move from `C15unfolder` to `Unfolder` because they are all computed based on the current maximal configuration.

Main algorithm The master thread executes the main algorithm by the function `Explore()` in Algorithm 5 that takes care of setting up the parallel environment and exploring the first maximal configuration. It creates other tasks to continue the exploration by worker threads. At the beginning, an arbitrary thread out of the thread pool can be chosen to run the main algorithm.

Worker algorithm We use OpenMP task to implement subworks described in Algorithm 5. Once a new alternative is found, a subwork is created and an OpenMP task undertaking that subwork is spawned as well. This queue of tasks is implicitly managed by OpenMP runtime system. Any idle thread can take a task from the queue and execute `ExploreOneMC` as its worker function. Ruuning worker function with the data provided by subwork might produce a lot of new tasks. The more tasks there are in the queue, the more work can be executed in parallel. Since OpenMP provides task scheduling mechanism, we do not need to maintain a queue of subworks as in Algorithm 5.

Synchronization We ensure mutual exclusion to shared unfolding with locks. As specified in Section 4.3.8, each process of unfolding has a lock, so whenever access to the unfolding is required to create and update an event, the lock of the corresponding process is set. Only accesses to events in the same process need to be mutually excluded.

5.5 Experiments

We have conducted experiments on the sequential implementation to compare the performance of our QPOR tool against state-of-art model checking and testing tools, to evaluate the new notion k-partial alternatives as well as the efficiency of the tree structure for computing conflict and causality.

5.5.1 Comparison to SDPOR

In this section we answer the following experimental questions:

- How does QPOR compare against SDPOR?
- For which values of k do k-partial alternatives yield optimal exploration?

Benchmark selection. We use realistic, data-race free POSIX thread programs that expose complex thread synchronisation patterns, allowing to highlight the differences between SDPOR and QPOR. All the benchmarks are put on the same repository with the tool on github [4]. The benchmarks include a job dispatcher (dispatcher.c), a multiple-producer multiple-consumer scheme (mulprodcon.c), parallel computation of π (pthread_pi_mutex.c), and a thread pool (poke.c). Each program contains between 2

[4] https://github.com/cesaro/dpu/tree/master/experiments/cav18/bench

Benchmark			DPU (k=1)		DPU (k=2)		DPU (k=3)		DPU (optimal)		NIDHUGG		
Name	Th	Confs	Time	SSB	Time	SSB	Time	SSB	Time	Mem	Time	Mem	SSB
DISP(5,2)	8	137	0.8	1K	0.4	43	0.4	0	**0.4**	37	1.2	33	2K
DISP(5,3)	9	2K	5.4	11K	1.3	595	1.0	1	**1.0**	37	10.8	33	13K
DISP(5,4)	10	15K	58.5	105K	16.4	6K	10.3	213	**10.3**	87	109	33	115K
DISP(5,5)	11	151K	TO	-	476	53K	280	2K	**257**	729	TO	33	-
DISP(5,6)	12	?	TO	-	TO	-	TO	-	TO	1131	TO	33	-
MPAT(4)	9	384	0.5	0	N/A		N/A		**0.5**	37	0.6	33	0
MPAT(5)	11	4K	2.4	0	N/A		N/A		2.7	37	**1.8**	33	0
MPAT(6)	13	46K	50.6	0	N/A		N/A		73.2	214	**21.5**	33	0
MPAT(7)	15	645K	TO	-	TO	-	TO	-	TO	660	**359**	33	0
MPAT(8)	17	?	TO	-	TO	-	TO	-	TO	689	TO	33	-
MPC(2,5)	8	60	0.6	560	0.4	0			**0.4**	38	2.0	34	3K
MPC(3,5)	9	3K	26.5	50K	3.0	3K	1.7	0	**1.7**	38	70.7	34	90K
MPC(4,5)	10	314K	TO	-	TO	-	391	30K	**296**	239	TO	33	-
MPC(5,5)	11	?	TO	-	TO	-	TO	-	TO	834	TO	34	-
PI(5)	6	120	**0.4**	0	N/A		N/A		0.5	39	19.6	35	0
PI(6)	7	720	**0.7**	0	N/A		N/A		0.7	39	123	35	0
PI(7)	8	5K	**3.5**	0	N/A		N/A		4.0	45	TO	34	-
PI(8)	9	40K	48.1	0	N/A		N/A		**42.9**	246	TO	34	-
POL(7,3)	14	3K	48.5	72K	2.9	1K	**1.9**	6	1.9	39	74.1	33	90K
POL(8,3)	15	4K	153	214K	5.5	3K	**3.0**	10	3.0	52	251	33	274K
POL(9,3)	16	5K	464	592K	9.5	5K	**4.8**	15	4.8	73	TO	33	-
POL(10,3)	17	7K	TO	-	17.2	9K	**6.8**	21	7.1	99	TO	33	-
POL(11,3)	18	10K	TO	-	27.2	12K	**9.7**	28	10.6	138	TO	33	-
POL(12,3)	19	12K	TO	-	46.3	20K	**13.5**	36	16.4	184	TO	33	-

