Abstractions and Independence

Marcelo André Barbosa de Sousa
Merton College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
2018

Acknowledgements

The journey of this dissertation was much longer than I anticipated when I first visited Oxford in 2012 to meet my future DPhil advisers: Daniel Kroening and Luke Ong. Without their guidance and support, this journey wouldn't have happened and I am honored to have them as my academic parents. I would also like to thank my academic brothers in both research groups for many interesting discussions throughout the years. I must *salute Cesar* Rodriguez who spent an incredible amount of time teaching me about unfoldings and pushed me to excel in research and beyond.

During my DPhil, I was fortunate to have excellent co-authors and I would like to thank them for their impact in my academic success. I was also fortunate to have done two internships which deeply impacted my work. The first internship at Microsoft Research Cambridge was the beginning of my collaboration with Isil Dillig. She has been a great mentor and shared many insights and research ideas which really enlighten my perspective on research and academic success. I also thank her for inviting me for a research visit at UT Austin and Thomas Dillig for welcoming me to their home and share amazing barbecues. In the second internship at Google, I was fortunate to be hosted by Vijay D'Silva and Domagoj Babic. It was through conversations with Arnaud Venet that I learned invaluable insights on how to implement mature static analyzers. A special thanks to Vijay who spent many hours discussing abstract interpretation and models of concurrency while showing me the cool spots around Berkeley. Our visit to Stanford and discussions with Vaughan Pratt was one of the highlights of my DPhil.

I thank the Department of Computer Science at Oxford and Google for funding my DPhil studies and all the staff at the department (special mention to Julie Sheppard) and at Merton College who made my life at Oxford very enjoyable.

I also want to thank my examiners throughout the DPhil: Jade Alglave and James Worrell (transfer); James Worrell and Stefan Kiefer (confirmation); and specially Marta Kwiatkowska and Parosh Abdulla who were my viva examiners. Their comments, insights, remarks and suggestions greatly improved the content of this dissertation.

I thank my Portuguese friends at Oxford: Inês, João Lima, João Sousa Pinto, Mafalda and Miguel who made my time at Oxford easier. Finally, I specially thank my partner in life, Ece Akilli, for being always there for me with her smile and love. I dedicate this dissertation to my family for their unconditional love and support throughout my life.

Abstract

Efficient state space exploration of a concurrent program is a fundamental problem in algorithmic verification because of the known *state explosion problem*. In the past decades, dynamic partial order reductions and Petri net unfoldings have been two promising approaches to address the state explosion problem. However, these techniques typically focus on the combinatorial explosion caused by the potential linearizations of partial orders and do not generally address multiple sources of explosion.

In this dissertation we present a multi-faceted approach to the state explosion problem in automated safety verification of concurrent programs. Our approach is based on two main ideas.

The first idea is that there is a formal connection between the notion of *independence of concurrent actions* and the fundamental notion of *event* as an atomic unit in the representation of the state space of a concurrent program. Using this idea we developed state-of-the-art exploration algorithms based on event structures which can achieve *super-optimal* explorations of the state space.

The second idea is to combine event structures with data abstractions to further mitigate the state explosion. We show that in this setting, the independence relation exploited in previous approaches is a particular instance of a larger class of independence relations. Our main contribution is an unfolding algorithm that uses a new notion of independence to avoid redundant transformer application, thread-local fixed points to reduce the size of the unfolding, and a novel cutoff criterion based on subsumption to guarantee termination of the analysis. Our experiments show that the abstract unfolding produces an order of magnitude fewer false alarms than a mature abstract interpreter, while being several orders of magnitude faster than solver-based tools that have the same precision.

Contents

List of Figures

List of Abbreviations

AI	Abstract Interpretation
BFS	Breath First Search
DFS	Depth First Search
DPOR	Dynamic Partial Order Reduction
ES	Event Structure
LPES	Labelled Prime Event Structure
LTS	Labelled Transition System
M-trace	Mazurkiewicz Trace
ODPOR	Optimal Dynamic Partial Order Reduction
PER	Partial Equivalence Relation
PES	Prime Event Structure
POR	Partial Order Reduction
poset	Partially Ordered Set
SDPOR	Source Dynamic Partial Order Reduction
SMC	Stateless Model Checking
SSB	Sleep Set Blocked Execution

Chapter 1

Introduction

Concurrency is a property of an algorithm in which its constituent statements are specified or executed as a partial order. Since the 1960s, this partial order perspective influenced a large number of semantics and verification techniques for concurrent programs. Currently, programmers implement concurrent algorithms in modern programming languages as a partial order of sequential programs known as processes, threads, actors, etc. Also, with the advent of asynchronous programming and weak memory models, we can even observe concurrent behavior in sequential programs caused by the low-level execution layers.

Unfortunately, it is common for a process in a concurrent program to interfere with the local state of other running processes. As a consequence, different linear executions of the partial order representation can result in distinct final states. This leads to a combinatorial explosion of possible behaviors, a phenomenon known as the *state explosion problem*. In this scenario, exhaustive state exploration is not practical to identify whether a state of the program is reachable.

In the past thirty years, the state explosion problem has been extensively addressed in a panoply of techniques forming an overwhelming compendium of publications. However, most techniques typically focus on the combinatorial explosion caused by the potential linearizations of partial orders. In the context of programs, some of these techniques are not applicable since they were designed for other models or because the state explosion is further exacerbated by the use of data types and the increasing size of the programs.

In this dissertation, we present a multi-faceted approach to the state explosion problem in automated safety verification of concurrent programs. We focus on multi-threaded programs that communicate via shared memory as they form a large domain of concurrent programs of practical relevance.

Our approach is based on two main ideas. The first idea is that there is a formal connection between the notion of *independence of concurrent actions* and the notion of *event* as an atomic unit in the representation of the state space of a concurrent program. The second idea is that once we consider data abstractions to further mitigate the state explosion, the independence relation exploited in previous approaches is a particular instance of a larger class of independence relations. These two ideas suggest that a promising direction (investigated in this dissertation) is to combine *abstractions and independence* to mitigate multiple sources of explosion.

1.1 Motivation

State space exploration is a widely used technique for algorithmic verification of state reachability problems in finite-state multi-threaded programs. The basic idea of this approach is to explore the state graph of the program and check whether the state of interest is observed. Besides its simplicity and ease of automation in a verification tool, state space methods are useful in stages across the development cycle such as design, debugging and testing.

A major obstacle to practical state space exploration methods is the well known *state explosion problem* [Val98]. Because the state space of a multi-threaded program grows exponentially with the number of threads, a complete state space exploration is infeasible. In techniques that reason about executions such as stateless model checking (SMC) [God97] this problem is even more severe. For example, a program with n threads, each composed of a single deterministic action, has 2^n control states and $n!$ executions.

The main idea to overcome the state explosion problem has been to consider a *relevant* portion of the state space. Typically this has been accomplished using more compact representations or approximations via abstraction [MR87]. As Valmari writes in [Val98]: *without abstraction there is very little or no room for obtaining reduction.* This key idea of *avoiding state explosion via reductions by abstraction* is also the main *motto* throughout the work presented in this dissertation.

We focus on two exploration-based approaches that are considered to be the best candidates to overcome the state explosion problem: dynamic partial order reductions (DPORs) [FG05, GFYS07, YWY06, YCGK08, AAJS14, AAA+15b] and Petri net unfoldings [EH08, McM93a]. Intuitively, these approaches compactly represent the state space by viewing *concurrent executions as partial orders*. For example, in the previous simple program with n threads, consider that for $n = 4$, all 24 executions

reach the same final state. Such situation occurs when two actions enabled at a state do not interfere which each other, a property between actions of different threads known as *independence*. In that case, it is more efficient to represent the state space by a single partial order with 4 unordered elements than the state graph that will necessarily contain at least 16 states corresponding to the vertices of a 4-cube. Thus, a key idea underlying both techniques is to use independence to obtain a partial order representation that is potentially more compact than the state graph. By construction, each partial order represents an equivalence class of executions with respect to their final state. To verify this property it suffices to explore one representative per equivalence class.

In this dissertation, we focus on identifying the set of final states of the program, also known as the set of *deadlock* states. This set is particularly interesting since we can reduce the verification of a safety property to deadlock reachability [GW92].

Note that to mitigate state explosion we are not forced to use partial orders: there are *true concurrency* semantics (e.g. *stable event structures* [Win88], *higher dimensional automata* [Pra91], *chu spaces* [Gup94]) which cannot be represented using partial orders. Also, there are exploration algorithms such as the family of POR algorithms based on stubborn sets [Val91, VH17] and recent methods [CCP+17] which do not rely on partial orders. The fundamental intuition behind these exploration methods is to obtain a *reduction of the state space via equivalence classes*. Thus, the core algorithmic problem both DPORs and net unfolding explorations tackle is how to efficiently explore these equivalence classes.