Table 5.1: Comparing QPOR and SDPOR. Machine: Linux, Intel Xeon 2.4GHz. TO: timeout after 8 min. Columns are: Th: nr. of threads; Confs: maximal configurations; Time in seconds, Memory in MB; SSB: Sleep-set blocked executions. N/A: analysis with lower k yielded 0 SSBs.

and 8 assertions, often ensuring invariants of the used data structures. All programs are safe and have between 90 and 200 lines of code.

We do not use SV-COMP benchmarks because almost all of them contain very simple synchronization patterns, where QPOR and SDPOR will perform similar explorations. DPU is able to exhaustively check almost all benchmarks from the SV-COMP'17 (adapted to remove data-races) and to find all bugs present in them (see tool website).

Tool selection. In this experiment, we use NIDHUGG [1], an efficient implementation of SDPOR for multithreaded C programs, that has recently compared favourably [3] against CBMC [8], a well established verification tool.

Analysis. Table 5.1 presents our experimental results. We run k-partial alternatives with $k \in \{1, 2, 3\}$ and optimal alternatives ($k = 0$). The number of sleep-set blocked executions (SSBs) dramatically decreases as k increases. POL benchmark

shows the most significant decline in the number of SSBs. Let us take the instance POL(9,3) for example. It decreases from 592K with $k = 1$ to 5K with $k = 2$ and only 15 SSBs with $k = 3$. The number of SSBs decreases hundred of thousands times from $k = 1$ to $k = 3$. With $k = 3$ almost no instance produces SSBs (except MPC(4,5)). All instances are explored optimally with $k = 4$ (not shown owing to space constraints). Programs with simpler synchronisation patterns, e.g., the Pi and MPAT benchmarks, are explored optimally both with $k = 1$ and by SDPOR, while more complex synchronisation patterns require $k > 1$.

Overall, if the benchmark exhibits many SSBs (MPC(3,5) for example), the runtime reduces as k increases, and optimal is the fastest option.

MPC(3,5) takes 26.5 seconds with $k = 1$ but only 3 seconds with $k = 2$. The optimal is the fastest with 1.7 seconds. However, when the benchmark contains few SSBs (cf., Mpat, Pi), k-partial alternatives can be slightly faster than optimal POR, an observation inline with previous literature [2]. Our explanation for this observation is the fact that when the comb is large and contains many solutions, both optimal and non-optimal POR will easily find them, but optimal POR will spend additional time constructing a larger comb. As a result, optimal POR could benefit from a lazy construction strategy for the comb.

Nidhugg only outperforms Dpu in the benchmark Mpat which uses a simple synchronization pattern. DPU is faster than Nidhugg in the majority of the benchmarks because it can greatly reduce the number of SSBs. In the cases where both tools explore the same set of executions, DPU is in general faster than Nidhugg as it JIT-compiles the program while Nidhugg interprets it. Since Nidhugg is aware of data-races (and attempts to revert them) while DPU assumes the program is data race free, we enforce that our benchmarks are data-race free for a fair comparison.

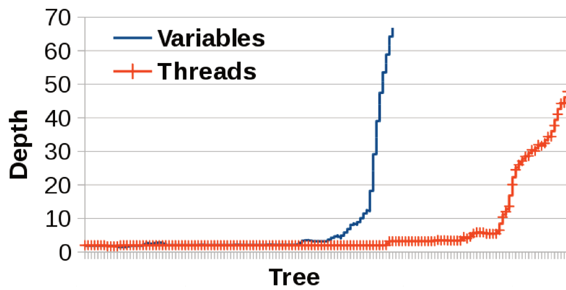
In contrast with the observations implied in previous experiments [2, 1], the results in Table 5.1 show that SSBs can occur in very small benchmarks (Mpc(2,5) and disp(5,2)) and can dramatically slow down the operation of SDPOR.

5.5.2 Evaluation of the Tree-based Algorithms

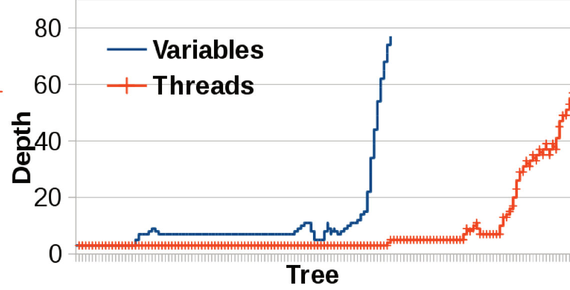
We now evaluate the efficiency of our tree-based algorithms from Section 3.6 answering:

(a) What are the average/maximal depths of the thread/variable sequential trees?