DPORs have been successfully applied in SMC of multi-threaded programs to detect erroneous behavior such as deadlocks and data races. The main strategy in SMC is a DFS-style exploration of a reduced computation tree where executions of the program are representatives of the equivalence classes, known as Mazurkiewicz traces (M-traces). The *actual* reduction of the state space consists in identifying for each state, a sound subset of enabled transitions to explore. This is typically done by combining forward reasoning (e.g. using *persistent*, *source* sets) and backwards reasoning (e.g. using *sleep* sets). In SMC, termination is usually assumed by restricting the input programs to those with finite acyclic state spaces.

On the other hand, unfolding explorations have been almost exclusively applied to 1-safe Petri nets [McM93a]. It is well known that the unfolding of a 1-safe Petri net is a prime event structure (PES) [NPW81], a particular event-based model at the bottom of a rich hierarchy of models of concurrency [vGP09]. Current unfolding explorations typically build the PES using a saturation procedure that extends the

current prefix with a single event. The exploration does not resort to linearizations or the interleaving semantics as it directly leverages the partial order representation to explore its constituent elements [1]. Since the state space of the net typically contains cycles, its PES is infinite and require stateful explorations combined with the theory of cutoffs [ERV02] to obtain finite complete prefixes.

DPORs and net unfoldings also tackle different properties. In particular, DPORs aim to identify the set of deadlock states by exploring the set of M-traces while unfoldings aim to explicitly represent the set of local states of the system which are encoded in the set of events. It is not surprising then that: 1) it is possible to represent an exponential number of M-traces with a polynomial-size PES and 2) given a PES, checking whether a state is a deadlock is a NP-complete problem [McM93a]. This suggests that a prime event structure is a suitable intermediate data-structure to combine algorithmic insights from both explorations in an effort to develop new algorithms that can further exploit the succinctness of a partial order representation.

However, in the context of multi-threaded programs, we need to define what is the event structure representation for the state space and also overcome other sources of state explosion to achieve scalable explorations. Similarly to the computation tree explored in SMC, a prime event structure does not scale to other sources of non-determinism that arise in the presence of data. Since we are still in the domain of *computation trees*, the structure is fundamentally based on executions and still suffers from the known *path explosion problem*.

In the sequential case, the path explosion problem can be circumvented using an approach that reasons about states as opposed to executions. In this context, two approaches have been extensively studied in the literature: 1) compositional methods such as proof systems and 2) fixpoint approximation using solvers or abstract interpreters. A standard proof system typically applies structural induction over the programming language constructs. One of the desired properties of a proof system is compositionality, i.e. the ability to soundly compose reasoning of disjoint parts of the code. This approach has two limitations in our context: 1) it is hard to automate since it requires an induction hypothesis and 2) the reasoning on a general multi-threaded program is hardly ever compositional in the sense that there is behavior in the program which cannot be observed by only reasoning over the threads in isolation. To the best of our knowledge, the best known compositional method for

[1] In this sense, standard unfolding explorations follow the *true semantics* philosophy and seem to be a promising approach to design parallel verification algorithms for concurrent programs.

concurrency [Jon83] has not been automatically applied to industrial multi-threaded programs.

In the second approach, one can compute the set of reachable states in a program by starting with the initial set of states, and at each step compute the next set of states reachable from the current set until fixpoint. Since the state of a program is composed of control and data, the procedure usually uses the control flow graph of the program to guide the exploration. A state-based exploration can overcome path explosion in multiple ways: 1) it can execute multiple executions simultaneously, 2) it can use compact representations of states or data abstractions to increase the scalability of the analysis or handle infinite state-spaces and 3) it can use the control flow graph to *merge* states from multiple executions. The major problem in applying this approach to multi-threaded programs is the lack of a compact control flow graph representation of the entire program. Using the product control flow graph of the components immediately leads to an exponential representation which we want to avoid. Existing approaches address this problem by performing thread-modular static analysis [Min14, YB12]. However, it is common for such proofs to require path-sensitive interference conditions. Current abstract interpreters for multi-threaded programs either do not scale when computing such constraints or have a high false positive rate.

The main motivation for this dissertation is to investigate the potential of a state exploration based on abstract interpretation over event structures as a multi-faceted approach to the state explosion problem. Such approach should lift current limitations of SMC-based DPORs and thread-modular abstract interpreters.

1.2 Research Hypothesis

The main research hypothesis of this dissertation is that a DPOR based on prime event structures can be combined with a stateful exploration, data abstractions and thread-local fixed points leading to exponential reductions of the state space explored by optimal DPORs and orders of magnitude fewer false positives than mature abstract interpreters. To this end, we present two auxiliary hypotheses.

1. A stateful DPOR exploration based on a prime event structure equivalent (via *independence*) to the set of Mazurkiewicz traces can handle programs with non-acyclic state spaces and obtain an exponential reduction of the state space explored by optimal DPORs.

2. A DPOR exploration can be combined with data abstractions and thread-local fixed points extending the current use of DPORs to multi-threaded programs with data while outperforming state-of-the-art symbolic DPORs and reducing the false positive rate of mature abstract interpreters by orders of magnitude.

1.2.1 Research Questions

In this section, we present the main research questions connected to our hypothesis and argue about the overall novelty of the approach.

The first step in the design of our analyzer is to define a semantics of multi-threaded programs for efficient algorithmic verification. In particular, our main interest from a semantic perspective is to define a suitable data-structure for the novel explorations hypothesized above.

In this dissertation, we focus on event-based models as they are the basis of the two algorithmic approaches of interest: DPORs which are based on the theory of M-traces, a linear-time model, and Petri net unfoldings which are based on PES, a branching-time model. Both semantics are partial order semantics but have been typically defined for different domains. DPORs reduce the state space based on the independence of program statements by using it to represent the interleaving space as a set of partial orders. Prime event structures can be seen as a compact representation of a set of partial orders. In the context of programs it is not clear how to directly derive a prime event structures semantics parameterized by the independence relation as they have been mainly associated with semantics of Petri nets.

Research Question 1. In the context of multi-threaded programs, can we define a family of prime event structures equivalent to the family of M-traces explored by DPORs and relate it with the set of independence relations?

This multipartite question establishes the semantic foundation that motivates the first part of our research hypothesis.

Algorithmically, there is an interaction between two sources of reduction: 1) the state reduction achieved by the compactness of the representation and 2) the transition reduction achieved by the exploration. The key semantic notion that drives both reductions is **independence**. Without independence, these exploration methods simply explore the computation tree of the program. Therefore, we refer to both DPORs and unfolding explorations as **independence-based explorations**. DPORs

explore the equivalence classes defined by independence while trying to minimize the number of times they explore each class. An exploration is sound when it visits at least one execution per M-trace and it is optimal when it always visits at most one execution per M-trace. This leads to our second research question.

Research Question 2. Can we define an optimal DPOR over prime event structures and use cutoffs to achieve further reduction based on states?

As we mentioned, prime event structures are inapt to overcome *path explosion* caused by other sources of non-determinism as it is based on executions. On the other hand, state-based explorations performed by static analyzers typically do not suffer from this limitation. Abstract interpretation is a natural theoretical framework to use in the design of an analyzer that overcomes the multiple sources of explosion mentioned. Since event structures already provide the fundamental data-structure (the analog of the control flow graph for a sequential program), the natural question is how to use or implement an abstract interpreter on top of an event structure.

Research Question 3. How to combine independence-based explorations with data abstractions?

The main obstacle in this last research question is to investigate the interplay between independence and abstraction in the combination of the strengths of both static and dynamic analysis.

Finally, we consider that representations of multithreaded programs fundamentally rely on *events* as an atomic unit of behavior. In current algorithmic approaches this notion is usually unclear which thwarts the design and implementation of more advanced reduction strategies as they are not typically connected to a mature model of concurrency such as Mazurkiewicz trace theory or event structures. We believe that it is possible to obtain a systematic design of program analyzers that employ more aggressive reductions based on event-based models. The first step towards such goal is obtain a general event model for semantics of multi-threaded programs using a general definition of **event**.

Research Question 4. Can we obtain a general notion of event from the interleaving semantics to derive semantics for more aggressive reductions?

1.3 Contributions

In this section, we present the main contributions of this dissertation and the connection with the research questions presented in Section 1.2.