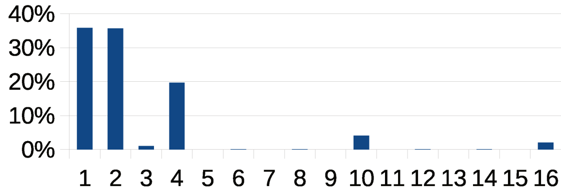
(b) What is the average depth difference on causality/conflict queries?



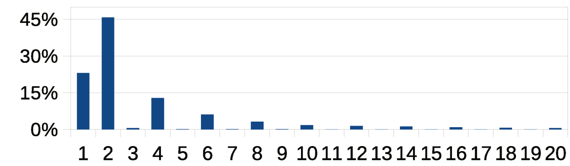
(a) Average depth of the tree nodes



(b) Maximum depth of the trees



(c) Frequency of depth distances on causality queries nodes



(d) Average depth of the tree nodes

Table 5.2: (a), (b): depths of variable/thread trees; (c), (d): frequency of depth distances on causality/conflict queries.

(c) What is the best step for branch skip lists?

This experiment aims to demonstrate that causality checking becomes trivial with the proposed data structures and providing shape information about the trees, highly valuable for implementers of unfolding-based POR techniques.

Actually, we do not compare against other causality-checking algorithms because to our best knowledge, there is not any to solve the same problem other than naively scanning all predecessors of an event.

To evaluate the depth of trees (address (a)), we ran DPU with an optimal exploration over a selection of 15 programs from Table 5.1, whose unfoldings size ranges from 380 to 204K maximal configurations. The execution time of DPU ranges from 0.2 second to 2 minutes. In total, the 15 unfoldings amount to 246 trees (150 thread and 96 variable trees) with 5.2M nodes. Table 5.2 displays the (sorted) node depth averages and maximum depths for the 246 trees.

While the average depth of a node was 22.7, as much as 80% of the trees had a maximum depth of less than 8 nodes, and 90% of them less than 16 nodes. The average of 22.7 is however larger because deeper trees contain proportionally more nodes. The depth of the deepest node of every tree was between 3 and 77. With the average size of tree is about 21K nodes, the average depth of 22.7 illustrates that the trees tend to be broad, not very deep.

We next evaluate depth differences in the causality and conflict queries over these trees to address (b). Tables 5.2c and 5.2d respectively show a histogram indicating the frequency of various depth distances associated to causality and conflict queries made by optimal POR. Surprisingly, depth differences are very small for both causality and conflict queries. When deciding causality between events, as much as 92% of the queries were for tree nodes separated by a distance between 1 and 4, and 70% had a difference of either 1 or 2 nodes. This means that optimal POR, and specifically the procedure that adds $ex(C)$ to the unfolding (which is the main issuer of causality queries), systematically performs causality queries which are trivial with the proposed data structures. When checking conflicts, the situation is similar: 82% of the queries request information about tree nodes whose depth difference is between 1 and 4. Even distances are also much more probable than odd distances.

These experiments show that all trees are not very deep and most of the queries on the causality trees require very short walks, which strongly motivates the use of the data structure proposed in Section 3.6.

Finally, we chose a (somehow arbitrary) skip step of 4 for all experiments. However, we observe that other values do not significantly impact the runtime/memory consumption for most benchmarks owing to the fact that the depth differences on causality/conflict requests are already very low.

5.5.3 Evaluation Against the State-of-the-art on System Code

We now evaluate the scalability and applicability of Dpu on five multithreaded programs in two Debian packages: *blktrace* [26], a block layer I/O tracing mechanism, and *mafft* [72], a tool for multiple alignment of amino acid or nucleotide sequences. The code size of these utilities ranges from 2K to 40K LOC, and *mafft* is parametric in the number of threads.

We compared Dpu against Maple [106], a state-of-art testing tool for multithreaded programs, as the top ranked verification tools from SV-COMP'17 are still unable to cope with such large and complex multithreaded code. Unfortunately we could not compare against Nidhugg because it cannot deal with the (abundant) C-library calls in these programs.

Table 5.3 presents our experimental results. We use Dpu with optimal exploration and the modified version of Maple used in [95]. To test the effectiveness of both approaches in state space coverage and bug finding, we introduce bugs (some assertion violations) in 4 of the benchmarks (Add, Dnd, Mdl, pla). For the buggy

Benchmark			Dpu			Maple		
Name	LOC	Th	Time	Ex	R	Time	Ex	R
Add(2)	40K	3	24.3	2	U	2.7	2	S
Add(4)	40K	5	25.5	24	U	34.5	24	U
Add(6)	40K	7	48.1	720	U	TO	316	U
Add(8)	40K	9	TO	14K	U	TO	329	U
Add(10)	40K	11	TO	14K	U	TO	295	U
Blk(5)	2K	2	0.9	1	S	4.6	1	S
Blk(15)	2K	2	0.9	5	S	23.3	5	S
Blk(18)	2K	2	1.0	180	S	TO	105	S
Blk(20)	2K	2	1.5	1147	S	TO	106	S
Blk(22)	2K	2	2.6	5424	S	TO	108	S
Blk(24)	2K	2	10.0	20K	S	TO	105	S
Dnd(2,4)	16K	3	11.1	80	U	122	80	U
Dnd(4,2)	16K	5	11.8	96	S	151	96	S
Dnd(4,4)	16K	5	TO	13K	U	TO	360	U
Dnd(6,2)	16K	7	149.3	4320	S	TO	388	S
Mdl(1,4)	38K	7	26.1	1	U	1.4	1	U
Mdl(2,2)	38K	5	29.2	9	U	13.3	9	U
Mdl(2,3)	38K	5	46.2	576	U	TO	304	U
Mdl(3,2)	38K	7	31.1	256	U	402	256	U
Mdl(4,3)	38K	9	TO	14K	U	TO	329	U
Pla(1,5)	41K	2	22.8	1	U	1.7	1	U
Pla(2,4)	41K	3	37.2	80	U	142.4	80	U
Pla(4,3)	41K	5	160.5	1368	U	TO	266	U
Pla(6,3)	41K	7	TO	4580	U	TO	269	U

Table 5.3: Comparing DPU with Maple (same machine). LOC: lines of code; Execs: nr. of executions; R: safe or unsafe. Other columns as before. Timeout: 8 min.

benchmarks, we use the random scheduler of MAPLE, considered to be the best baseline for bug finding [95]. First, we run DPU to retrieve a bound on the number of random executions to answer whether both tools are able to find the bug within the same number of executions. Both DPU and MAPLE found bugs in all buggy programs (except for one variant in ADD) even though DPU greatly outperforms and is able to achieve much more state space coverage.

For the safe benchmark BLK, we perform exhaustive state-space exploration using MAPLE's DFS mode. On this benchmark, DPU outperforms MAPLE by several orders of magnitude: DPU explores up to 20K executions covering the entire state space in 10s, while MAPLE only explores up to 108 executions in 8 min.

5.5.4 Profiling a Stateless POR

In an effort to understand the cost of each component of QPOR, we profiled the sequential version of DPU on a selection of 7 programs from Table 5.1.

DPU discovers new maximal configurations executing the program until completion. DPU spends between 30% and 90% of the runtime on this task (65% in average), depending on the computation workload of the program. The remaining runtime is spent computing alternatives, distributed as follows: adding events to the event structure (15% to 30%), building the spikes of a new comb (1% to 50%), searching for solutions in the comb (less than 5%), and computing conflicting extensions (less than 5%).

Counterintuitively, building the comb is more expensive than exploring it, even in the optimal case. One of the reasons for this is the fact that filling the spikes with events is a more memory-intensive operation than exploring the comb, which exploits locality of data.

Our profiling results show that running the program is often more expensive than computing alternatives. We parallelized this phase as a part of our parallel version described in Chapter 4 and its implementation in Section 5.4.2. Moreover, efforts in reducing the number of redundant executions, even if significantly costly, are likely to reduce the overall execution time.

5.6 Conclusions

This chapter presented in details both sequential and parallel implementations of our algorithms. The parallelism requires some additional features from OpenMP, so we briefly pointed out the advantages that lead us to choose it for parallelizing the

sequential implementation in a task-based parallel model. In the second part of the chapter, we have shown several analyses on selected experiments conducted on various benchmarks to evaluate our tool and to compare it with state-of-art verification and testing tools such as Nidhugg and Maple. As a result, DPU has exhibited better performance than the other tools we compare it with.

Chapter 6

Conclusions and Perspectives

6.1 Conclusions

This thesis presents our research on model checking techniques including partial order reduction and unfolding in Chapter 1. By revealing their strengths and drawbacks, we combined Dynamic Partial Order Reduction and unfolding techniques into verification for concurrent programs.

As contributions, in Chapter 3 we first proved that computing alternatives in optimal exploration of a Petri net or a program is a NP-complete problem (Theorems 4 and 5). To the best of our knowledge this is the first formal complexity result for an important problem that optimal and non-optimal DPORs need to solve. This finding raises the demand of an algorithm that can compute alternatives in polynomial time.