The main contributions of this dissertation are new algorithms for more scalable and precise analysis of multithreaded programs. In particular, we advance the state-of-the-art DPORs with a novel exploration algorithm over a representation of the interleaving space of a program inspired by *Petri net unfoldings*. The unfolding of a program is an element of a family of prime event structures where each unfolding is parameterized by an independence relation and corresponds to a particular set of M-traces that represents the interleaving space. The definition of this set of unfoldings directly from the interleaving space of a program and the set of independence relations is a semantic contribution of this dissertation. The algorithm presented is novel because it is able to combine key insights from optimal DPORs and net unfolding constructions to identify which events can be discarded during exploration while keeping useful state information. This combination lifts current memory limitations in unfolding constructions and limitations of DPORs regarding stateful explorations and non-acyclic state-spaces. Compared to state-of-the-art DPORs, this algorithm is able to achieve super-optimal explorations, i.e., sound explorations that explore fewer executions than M-traces, which is experimentally observed by dramatic performance improvements. The second main contribution of this dissertation is a new unfolding exploration that combines insights from dynamic and static analysis of multithreaded programs. The main novelties of this algorithm are: 1) a new notion of independence to avoid redundant transformer application, 2) thread-local fixed points to reduce the size of the unfolding, and 3) a novel cutoff criterion based on subsumption to guarantee termination of the analysis. Our experiments show that this new technique using abstract unfoldings produces an order of magnitude fewer false alarms while being several orders of magnitude faster than state-of-the-art techniques.

Research Question 1. (Chapter 3) We present a new definition of a family of prime event structure semantics where the conflict relation is statically determined by a symmetric relation over actions. We show how to use this parametric semantics to represent the interleaving space of a multithreaded program using binary independence relations. Finally, we show that this representation is equivalent to the set of Mazurkiewicz traces generated by the same independence relation. The results in this chapter have been partially published in [RSSK15a].

Research Question 2. (Chapter 4) We investigate the relationship between DPOR algorithms as instances of a general independence-based exploration algorithm over prime event structures. The main result is a template algorithm that combines the advantages of ODPOR and unfolding exploration algorithms. We study the properties of this algorithm and describe its implementation together with optimizations in a tool called Poet. Finally, we show via experiments that Poet can achieve super-optimal explorations and outperform state-of-the art DPOR explorations. The results in this chapter have been partially published in [RSSK15a] and [HRS+18].

Research Question 3. (Chapter 5) We present a new technique that uses abstract interpretation with prime event structures baptized *causality-based abstract interpretation*. The main contribution of this chapter is an unfolding algorithm that uses a new notion of independence to avoid redundant transformer application, where prime event structures are used to compactly represent causal dependence and interference between sequences of transformers. Moreover, we present an optimization of this algorithm that combines thread-local fixed points with a global fixed point over the unfolding. We describe an implementation of the algorithm in an extension of Poet called APoet. We show that causality-based abstract interpretation produces an order of magnitude fewer false alarms than a mature abstract interpreter, while being several orders of magnitude faster than solver-based tools that have the same precision. The results in this chapter have been partially published in [SRDK17].

Research Question 4. (Chapter 6) We apply abstract interpretation to study event based models of concurrency using labelled transition systems as the concrete semantics. Our main contribution is a general formalization of events as Boolean abstractions of transition sequences. Furthermore, we characterize the event abstractions whose event sequences can be represented with prime event structures. The results in this chapter have been partially published in [DKS17].

Publications and Main Contributions

In this section, I present my main contributions in each publication mentioned above as a contribution to this dissertation:

- [HRS+18] Huyen T. T. Nguyen, Cesar Rodriguez, Marcelo Sousa, Camille Coti, Laure Petrucci. *Quasi-Optimal Partial Order Reduction*. CAV 2018:

  - Technical discussions, formalization and proof reading in all sections except Section 5.1.
  - Benchmarks with O(n) M-traces where SDPOR explores O(2^n) executions.
  - Experimental evaluation against MAPLE.

- [SRDK17] Marcelo Sousa, Cesar Rodriguez, Vijay D'Silva, Daniel Kroening. *Abstract Interpretation with Unfoldings*. CAV 2017.
  - Technical discussions, formalization and proof reading in all sections.
  - Main author of technical contributions in Sections 4, 5 and algorithm.
  - Implementation and experimental evaluation.

- [DKS17] Vijay D'Silva, Marcelo Sousa, Daniel Kroening. *Independence Abstractions and Models of Concurrency*. VMCAI 2017.
  - Technical discussions and formalization in all sections.
  - Main author of technical contributions in Sections 4 and 5.

- [RSSK15a]: Cesar Rodriguez, Marcelo Sousa, Subodh Sharma, Daniel Kroening. *Unfolding-based Partial Order Reduction (Best Paper Award)*. CONCUR 2015.
  - Technical discussions, formalization and proof reading in all sections.
  - Main author of technical contributions in Sections 3 and 6.
  - Co-author of the exploration algorithm.
  - Implementation and experimental evaluation.

Additionally, I list the remaining publications obtained while pursuing the DPhil degree which are not included as contributions in the dissertation:

- [SDL18] Marcelo Sousa, Isil Dillig, Shuvendu Lahiri. *Verified Three-way Program Merge*. OOPSLA 2018.
- [DS17] Vijay D'Silva, Marcelo Sousa. *Complete Abstractions and Subclassical Modal Logics*. VMCAI 2017.
- [SD16] Marcelo Sousa, Isil Dillig. *Cartesian Hoare Logic for Verifying k-Safety Properties*. PLDI 2016.

- [SDV+14] Marcelo Sousa, Isil Dillig, Dimitrios Vytionitis, Thomas Dillig, Christos Gkantsidis. *Consolidation of Queries with User-Defined Functions*. PLDI 2014.

- [SS13] Marcelo Sousa, Alper Sen. *LLVMVF: A Generic Approach for Verification of Multicore Software*. JETTA 2013.

1.4 Overview

We now provide an overview of the dissertation.

In Chapter 2, we present mathematical background used throughout this dissertation. Furthermore, we formally present a simple programming language for concurrent programs and define several models of concurrency that will be studied.

The diagram in Figure 1.1 displays the main concepts and transformations presented after Chapter 2.

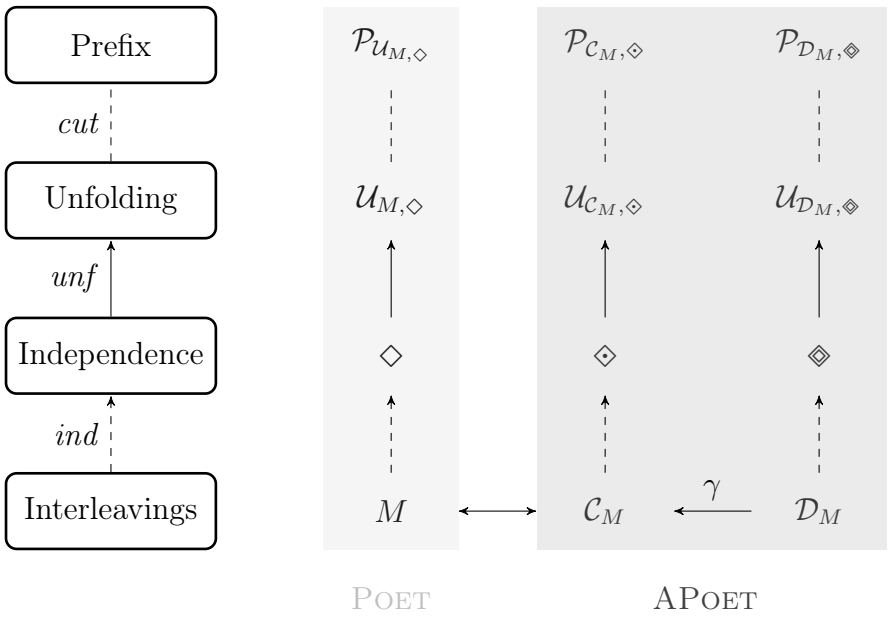


Figure 1.1: Overview diagram.

The first column shows the conceptual flow of Chapters 3 to 5. We consider as a starting point the interleaving semantics defined from the labelled transition system of a program. The first step in the application of our framework is to derive a suitable definition of independence that will drive the state space reduction. Once we obtain a method to compute independence relations, we use it to build a prime event structure

semantics from labelled transition systems referred to as the unfolding. Similarly to computation trees, if the program has infinite executions, the unfolding will be an infinite object. We use the theory of cutoffs to define suitable prefixes useful for practical verification.

The second column describes an instance of this framework applied to deterministic concurrent programs which is studied in detail in Chapters 3 and 4 and implemented in POET.

The third and fourth columns represent instances of the framework for non-deterministic concurrent programs and abstractions using Galois connections. We remark that the second and third column are the same if the program M is deterministic. We study new types of independence that arise in both cases: we denote by $\diamond$ the independence for the non-deterministic case and by $\diamondsuit$ the abstract independence relation. The formal connections between independence relations, the unfoldings and potential suitable prefixes (which are not represented with arrows in this diagram) are formalized in Chapter 5 and implemented in APOET.

In Chapter 6, we define events and event sequences as abstractions over the interleaving semantics and use this formalization to characterize what notions of events accept a prime event structure representation.

Finally, in Chapter 7 we analyze the main results presented and identify research directions for future work.

Chapter 2

Preliminaries

In this chapter, we present the mathematical background, the various models of concurrency and the model of computation for multithreaded programs used in the subsequent chapters.

Notation and Terminology

We follow standard notation in [DP90, Sip97]. We distinguish between $X \mathrel{\hat{=}} Y$, *X is defined as Y*, and $X = Y$, *X is equal to Y*. When it is clear from the context, we drop superscripts and subscripts. We shorten *if and only if* to *iff*.

Sets. We denote sets with capital letters $A, B, C, \ldots$, etc. (usually starting at letter X) and elements of sets with lowercase letters $a, b, c, \ldots$, etc. Some exceptions include $\mathbb{N}$ for the set of natural numbers, $\mathbb{Z}$ for the set of integers and $\mathbb{B}$ for the set of booleans. We use subscripts to represent a family of sets, e.g. $X_1, \ldots, X_n$ where X_i is a set.

For the remainder of this section, let X be a set. We denote the cardinality of X by $|X|$, the powerset of X as $\wp(X)$, and the subset ordering (strict subset ordering) by $\subseteq$ ($\subset$).

Sequences. We denote by X^* the set of finite sequences of elements of X whose elements will be represented with lowercase letters. We denote a sequence $(x_1, x_2, \ldots, x_n)$ as $x_1 x_2 \ldots x_n$ where the empty sequence is denoted by the special symbol ε. Let $\mathit{perm}(X)$ represent all permutations of X.

The *length* of a sequence $u \mathrel{\hat{=}} x_1 x_2 \ldots x_n$, denoted $|u|$, is n, where $|\epsilon| \mathrel{\hat{=}} 0$. For any two sequences $u \mathrel{\hat{=}} x_1 x_2 \ldots x_n$ and $v \mathrel{\hat{=}} y_1 y_2 \ldots y_m$, the concatenation operation $u \cdot v \mathrel{\hat{=}} x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$. When clear from the context, we use uv instead of $u \cdot v$. The

sequence u is the prefix of $w \mathrel{\hat{=}} uv$, denoted $u \preccurlyeq w$, where $\preccurlyeq$ is the prefix order. We denote by $u[i]$ the element at the i-th position in u; by $u[i,j]$ the sequence of elements of u between positions i and j; and by $u[i\ldots]$ the subsequence of u starting at position i. A k-tuple is a finite sequence of length k, e.g. the pair (2-tuple) composed of the first element a and the second element b is denoted as (a,b). We denote by π_i the standard projection defined as $\pi_i(x_1,\ldots,x_k) \mathrel{\hat{=}} x_i$.

In a language-theoretical context, we call the set X an alphabet and use the Greek alphabet to represent sets and their elements, i.e. $\Sigma, \Delta, \ldots$, instead of capital letters, and $\sigma, \delta, \ldots$, to represent sequences. For an alphabet Σ we refer to the sequences $\sigma \in \Sigma^*$ as *strings* over Σ.

Relations and Functions. The n-ary Cartesian product over $X_1,\ldots,X_n$ denoted by $X_1 \times \ldots \times X_n$ is the set of ordered n-tuples $\{(x_1,\ldots,x_n) \mid x_i \in X_i\}$. For a set X, we denote by X^n the n-ary Cartesian power of a set X where $X^n \mathrel{\hat{=}} \underbrace{X \times \ldots \times X}_{n}$. An n-ary relation R on X is a subset of X^n, i.e. a set of ordered n-tuples $\{(x_1,\ldots,x_n) \mid x_i \in X\}$. 1-ary, 2-ary and 3-ary relations are known, respectively, as unary, binary and ternary relations. We denote by $id \mathrel{\hat{=}} \{(x,x) \mid x \in X\}$ the binary identity relation on a set X.

Since relations are sets, we also denote them by capital letters (usually starting at R). However, for certain relations (such as orders and independence relations) we use symbols such as $\subseteq, \sqsubseteq, \leqslant, \preccurlyeq, \diamond$ and $\subset, \sqsubset, <, \prec$, respectively, for the strict version (if applicable). Further, will use infix notation to represent elements of orders, i.e. $x \leqslant y \mathrel{\hat{=}} (x,y) \in \leqslant$.

The *image* of a set $X \subseteq A$ w.r.t. $R \subseteq A \times C$ is $R(X) \mathrel{\hat{=}} \{y \in C \mid x \in X \text{ and } (x,y) \in R\}$. The *preimage* of $X \subseteq C$ is $R^{-1}(X)$. The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is the relation $S \circ R \mathrel{\hat{=}} \{(a,c) \mid (a,b) \in R \text{ and } (b,c) \in S \text{ for some } b \in B\}$.

Let R be a relation on a set X. We denote by $R^{\leftrightarrow} \mathrel{\hat{=}} R \cup R^{-1}$ the symmetric closure of R and by $R^+ \mathrel{\hat{=}} \bigcup_{i \in \mathbb{N}} R^i$ the transitive closure of R where $R^1 \mathrel{\hat{=}} R$ and $R^n \mathrel{\hat{=}} R^{n-1} \circ R$.

We denote functions with lowercase letters $a, b, c, \ldots$, (usually starting at letter f). A function f with domain X and codomain Y is denoted as $f\colon X \to Y$. We denote the case where f is a *partial* function with $f\colon X \rightharpoonup Y$, where $f \downarrow x$ denotes that $f(x)$ is defined and $f \uparrow x$ denotes that $f(x)$ is undefined. Similarly to the composition of relations, the composition of $f\colon X \to Y$ and $g\colon Y \to Z$ is the function $(g \circ f)(x) \mathrel{\hat{=}} g(f(x))$.

Given a binary relation $R \subseteq X \times X$ and a function $f\colon X \to Y$, the pointwise lifting of R by f is the relation $R_f \subseteq Y \times Y \mathrel{\hat{=}} \{(f(a), f(b)) \mid (a,b) \in R\}$.

Maps. A map $m\colon K \to V$ is a function from elements in K known as *keys* to elements in V known as *values*. We abbreviate the type as $\mathsf{Map}\ K\ V$. We denote by $m(k)$ the lookup of a key k in a map m which returns the default value $\bot$ if there is no value associated with the key in the map. We denote by $m(k_1, k_2, \ldots, k_n) \mapsto (v_1, v_2, \ldots, v_n)$, the update of m where the new value of $k_1, k_2, \ldots, k_n$ is, respectively, $v_1, v_2, \ldots, v_n$. The difference between two maps m_1, m_2 is defined as:

$$(m_1 \backslash\backslash m_2)(k) \mathrel{\hat{=}} \begin{cases} m_2(k) & \text{if } m_1(k) \neq m_2(k) \\ \bot & \text{otherwise} \end{cases} \tag{2.1}$$

The underlying binary relation of a map m is defined as $\mathsf{elems}\colon \mathsf{Map}\ K\ V \to K \times V$, $\mathsf{elems}(m) \mathrel{\hat{=}} \{(k, v) \mid m(k) = v \text{ and } v \neq \bot\}$. Finally, we define the functions $\mathsf{keys} \mathrel{\hat{=}} \pi_1 \circ \mathsf{elems}$ and $\mathsf{vals} \mathrel{\hat{=}} \pi_2 \circ \mathsf{elems}$ that retrieve, respectively, the set of keys defined in the map and the set of values.

2.1 Order Theory Primer

Standard properties of a binary relation R on X include:

- Reflexivity: $\forall x \in X.\ (x, x) \in R$;
- Irreflexivity: $\forall x \in X.\ (x, x) \notin R$;
- Symmetry: $\forall x, y \in X.\ (x, y) \in R$ iff $(y, x) \in R$;
- Anti-Symmetry: $\forall x, y \in X.$ if $(x, y) \in R$ and $(y, x) \in R$, then $x = y$;
- Asymmetry: Irreflexive and anti-symmetric relation;
- Transitivity: $\forall x, y, z \in X.$ if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$;
- Totality: $\forall x, y \in X. (x, y) \in R$ or $(y, x) \in R$.

Ordered Sets. An order on a set X is a transitive binary relation on X. An *ordered set* is a pair $O \mathrel{\hat{=}} (X, \leqslant_O)$ where X is a set and $\leqslant_O$ is an order on X. When it is clear from the context, we refer to the ordered set O by the order $\leqslant$ dropping the subscript.

Various orders satisfying additional properties include:

- Preorder/Quasi-order: Reflexive order;
- Partial Order: Reflexive and anti-symmetric order;

- Total Order: Total and anti-symmetric order;
- Partial Equivalence: Symmetric order;
- Equivalence: Reflexive and symmetric order;

Partial Orders. We use the term *poset* to refer to a partially ordered set. In this dissertation, we will extensively use strict partial (and strict total) orders which are the asymmetric orders associated with partial (and total) orders. In particular, for a strict partial order $<$ we denote the inverse as $>\hat{=}\{(y,x) \mid x < y\}$ and their respective reflexive closures $\leqslant\hat{=}< \cup\, id$ and $\geqslant\hat{=}> \cup\, id$.