Second, we proposed a trade-off solution between exploring redundant schedules and solving the NP-complete problem of computing alternatives called QPOR presented as the main content of Chapter 3. QPOR provides a new concept k-partial alternative (k is a user-defined constant) in Definition 12 that finds a *clue* to an alternative execution in polynomial time. It is an approximate algorithm as a clue does not guarantee an actual alternative. Actually, experiments show that optimal exploration is achieved with a low value for k. We also propose a new concept of *comb* (Definition 13) that facilitates the computation of alternatives.

Third, we presented in Section 3.5 an efficient structure for computing conflict and causality between events. We organize events in trees for their thread and mutex. Checking conflict and causality between events turns into checking structural relations between nodes in a tree. This can be done by simply scanning the tree node by node but we propose to use a skip list based on a skip step to quickly navigate through these trees. Experiments show that using this structure, the trees are not very deep and

the distance to walk between two nodes are short, so computing structural relations between nodes becomes simpler.

Four, we implemented all proposed algorithms in a new tool DPU that is a dynamic analysis unfolder of C multithreaded programs. A specialized data structure is applied to obtain an easy-to-retrieve unfolding. The tool is now available to the public on github [1].

To speed up a DPOR algorithm, we can either reduce the state space or increase the amount of data processed at an unit of time. The former is addressed by QPOR and the latter can be achieved by parallelism.

Our fifth contribution is the parallelization detailed in Chapter 4. We designed a parallel algorithm (Algorithm 5) that partitions the exploration into concurrent subworks, each of which produces a maximal configuration of the unfolding and finds alternatives to that. A parallel program implements the algorithm using OpenMP, an API for shared memory parallel programming is detailed in Section 5.4.

Finally, we conducted experiments on selected benchmarks. We first used realistic C multithreaded programs to compare QPOR (implemented in DPU) against SDPOR [2] (implemented in Nidhugg). The results show that our tool outperforms Nidhugg on most of the benchmarks in terms of execution time and the number of redundant exploration avoided. The second experiment is on system code (packages from Debian) to compare our tool and Maple [106], a state-of-art testing tool, on the capability of bug detection. The results have shown that our tool DPU is an efficient state space exploring model checker as it explores much more executions in much less time than Maple. The third experiment evaluated the sequential tree proposed to facilitate the causality and conflict computation. It reveals that the data structure is efficient as most of the queries on the sequential trees require very short walks.

6.2 Perspectives

This section discusses some future works in the context of automated and distributed verification.

Improvement in avoiding redundant exploration In Section 4.3.6, to avoid redundant exploration, we use a naive algorithm to verify the existence of a replay by comparing it with all existing replays. Perspectively, we would like to store more

[1] github.com/cesaro/dpu

information from the parent subwork in a child subwork so that the child may avoid to explore the path its parents have been visited.

Parallelization improvements We have parallelized the algorithm using shared memory model on multi-core computers but the speedup is not significant. The reason is that all subworks refer to events in the shared unfolding frequently. Low level synchronization using locks forces a lot of computation to be executed in sequence as well. Even though we have tried to use locks for processes instead of the whole unfolding, the speedup is not as much as expected. We are about to design subworks in another way such that each subwork maintains its own partial unfolding which is a part of unfolding necessary for its work. All operations of a subwork are executed on its partial unfolding. The shared unfolding is accessed by a task only when it takes the subwork (copy related events to partial unfolding) and when it finishes (update new information from partial unfolding to the whole unfolding). Like that, subworks are more independent and the shared unfolding has less accesses. We believe that this design will improve the performance of the parallel implementation.

Using another parallel programming model The shared memory model applied in our parallel tool has shown some drawbacks where the shared unfolding is too big and accessed too frequently, so it limits the parallel execution. There are other models available for parallelism such as message passing, data parallel. In the future, we intend to exploit message passing programming model to have another implementation of our algorithm, so that we hopefully achieve better performance.

Distributed QPOR Many researches have combined distributed methods with partial order reduction by distributing the state space among several nodes (workstation or computers) in a distributed computer system [28, 24]. We would like to design a distributed memory algorithm and implement it to take advantage of multi-node computer systems to increase the computational power.

Extend the range of input programs Up to now, our algorithm QPOR and tool DPU assume input programs to be terminating to achieve a finite unfolding, so that the exploration terminates. Non-terminating programs such as server programs are also interesting to verify but they require more work on infinite executions. Researches have been conducted to test and verify this class of programs such as [56, 17, 41, 14] but non-terminating executions are not solved efficiently with existing DPOR. In

the future, we also intend to adapt our algorithm to handle with non-terminating programs. We need to identify and pre-process non-terminating executions to obtain a finite unfolding.

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 9035 in LNCS, pages 353–367. Springer, 2015.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, ACM, 2014.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *International Conference on Computer Aided Verification*, pages 134–156. Springer, 2016.

[4] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. Unfoldings of Unbounded Petri Nets. In *Computer Aided Verification*, pages 495–507, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[5] Hussain Al-Asaad and John P. Hayes. Design Verification via Simulation and Automatic Test Pattern Generation. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '95, pages 174–180, Washington, DC, USA, 1995. IEEE Computer Society.