Let $P \mathrel{\hat{=}} (X, \leqslant_P)$ be a poset. When clear from the context, we refer to $\leqslant_P$ as $\leqslant$. An element $y \in X$ *covers* $x \in X$, denoted $x \prec y$ if $x \leqslant y$ and $x \leqslant z \leqslant y$ implies $z = x$. We depict partial orders using *Hasse diagrams* which explicitly represent the covering relation. We denote the special elements bottom and top by, respectively, $\bot, \top \in X$, that satisfy $\bot \leqslant x$, $x \leqslant \top$ for all $x \in X$.

Let $Q \subseteq X$. Q is a downward-closed set if, whenever $x \in Q$ and $y \leqslant x$, then $y \in Q$. The downward-closed set associated with an element $x \in X$ is the set $\downarrow x \mathrel{\hat{=}} \{y \in X \mid y \leqslant x\}$. By duality, Q is an upward-closed set if, whenever $x \in Q$ and $x \leqslant y$, then $y \in Q$. The upward-closed set associated with an element $x \in X$ is the set $\uparrow x \mathrel{\hat{=}} \{y \in X \mid x \leqslant y\}$. The minimal elements of Q is the set $\min_Q \mathrel{\hat{=}} \{x \in Q \mid \forall y \in Q.y \leqslant x \implies y = x\}$. If $\min_Q = \{x\}$, then we say that x is the *least* or *minimum* element of Q. Dually, by reversing the order, one obtains the set of maximal elements of Q, $\max_Q$ and the *greatest* or *maximum* (if it exists). The set of lower bounds of Q is the set $Q^l \mathrel{\hat{=}} \{x \in X \mid \forall y \in Q.\ x \leqslant y\}$. The greatest element of Q^l is known as the *greatest lower bound* and denoted $\sqcap_Q$. Dually, the set of upper bounds is $Q^u \mathrel{\hat{=}} \{x \in X \mid \forall y \in Q.\ y \leqslant x\}$. The least element of Q^u is known as the *least upper bound* and denoted $\sqcup_Q$. We say that Q is *direct* if it is non-empty and for every pair of elements $x, y \in Q$, then $\{x,y\}^u \subseteq Q$. A poset P is a *directed complete partial order* (DCPO) if it has a bottom element $\bot$ and the $\sqcup_D$ exists for every directed subset D of P.

It is clear that a total order is also a partial order. A total order is also know as a *chain* or a *linear order*. A linear order $L \mathrel{\hat{=}} (X, \leqslant_L)$ is a *linear extension* or *linearization* of P if $\leqslant_P$ is a subset of $\leqslant_L$. A consequence the Szpilrajn extension theorem [Szp30] is that a partial order P is the intersection of its linear extensions [BP82]. Finally, a set of linear extensions L of P is a *realizer* of P if $P = \bigcap L$. Clearly, the set of linear extensions of a partial order is a realizer. We denote by $lin(P)$ the set of linear extensions of P.

Equivalence Relations. We denote equivalence relations with the symbol $\equiv$. A *partition* of X is a set of non-empty sets of X such that any two of them are disjoint and the union of all is X. The equivalence class of $x \in X$ w.r.t. $\equiv$ is the set $\{x' \in X \mid x \equiv x'\}$ and denoted as $[x]_{\equiv}$. If $\equiv$ is not reflexive, $[x]_{\equiv}$ may not be defined. If $\equiv$ is clear from the context we will write $[x]$ for $[x]_{\equiv}$. The quotient $X/\equiv$ contains the equivalence classes of X with respect to $\equiv$ defined as $\{[x] \mid x \in X\}$ and is a partition of X.

Given a total function $f\colon X \to Y$, there is an equivalence relation $\equiv_f$ on X defined as $\{(x, y) \in X \times X \mid f(x) = f(y)\}$. It follows that if f is a partial function, we obtain a partial equivalence relation.

Lattices. A lattice $(L, \sqsubseteq_L, \sqcap_L, \sqcup_L)$ is a poset with a binary, least upper bound operator $\sqcup_L$ called *join* and a binary, greatest lower bound operator $\sqcap_L$ called *meet*. When no ambiguity occurs, we omit subscripts and denote a lattice as $(L, \sqsubseteq)$ or simply L. A *bounded lattice* has a least element $\bot$ and a greatest element $\top$, called bottom and top, respectively. A lattice L is *complete* if every subset $S \subseteq L$ has a meet $\bigsqcap S$ and a join $\bigsqcup S$. Every complete lattice is bounded and a DCPO.

Functions and Fixpoints. Consider lattices $(L, \sqsubseteq, \sqcap, \sqcup)$ and $(M, \preccurlyeq, \wedge, \vee)$. A function $g\colon L \to M$ is *monotone* if, for all x and y in L, $x \sqsubseteq y$ implies $g(x) \preccurlyeq g(y)$. The function g is a *lattice homomorphism* if it satisfies $h(x \sqcap y) = h(x) \wedge h(y)$ and $h(x \sqcup y) = h(x) \vee h(y)$ for all x and y. A homomorphism of complete lattices must further commute with arbitrary meets and joins, of Boolean lattices must commute with complements, etc. A homomorphism with respect to some set of lattice operations is one that commutes with those operations. A bijective homomorphism is called an *isomorphism*.

The function g is *additive* if every x and y satisfy $g(x \sqcup y) = g(x) \vee g(y)$, is *completely additive* every set $S \subseteq L$, satisfies $g(\bigsqcup S) = \bigvee g(S)$. The definitions of *multiplicative* and *completely multiplicative* functions follow by replacing joins with meets. The function g is *Scott-continuous* if for every directed subset D of a poset L (not necessarily a lattice) we have that $\sqcup_{f(D)} = f(\sqcup_D)$. Note that a *Scott-continuous* function is necessarily monotone.

An element $x \in L$ is a fixpoint of the function g iff $g(x) = x$. The following two theorems related to fixpoints are relevant to the work in this dissertation:

- The *Knaster-Tarski theorem* states that the set of a fixpoints of a monotone function over a complete lattice is also a complete lattice.

- The *Kleene fixed-point theorem* states that the least fixpoint (lfp) of a Scott-continuous function $f: L \to L$ over a DCPO $(L, \sqsubseteq)$ is the least upper bound of the chain $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \ldots f^n(\bot) \sqsubseteq \ldots$.

Galois connections and Closures. Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. A *Galois connection* $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \preccurlyeq)$, is a pair of functions $\alpha : L \to M$ and $\gamma : M \to L$ satisfying that for all $x \in L$ and $y \in M$, $\alpha(x) \preccurlyeq y$ exactly if $x \sqsubseteq \gamma(y)$. When the orders involved are clear, we write $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$. A Galois connection is a *Galois insertion* if γ is injective. A function in $L \to L$ is called an *operator*. The operator f is *extensive* if $x \sqsubseteq f(x)$, *reductive* if $f(x) \sqsubseteq x$ and *idempotent* if $f(f(x)) = f(x)$. An operator is an *upper closure* if it is monotone, idempotent and extensive and is a *lower closure* if it is monotone, idempotent and reductive. A closure is an upper or a lower closure.

2.2 Abstract Interpretation Primer

Abstract Domains. A *domain* in the sense of abstract interpretation, is a complete lattice equipped with monotone functions, called *transformers*, and non-monotone operations called widening and narrowing for enforcing the convergence of an analysis. We do not consider widening and narrowing in our work. We use signatures to compare transformers from different domains.

Fix a signature containing a set of symbols *Sig* with an arity function $ar : Sig \to \mathbb{N}$. A domain $\mathcal{A} = (A, F_A)$ is a complete lattice A and a collection of transformers $f^{\mathcal{A}} : A^{ar(f)} \to A$, for each symbol in *Sig*. For notational simplicity, the next definition uses unary transformers. A domain $\mathcal{A} = (A, F_A)$ is an *abstraction* of $\mathcal{C} = (C, F_C)$ if there exists a Galois connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ such that for all f in *Sig*, $\alpha \circ f^C \sqsubseteq f^A \circ \alpha$. Note that since we have a Galois connection it holds that for all $c \in C$ and $a \in A$, $\alpha(c) \sqsubseteq a$ exactly if $c \leqslant \gamma(a)$. The *best abstract transformer* corresponding to f^C is the function $\alpha \circ f^C \circ \gamma$, which represents the most precise approximation of a single application of f^C. We assume that transformers are *bottom-strict*, i.e. $f(\bot) = \bot$.

We use the following notion of abstract domain equivalence for proving properties of our abstractions.