[6] Albert-Jan N. Yzelman. Introduction to the Bulk Synchronous Parallel mode. http://albert-jan.yzelman.net/education/parco14/A2.pdf.

[7] Alfons Laarman and Anton Wijs. Partial-Order Reduction for Multi-core LTL Model Checking. In *HVC 2014. LNCS, vol. 8855 Springer, Heidelberg*, 2014.

[8] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157. Springer, 2013.

[9] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[10] Étienne André. IMITATOR: A Tool for Synthesizing Constraints on Timing Bounds of Timed Automata. In *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, pages 336–342, 2009.

[11] Étienne André, Camille Coti, and Sami Evangelista. Distributed behavioral cartography of timed automata. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 109. ACM, 2014.

[12] Étienne André, Camille Coti, and Hoang Gia Nguyen. Enhanced distributed behavioral cartography of parametric timed automata. In *International Conference on Formal Engineering Methods*, pages 319–335. Springer, 2015.

[13] Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A Trustworthy Proof Checker. *Journal of Automated Reasoning*, 31(3):231–260, Nov 2003.

[14] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting Fair Non-termination in Multithreaded Programs. In *Computer Aided Verification*, pages 210–226, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[16] C. Baier, M. Grosser, and F. Ciesinski. Partial order reduction for probabilistic systems. In *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pages 230–239, 2004.

[17] Alexey Bakhirkin. *Recurrent sets for non-termination and safety of programs.* PhD thesis, University of Leicester, UK, 2016.

[18] Paolo Baldan, Stefan Haar, and Barbara König. Distributed unfolding of Petri nets. In *Proc. FoSSaCS*, volume 3921 of *LNCS*, pages 126–141. Springer, March 2006.

[19] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification*, pages 863–868, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] Jiří Barnat, Lubos Brim, and Petr Rockai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *International Symposium on Automated Technology for Verification and Analysis*, pages 234–239, 2008.

[21] Jiří Barnat, Lubos Brim, and Jitka Stříbrná. Distributed LTL model-checking in SPIN. In *International SPIN Workshop on Model Checking of Software SPIN'01*, pages 200–216, 2001.

[22] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. Divine–a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281, 2006.

[23] Bernhard Beckert and Michał Moskal. Deductive Verification of System Software in the Verisoft XT Project. *KI - Künstliche Intelligenz*, 24(1):57–61, Apr 2010.

[24] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Formal Methods in Computer-Aided Design*, pages 427–441, Berlin, Heidelberg, 2000.

[25] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[26] blktrace. http://brick.kernel.dk/snaps/.

[27] Dragan Bošnački, Stefan Leue, and Alberto Lluch Lafuente. Partial-Order Reduction for General State Exploring Algorithms. In *Model Checking Software*, pages 271–287, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] L. Brim, I. Cerna, P. Moravec, and J. Simsa. Distributed Partial Order Reduction of State Spaces. *Electronic Notes in Theoretical Computer Science*, 128(3):63 – 74, 2005. Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004).

[29] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed Partial Order Reduction of State Spaces. *Electron. Notes Theor. Comput. Sci.*, 128(3):63–74, April 2005.

[30] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *ACM Trans. Program. Lang. Syst.*, 35(8):677–691, 1986.

[31] Randal E. Bryant. *Binary Decision Diagrams*, pages 191–217. Springer International Publishing, Cham, 2018.

[32] Sagar Chaki and Arie Gurfinkel. *BDD-Based Symbolic Model Checking*, pages 219–245. Springer International Publishing, Cham, 2018.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[34] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[35] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, Berlin, Heidelberg, 2001. Springer-Verlag.

[36] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

[37] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[38] Camille Coti, Sami Evangelista, and Laure Petrucci. One-Sided Communications for More Efficient Parallel State Space Exploration over RDMA Clusters. In *European Conference on Parallel Processing*, pages 432–446. Springer, 2018.

[39] Jörg Desel, Gabriel Juhás, and Christian Neumair. Finite Unfoldings of Unbounded Petri Nets. In *Applications and Theory of Petri Nets 2004*, pages 157–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[40] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, May 2005.

[41] Emmi, Michael and Lal, Akash. Finding Non-terminating Executions in Distributed Asynchronous Programs. In *Static Analysis*, pages 439–455, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] Javier Esparza and Keijo Heljanko. A New Unfolding Approach to LTL Model Checking. In *Automata, Languages and Programming*, pages 475–486, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[43] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In *Proc. SPIN*, volume 2057 of *LNCS*, pages 37–56, 2001.

[44] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[45] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

[46] Sami Evangelista. High Level Petri Nets Analysis with Helena. In *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, pages 455–464, 2005.