Definition 1. *Two abstractions $\mathcal{A} = (A, F_A)$ $\mathcal{B} = (B, F_B)$, of a domain $\mathcal{C} = (C, F_C)$, specified by Galois connections $(C, \leqslant) \underset{\alpha_{\mathcal{A}}}{\overset{\gamma_{\mathcal{A}}}{\leftrightarrows}} (A, \sqsubseteq_A)$ and $(C, \leqslant) \underset{\alpha_{\mathcal{B}}}{\overset{\gamma_{\mathcal{B}}}{\leftrightarrows}} (B, \sqsubseteq_B)$ are* equivalent *if there exists a complete lattice isomorphism $h : A \to B$, that is an isomorphism for the transformers in the domain and satisfies that $h(\alpha_A(c)) = \alpha_B(c)$ and $\gamma_A(a) = \gamma_B(h(c))$.* ∎

Fixpoint Approximation. The soundness theorem of abstract interpretation states that in order to over-approximate a fixpoint, it suffices to over-approximate the lattice and transformer used to define the fixpoint. Formally, given a Galois connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ and a sound transformer $f^A : A \to A$ of $f : C \to C$, it follows that $\alpha(\mathsf{lfp}(f)) \sqsubseteq \mathsf{lfp}(f^A)$.

2.3 Models of Concurrency

We now present the models of concurrency relevant to the work in this dissertation.

Let *Act* be a set of actions and *State* be a set of states. A labelled relation on *State* is a subset of $Rel \mathrel{\hat{=}} State \times Act \times State$. A *transition* (s, a, t) is an element of a labelled relation in which s is the *source*, t is the *target* and a is the *label*. Moreover, s is the *predecessor* of t and t is the *successor* of s. We denote the set of finite transition sequences by Rel^*.

The function $act : Rel^* \to Act^*$ retrieves the sequence of actions associated with a sequence of transitions where: $act(\varepsilon) \mathrel{\hat{=}} \varepsilon$ and $act((r, a, s) \cdot \sigma) \mathrel{\hat{=}} a \cdot act(\sigma)$. We overload *act* for sets of transition sequences, $act(T) \mathrel{\hat{=}} \{act(\sigma) \mid \sigma \in T\}$.

2.3.1 Labelled Transition Systems

A *labelled transition system* (LTS) $M \mathrel{\hat{=}} (State, Trans, \tilde{s})$ consists of a set of states, a *transition relation* $Trans \subseteq Rel$ and an initial state $\tilde{s} \in State$ [1]. A transition (s, a, t) is *enabled* in a state s if (s, a, t) is in *Trans*. We denote by $en(s)$ the set of transitions enabled in a state s. When clear from context, we also denote by $en(s)$ the set of actions enabled in a state s. An *initial* transition is one that is enabled in $\tilde{s}$. We refer to the sequence $((s_1, a_1, t_1), (s_2, a_2, t_2), \ldots, (s_n, a_n, t_n))$ as a *transition sequence* (of length n) where the first and last states in the sequence are s_1 and t_n, while (s_1, a_1, t_1) and (s_n, a_n, t_n) are the first and last transitions.

A transition sequence is *feasible* if it only contains transitions from *Trans* and is *infeasible* otherwise. Furthermore, a transition sequence of length n is *consistent* if adjoining target and source states are equal: for all $1 \leq i < n$. $t_i = s_{i+1}$. If a transition sequence is not consistent we say that it is *inconsistent*. Feasibility and consistency are different concepts since there are transition sequences that infeasible and consistent, and sequences that are feasible and inconsistent.

[1] We follow standard practice in the verification/model checking community and associate initial states to a transition system.

A *history* is a feasible, consistent transition sequence. Following standard practice, we represent histories with the notion of *path*. A path of length n is an alternating sequence of states and actions, $s_1, a_1, s_2, \ldots, a_n, s_{n+1}$, in which every tuple (s_i, a_i, s_{i+1}) is an element of the transition relation. In that case, we typically represent a transition (s, a, t) as $s \xrightarrow{a} t$. Given two states $s, t \in \mathit{State}$, and $\sigma \hat{=} a_1 \cdot a_2 \ldots a_n \in \mathit{Act}^*$ (a_1 concatenated with a_2, $\ldots$ until a_n), we denote by $s \xrightarrow{\sigma} t$ the fact that there exist states $s_1, \ldots, s_{n-1} \in \mathit{State}$ such that $s \xrightarrow{a_1} s_1$, $\ldots$, $s_{n-1} \xrightarrow{a_n} t$.

We now recall the main properties of interest for verification, which are reachability of states and the analogous property, firability of transitions.

A history $s \xrightarrow{\sigma} t$ is *firable* if $s = \tilde{s}$. In that case, we say that t is *reachable* and σ is a *run* of M. We assume that ε is a valid run, i.e. $\tilde{s} \xrightarrow{\varepsilon} \tilde{s}$. Since M can be non-deterministic, we denote by $\mathit{st}(\sigma)$ the set of states reachable from σ. More formally, $\mathit{st}(\sigma) \hat{=} \left\{t \in \mathit{State} \mid \tilde{s} \xrightarrow{\sigma} t\right\}$ with $\mathit{st}(\varepsilon) \hat{=} \{\tilde{s}\}$. A transition is *firable* if it is the last transition of a firable history and an action is firable if it is the label of a firable transition. A state is a *deadlock* if $\mathit{en}(s) = \varnothing$.

We will use the domain of transition sequences, which consists of all possible sets of transition sequences and transformers for extending such sequences.

Definition 2 (Domain of transition sequences)**.** *The* lattice of transition sequences *is* $(\wp(\mathit{Rel}^*), \subseteq)$.

The forward and backwards enabled *transformers* $\mathit{en}_\rightarrow, \mathit{en}_\leftarrow : \wp(\mathit{Rel}^*) \to \wp(\mathit{Rel})$ *map a transition sequence to the set of transitions enabled either at the end or before the beginning of the sequence.*

$$\mathit{en}_\rightarrow(X) \hat{=} \{(s, b, t) \in \mathit{Trans} \mid \tau(r, a, s) \in X \text{ or } (\varepsilon \in X \text{ and } s = \tilde{s})\}$$
$$\mathit{en}_\leftarrow(X) \hat{=} \{(r, a, s) \in \mathit{Trans} \mid (s, b, t)\tau \in X\}$$

The transformers below have type $\wp(\mathit{Rel}^*) \to \wp(\mathit{Rel}^*)$.

$$\mathit{ext}_\rightarrow(X) \hat{=} \{\tau(s, b, t) \mid \tau \in X, (s, b, t) \in \mathit{en}_\rightarrow(\{\tau\})\}$$
$$\mathit{ext}_\leftarrow(X) \hat{=} \{(r, a, s)\tau \mid \tau \in X, (r, a, s) \in \mathit{en}_\leftarrow(\{\tau\})\}$$

■

The *forward extension* transformer $\mathit{ext}_\rightarrow$ extends the end of a sequence with transitions that respect the transition relation and the *backward extension* transformer $\mathit{ext}_\leftarrow$ extends a sequence backwards in a similar manner. The transformers above are existential in that they rely on the existence of a sequence in their argument. These

transformers have universal variants defined in the standard way by complementation [CC00]. In this dissertation, we will not use the backwards transformers.

The standard characterization of state reachability from initial states lifts to histories as the following fixpoint:

$$Hist(M) \mathrel{\hat{=}} \mathsf{lfp}\ x.\ \{\varepsilon\} \cup ext_{\rightarrow}(x)$$

Furthermore, the set of runs of M (known as the *interleaving space*) is defined as:

$$Runs(M) \mathrel{\hat{=}} \left\{\sigma \in Act^{*} \mid \tilde{s} \xrightarrow{\sigma} t \in Hist(M)\right\}$$

and the set of reachable states from the initial state as:

$$Reach(M) \mathrel{\hat{=}} \{s \in st(\sigma) \mid \sigma \in Runs(M)\}$$

2.3.2 Event Structures

Event structures [Win80] are a family of models of concurrency where the behavior of the system is represented using relations over a set whose elements are known as *events*. The exact type of these relations and the restrictions upon them leads to the many kinds of event structures in the literature (see [vGP04] for a comparative study on event structures). They have been extensively investigated as a behavioral model of Petri nets [Pet66] and the many variants are usually motivated by concurrent behavior in the many variants of Petri nets [PE01]. In this dissertation, we will use event structures as a behavioral model of concurrent programs.

We now recall the general definition of event structures in [vGP95, vGP09].

Definition 3 (Event Structure)**.** *An event structure is a pair* $\mathsf{E} \mathrel{\hat{=}} (Event, \vdash)$ *where:*

- *Event is a set of events and*
- $\vdash \subseteq \wp(Event) \times \wp(Event)$ *is a binary relation on sets of events called* enabling relation.

■

Since we are interested in behavioral representations, we assume that event structures are *pure*, i.e. their enabling relation satisfies: if $X \vdash Y$, then $X \cap Y = \varnothing$.