[47] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol. Improved Multi-Core Nested Depth-First Search. In *International Symposium on Automated Technology for Verification and Analysis ATVA*, pages 269–283, 2012.

[48] Sami Evangelista, Laure Petrucci, and Samir Youcef. Parallel nested depth-first searches for LTL model checking. In *International Symposium on Automated Technology for Verification and Analysis*, pages 381–396. Springer, 2011.

[49] Eric Fabre, Albert Benveniste, Stefan Haar, and Claude Jard. Distributed monitoring of concurrent and asynchronous systems*. *Discrete Event Dynamic Systems*, 15(1):33–84, Mar 2005.

[50] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397, Aug 2011.

[51] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, pages 110–121. ACM, 2005.

[52] A. Galton. Logic As a Formal Method. *Comput. J.*, 35(5):431–440, October 1992.

[53] Y. Gao and X. Li. An effective model extraction method with state space compression for model checking SystemC TLM designs. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 64–71, 2013.

[54] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[55] Valentin Goranko and Antony Galton. Temporal Logic. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition, 2015.

[56] A. Gotlieb and M. Petit. Towards a Theory for Testing Non-terminating Programs. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 160–165, July 2009.

[57] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *Model Checking Software (SPIN)*, volume 4595 of *LNCS*, pages 95–112. Springer, 2007.

[58] Frédéric Herbreteau, Grégoire Sutre, and The Quang Tran. Unfolding Concurrent Well-Structured Transition Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 706–720, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[59] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[60] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[61] Gerard J. Holzmann. Parallelizing the Spin Model Checker. In *International SPIN Workshop on Model Checking of Software SPIN'12*, pages 155–171, 2012.

[62] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)* . Prentice Hall, 2006.

[63] Apple Inc. Grand Center Dispatch. https://developer.apple.com/documentation/dispatch.

[64] Intel. Cilk++. https://www.cilkplus.org/.

[65] Intel. Thread Building Blocks. https://www.threadingbuildingblocks.org/.

[66] Kari Kähkönen and Keijo Heljanko. Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution. In *Application of Concurrency to System Design (ACSD)*, pages 142–151. IEEE, June 2014.

[67] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

[68] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[69] Alfons Laarman, Rom Langerak, Jaco Van De Pol, Michael Weber, and Anton Wijs. Multi-core Nested Depth-first Search. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 321–335, Berlin, Heidelberg, 2011. Springer-Verlag.

[70] K. Leino and M. Rustan. Automating Theorem Proving with SMT. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 2–16, Berlin, Heidelberg, 2013. Springer-Verlag.

[71] Gavin Lowe. Concurrent Depth-First Search Algorithms. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 202–216, 2014.

[72] MAFFT. http://mafft.cbrc.jp/alignment/software/.

[73] Comparison of the communication performance on the Paris 13 computation center. http://www.univ-paris13.fr/calcul/wiki/index.php?title=Benchs:communications.

[74] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[75] Kenneth L. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. In Pierre Wolper, editor, *Proc. CAV*, volume 939 of *LNCS*, pages 180–195. Springer, 1995.

[76] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[77] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-Optimal Partial Order Reduction. In *Computer Aided Verification*, pages 354–371, Cham, 2018. Springer International Publishing.

[78] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[79] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains. In *Proc. of the International Symposium on Semantics of Concurrent Computation*, volume 70 of *LNCS*, pages 266–284. Springer, 1979.

[80] Eric Noonan, Eric Mercer, and Neha Rungta. Vector-clock based partial order reduction for jpf. In *ACM SIGSOFT Software Engineering Notes 39(1)*, pages 1–5, 2014.

[81] OpenMP Application Programming Interface 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[82] Corina S. Păsăreanu and Willem Visser. Symbolic Execution and Model Checking for Testing. In *Hardware and Software: Verification and Testing*, pages 17–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[83] Doron Peled. Partial order reduction: Model-checking using representatives. In *Mathematical Foundations of Computer Science 1996*, pages 93–112, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[84] Doron Peled. Partial Order Reduction: Linear and Branching Temporal Logics and Process Algebras. In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*, POMIV '96, pages 233–257, New York, NY, USA, 1997. AMS Press, Inc.

[85] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 17–28, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[86] Pthreads. https://computing.llnl.gov/tutorials/pthreads/.

[87] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *Proc. CONCUR*, pages 456–469, 2015.

[88] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. *CoRR*, abs/1507.00980, 2015.

[89] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *International Conference on Runtime Verificatione*, 2012.

[90] Marcelo Sousa, César Rodríguez, Vijay D'Silva, and Daniel Kroening. Abstract interpretation with unfoldings. *CoRR*, abs/1705.00595, 2017.

[91] S. Stoica. System design verification tests - an overview. In *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pages 689–697, Sept 1999.

[92] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Formal Techniques for Distributed Systems*, pages 219–234, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[93] Yann Thierry-Mieg. Symbolic Model-Checking Using ITS-Tools. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 231–237, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[94] Thomas Neele and Anton Wijs and Dragan Bosnacki and Jaco van de Pol. Partial-Order Reduction for GPU Model Checking. In *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2016.

[95] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *TOPC*, 2(4):23:1–23:37, 2016.

[96] Top 500 Supercomputers. https://www.top500.org/.

[97] Uppaal Model Checker. http://www.uppaal.org/.

[98] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515. Springer, 1991.

[99] Ruud van de Pas. An introduction into OpenMP. *International Workshop, IWOMP*, 2005.

[100] Balaji Venu. Multi-core processors - An overview. *arXiv:1110.3535 [cs.AR]*, 2011.

[101] Nikolaos S. Voros, Wolfgang Mueller, and Colin Snook. *An Introduction to Formal Methods*, pages 1–20. Springer US, Boston, MA, 2004.

[102] Anton Wijs. BFS-Based Model Checking of Linear-Time Properties with an Application on GPUs. In *CAV*, 2016.

[103] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[104] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software. In *International SPIN Workshop on Model Checking of Software*, 2007.

[105] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction. *International Journal on Software Tools for Technology Transfer*, 12(2):113–122, May 2010.

[106] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA*, pages 485–502, 2012.

Titre: Vérification de modèle quasi-optimale de systèmes concurrents

Mots clés: *Dépliage, Réduction d'ordre partiel, Quasi-optimale, Alternative k-partielle, Méthods formelles*

Résumé: En effectuant une exploration exhaustive de tous les comportements possibles du système, le model checking fait face au problème de l'explosion de cet espace d'états. Notre but est de vérifier des programmes concurrents. Nous avons proposé de combiner la DPOR et le dépliage dans un algorithme appelé POR basée sur le dépliage.

Dans cette thèse, nous prouvons que le calcul des alternatives dans une DPOR optimale est un problème NP-complet. Nous proposons une approche hybride appelée réduction d'ordre partiel quasi-optimale (QPOR). En particulier, nous proposons une nouvelle notion d'alternative k-partielle et un algorithme en temps polynomial. Une autre contribution algorithmique de cette thèse est la représentation des relations de causalité et de conflit dans le dépliage comme un ensemble d'arbres dans lequel les événements sont encodés comme un ou deux nœuds dans deux arbres différents. Nous montrons que vérifier la causalité et le conflit entre deux événements revient à une traversée efficace d'un des deux arbres.

Nous détaillons l'implémentation de l'algorithme et les structures de données utilisées dans un nouvel outil. Outre les améliorations algorithmiques garanties par QPOR, la parallélisation est un autre moyen d'accélérer l'exploration. Par conséquent, nous proposons un algorithme de QPOR parallèle. Enfin, nous présentons des expériences sur l'implémentation séquentielle de QPOR et comparons les résultats avec d'autres outils de test et de vérification afin d'évaluer l'efficacité de nos algorithmes. L'analyse des résultats montre que notre outil présente de meilleures performances que ceux-ci.

Title: Quasi Optimal model checking for concurrent systems.

Keywords: *Unfolding, Partial Order Reduction, Formal methods, Model checking, k-partial alternative*

Abstract: By exhaustively exploring all possible behaviours of the system, model checking has to face the state space explosion problem. We target the verification of concurrent programs. Dynamic partial-order reduction (DPOR), is a mature approach to mitigate the state explosion problem based on Mazurkiewicz trace theory, whereas unfolding is still at an initial state for targetting programs.

We propose to combine DPOR and unfolding into an algorithm called Unfolding based POR. In order to obtain optimality, the algorithm is forced to compute sequences of transitions known as alternatives. In this thesis, we prove that computing alternatives in optimal DPOR is an NP-complete problem. As a trade-off solution, we propose a hybrid approach called Quasi-Optimal POR (QPOR). In particular, we provide a new notion of k-partial alternative and a polynomial algorithm to compute alternative executions.

Another main algorithmic contribution is to represent causality and conflict as a set of trees where events are encoded as one or two nodes in two different trees. We show that checking causality and conflict between events amounts to an efficient traversal in one of these trees. We also implement the algorithm and data structures in a new tool. Besides the algorithmic improvements guaranteed by QPOR, parallelization is another way to speed up the unfolding exploration, thus we propose a parallel algorithm based on parallelizing QPOR. Finally, we conduct experiments on the sequential implementation of QPOR and compare the results with other state-of-art testing and verification tools to evaluate the efficiency of our algorithms. The analysis shows that our tool outperforms them.

Laboratoire d'Informatique Paris Nord
CNRS UMR 7030, Université Paris 13, France