The notion of *state* in an event structure is captured in the definition of *configuration*.

Definition 4 (Configuration)**.** *Let* $\mathsf{E} \mathrel{\hat{=}} (Event, \vdash)$ *be an event structure. A configuration is a set of events* $C \in \wp(Event)$ *satisfying:*

For all $X \subseteq C$*, there exists* $Y \subseteq C$ *such that* $Y \vdash X$*.*

We denote the set of configurations of E *by* $conf(\mathsf{E})$*.* ■

The domain of configurations of event structures are known as *configuration structures.*

Definition 5 (Configuration Structure)**.** *A configuration structure is a pair* $\mathsf{C} \mathrel{\hat{=}} (Event, C)$ *where:*

- *Event is a set of events and*
- $C \subseteq \wp(Event)$ *is a collection of subsets of events.*

■

Given a pure event structure E, we define its associated configuration structure as $\mathsf{L}(\mathsf{E}) \mathrel{\hat{=}} (Event, conf(\mathsf{E}))$. Configuration structures and pure event structures are shown to be equivalent in [vGP09, Theorem 2] via a construction of a pure event structure from a configuration structure. We assume that the event and the configuration structures are *rooted*, i.e. $\varnothing \vdash \varnothing$ and $\varnothing \in C$.

We now provide some examples of concurrent behavior modeled with event structures.

Example 1. *Let* $Event = \{a, b, c\}$*.*

The event structure where b causally happens after a is given by the enabling relation $\varnothing \vdash X$ *for* $X \neq \{b\}$ *and* $\{a\} \vdash \{b\}$*.*

The event structure where a and b are concurrent is given by the enabling relation $\varnothing \vdash X$ *for* $X \subseteq \{a, b\}$*,* $\{a\} \vdash b$ *and* $\{b\} \vdash \{a\}$*.*

The event structure where a and b are concurrent and c is causally dependent on both a and b is given by the enabling relation $\varnothing \vdash X$ *for* $X \in \wp(\{a, b\}) \cup \{a, b, c\}$ *and* $\{a, b\} \vdash \{c\}$*.*

The event structure where a and b are in symmetric binary conflict is given by the enabling relation $\varnothing \vdash \{a\}$ *and* $\varnothing \vdash \{b\}$*. If* $\{a\} \vdash \{a, b\}$*, we have a situation of asymmetric conflict,i.e. b is in conflict with a but not vice versa. Note that this event structure is not pure.* ◁

We now associate a LTS representation to a configuration structure where states are configurations and labels are non-empty sets of events.

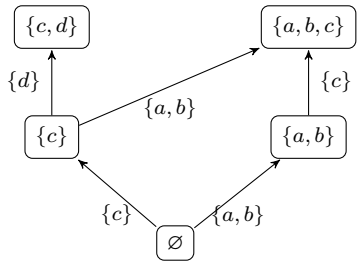
Definition 6 (Transition System of a Configuration Structure)**.** *Let* $\mathsf{C} \triangleq (Event, C)$ *be the configuration structure. The* LTS *associated to* C *is the system* $\mathsf{M}_{\mathsf{C}} \triangleq (C, Trans \subseteq C \times \wp(Event) \times C, \varnothing)$ *where:*

$$(C_1, X, C_2) \Leftrightarrow C_1 \subset C_2,\ X = C_2 \setminus C_1 \text{ and } (\nexists C' \in C.\ C_1 \subset C' \subset C_2).$$

■

Definition 6 can also be applied to a pure event structure E by considering its configuration structure $\mathsf{L}(\mathsf{E})$.

Example 2. *Let* $\mathsf{C} \triangleq (E \triangleq \{a, b, c, d\}, C \triangleq \{\{c\}, \{a, b\}, \{c, d\}, \{a, b, c\}\})$ *be a configuration structure. Below, we show the* LTS *that represents* C *on the left and the enabling relation of the corresponding pure event structure* $(E, \vdash)$ *on the right:*



$$\begin{aligned}
\varnothing &\vdash X \text{ for } X \in C \\
\{a\} &\vdash X \text{ for } X \in \{\{b\}, \{b, c\}\} \\
\{b\} &\vdash X \text{ for } X \in \{\{a\}, \{a, c\}\} \\
\{c\} &\vdash X \text{ for } X \in \{\{d\}, \{a, b\}\} \\
\{d\} &\vdash \{c\}
\end{aligned}$$

Note that $Y \vdash X$ *does not imply that* Y *or* $Y \cup X$ *is a configuration.* ◁

Despite the generality of the model, it is not entirely clear if it solves Pratt's motivation for higher dimensional automata [Pra00]. Let $E \triangleq \{a, b\}$. Pratt argues that in event structures, labelling of events is required to distinguish between concurrency and mutual exclusion. A potential solution would be to consider the domain of configurations $\{\varnothing, \{a, b\}\}$ for $a \parallel b$ and $\wp(\{a, b\})$ for $ab + ba$. However, that does not seem totally appropriated as it not clear what is the domain of configurations for the system where a is concurrent with b and a third event c is a causal successor of a and in conflict with b. Glabbeek and Plotkin adopt an asynchronous interpretation (see [vGP09, Section 2.1] for details) where events are executed concurrently only if they can be executed in any order, thus considering $\wp(\{a, b\})$ the domain of configurations for $a \parallel b$. As our main use of event structures will be as a representation of a language, this is not a major problem and we consider labelling of events.

The possible identification of the causes of an event via disjunctive or conjunctive causality leads to three instances of event structures:

- In the general event structures of [Win88], events can have multiple causes within a configuration (see [Win88, Example 1.1.7] for an example).

- In stable event structures, disjunctive causality can only occur across configurations, i.e. an event might have different causes in different configurations but within the same configuration, there is a unique set of causes.
- In prime event structures, the causes of an event are unique across the structure, i.e. we have conjunctive causality.

In [vGP09, Section 4], these event structures are characterized as instances of the event structures in Definition 3.

Prime Event Structures

We now focus on prime event structures [NPW81] as they will be the main model of concurrency used throughout this dissertation.

Definition 7 (Labelled Prime Event Structure)**.** *Given a set A, an* A-labelled prime event structure *(A-PES or* PES *in short) is a tuple $\mathcal{E} \mathrel{\hat{=}} (Event, <, \#, h)$ where:*

- *$Event$ is a set of* events*,*
- *$< \subseteq Event \times Event$ is a strict partial order on $Event$, called* causality*,*
- *$\# \subseteq Event \times Event$ is a symmetric, irreflexive relation, called* conflict *and*
- *$h\colon Event \to A$ is a labelling function from events to elements in A*

satisfying:

- *(finite causes) for all $e \in Event$, $\{e' \in Event : e' < e\}$ is finite, and* (2)
- *(hereditary conflict) for all $e, e', e'' \in Event$, if $e \# e'$ and $e' < e''$, then $e \# e''$.* (3)

■

The *causes* of an event $e \in Event$ is the set $\lceil e \rceil \mathrel{\hat{=}} \{e' \in Event : e' < e\}$ of events that need to happen before e for e to happen.

The notion of *state* in a PES is captured in their definition of configuration.

Definition 8 (Configuration of a PES)**.** *Let $\mathcal{E} \mathrel{\hat{=}} (Event, <, \#, h)$ be a* PES*. A* configuration *of $\mathcal{E}$ is any finite set $C \subseteq Event$ satisfying:*

- *(causally closed) for all $e \in C$ we have $\lceil e \rceil \subseteq C$ and* (4)

- *(conflict free) for all* $e, e' \in C$*, it holds that* $\neg(e \# e')$. *(5)*

We denote by $conf(\mathcal{E})$ *the set of configurations of* $\mathcal{E}$. ■

We now define several useful notions related to a PES $\mathcal{E} \hat{=} (Event, <, \#, h)$.

An event $e \in Event$ is an *immediate predecessor* of an event $e' \in Event$, $e <^i e'$, iff $e < e'$ and there is no event $e'' \in Event$ such that $e < e'' < e'$. Clearly, $<$ is the transitive closure of $<^i$.

The *local configuration* of an event $e \in Event$ is the $\subseteq$-minimal configuration that contains e. More formally, $[e] \hat{=} \lceil e \rceil \cup \{e\}$. Note that $[e]$ and $\lceil e \rceil$ are configurations and that a configuration can be seen as a partial order – thus, a local configuration $[e]$ is a partial order where e is the maximum element.

The causes of an event can be defined compositionally using the immediate predecessor relation, i.e. $\lceil e \rceil = \bigcup_{e' <^i e} [e']$. The reflexive relation $\lesseqgtr \hat{=} < \cup >$ relates events by causality. We denote by $\not\lesseqgtr \hat{=} Event \times Event \setminus \lesseqgtr$ the complement of $\lesseqgtr$.

Two events $e, e' \in Event$ are in *immediate conflict*, $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup \lceil e' \rceil$ and $[e] \cup \lceil e' \rceil$ are configurations. We denote by $\#^i(e) \hat{=} \{e' \in Event \mid e \#^i e'\}$ the set of immediate conflicts with an event e and by $\wr \hat{=} \# \cup id$ the reflexive closure of $\#$.

Two events $e, e' \in Event$ are *concurrent*, $e \parallel e'$, iff they are different, not causally related and not in conflict. More formally, $\parallel \hat{=} Event \times Event \setminus (\lesseqgtr \cup \#)$. Thus, we exclude the possibility of $e \parallel e$, a behavior known as *auto-concurrency*.

Note that by definition $\{id, <, >, \#, \parallel\}$ is a partition of $Event \times Event$.

Let $C \in conf(\mathcal{E})$ be a configuration of $\mathcal{E}$.

The *extensions* of C, denoted $ext(C)$, are all those events not in C whose causes are included in C. Formally, $ext(C) \hat{=} \{e \in Event \mid e \notin C \text{ and } \lceil e \rceil \subseteq C\}$.

We denote by $en(C)$ the set of events *enabled* by C, formally defined as $en(C) \hat{=} \{e \in ext(C) \mid C \cup \{e\} \in conf(\mathcal{E})\}$. Clearly the events enabled by C are always extensions of C. We denote the remaining events, $cext(C) \hat{=} \{e \in ext(C) \mid \exists e' \in C, e \#^i e'\}$ by *conflicting extensions*. By definition, $\{en(C), cext(C)\}$ is a partition of $ext(C)$.

We denote by $C \oplus E$ the fact that E is a feasible extension of C, i.e. $C \cup E \in conf(\mathcal{E})$ and $C \cap E = \varnothing$. A configuration C is *maximal* when $en(C) = \varnothing$.

A PES $\mathcal{E}$ is *deterministic* iff for every configuration $C \in conf(\mathcal{E})$, if $e, e' \in en(C)$ and $h(e) = h(e')$, then $e = e'$.

Let $U \subseteq Event$. For a n-ary relation R on $Event^n$, we denote by $R_U \hat{=} R \cap U^n$ the relation restricted to U^n.

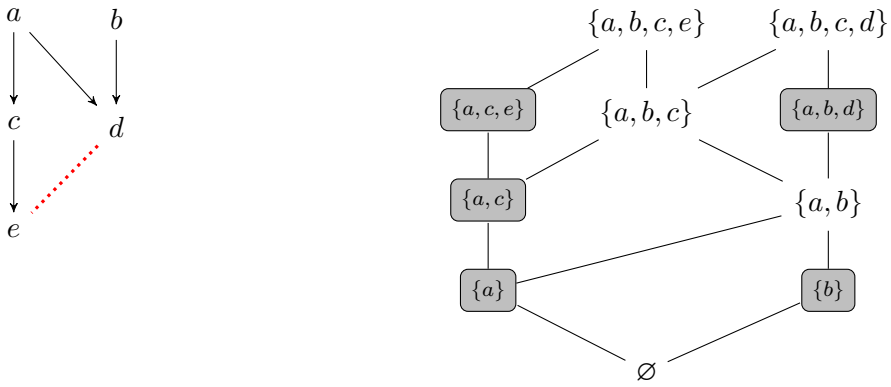


Figure 2.1: Example of a prime event structure and its domain of configurations.

Example 3. *Consider the prime event structure of the left of Figure 2.1. For simplicity of presentation, in our representation of prime event structures, we only represent immediate causality where $e <^i e'$ is denoted as $e \to e'$ and immediate conflict where $e \#^i e'$ is denoted as $e \cdots e'$.*

The PES *in Figure 2.1 is composed of five events: a, b, c, d and e. For example, the set $\{b, d\}$ is not a configuration since $\lceil d \rceil \not\subseteq \{b, d\}$. Also, the set $\{a, b, c, d, e\}$ is not a configuration since d is in conflict with e. Consider the configuration $C \hat{=} \{a, b\}$. It is simple to check that C is a configuration as both a and b have the empty set as their causes and they are not in conflict. The extensions of C are $\{c, d\}$. In this case, we have that $en(C) = ext(C)$. Thus, we have that $C \oplus \{c, d\}$. Consider the configuration $C' = C \cup \{c, d\}$. This configuration is maximal since $en(C') = \varnothing$. The event e is a conflicting extension of C'. Note that while the intersection of two configurations is a configuration, the union of two configurations is not necessarily a configuration. For example, $\{a, b, d\}$ and $\{a, c, e\}$ are configurations, but $\{a, b, c, d, e\}$ is not.* ◁

Prime Configuration Structures. The configuration structure $\mathsf{L}(\mathcal{E})$ associated with a PES $\mathcal{E}$ is called a *prime configuration structure*. In [NPW81], it is shown that set of configuration of a PES ordered by set inclusion correspond to coherent prime algebraic posets where events are prime elements of the poset. A representation theorem between prime event structures and coherent prime algebraic posets shows that we can move between coherent prime algebraic posets and prime event structures without loss of information.

Example 4. *In Figure 2.1, we present a* PES *on the left and its configuration structure on the right as a Hasse diagram. The representation theorem allows to retrieve an isomorphic* PES *by considering the join-irreducible elements of the configuration structure as events. These are denoted in Figure 2.1 by gray boxes.* ◁

Language of a PES**.** The language of a PES $\mathcal{E}$, $\mathcal{L}(\mathcal{E}) \hat{=} \bigcup_{C \in \mathsf{L}(\mathcal{E})} lin(C)$, is the union of the linear extensions of $\mathsf{L}(\mathcal{E})$. Furthermore, we define the set of *interleavings* of a configuration C as $inter(C) \hat{=} \{h(e_1), \ldots, h(e_n) \mid e_1 \ldots e_n \in lin(C)\}$.

Example 5. *The language of the* PES *in Figure 2.1 is the prefix-closure of the union of linearizations of the set of maximal configurations $\{a, b, c, e\}$ and $\{a, b, c, d\}$. For example $abcd \in \mathcal{L}(\mathcal{E})$ but $adbc \notin \mathcal{L}(\mathcal{E})$.* ◁

Orders on PES**.** We now recall several orders on PES. For the remainder of this section, consider the PES $\mathcal{E} \hat{=} (E, <, \#, h)$ and $\mathcal{E}' \hat{=} (E', <', \#', h')$.

Definition 9 (Prefix-order)**.** *We say that $\mathcal{E}$ is a* prefix *of $\mathcal{E}'$, denoted $\mathcal{E} \trianglelefteq \mathcal{E}'$, iff*

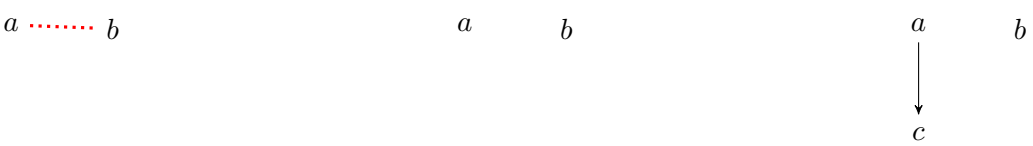
1. $< = <' \cap (E \times E)$,
2. $\# = \#' \cap (E \times E)$,
3. $h = h' \cap (E \times E)$ *and*
4. *E is causally closed in $\mathcal{E}'$, i.e., for all $e \in E$ and $e' \in E'$, if $e' < e$, then $e' \in E$.*

■

Definition 10 (Configuration-order)**.** *We say that $\mathcal{E}$ is a* configuration-prefix *of $\mathcal{E}'$ denoted $\mathcal{E} \trianglelefteq_{\mathsf{L}} \mathcal{E}'$ iff $\mathsf{L}(\mathcal{E}) \subseteq \mathsf{L}(\mathcal{E}')$.* ■

Example 6. *Consider the following* PES*s:*

$\mathcal{E}_1 \hat{=}$ $\quad$ $\mathcal{E}_2 \hat{=}$ $\quad$ $\mathcal{E}_3 \hat{=}$



We have that $\mathcal{E}_1 \trianglelefteq_{\mathsf{L}} \mathcal{E}_2 \trianglelefteq_{\mathsf{L}} \mathcal{E}_3$. However, $\mathcal{E}_1 \not\trianglelefteq \mathcal{E}_2$ and $\mathcal{E}_2 \not\trianglelefteq \mathcal{E}_1$ since their conflict relation differs. Further, we have that $\mathcal{E}_2 \trianglelefteq \mathcal{E}_3$. ◁

We now recall the definition of a PES morphism from [SNW96, Def 3.1].

Definition 11 (Morphism)**.** *A labelled $\mathcal{E}$ morphism from $\mathcal{E}_0 \hat{=} (E_0, <_0, \#_0, h_0)$ to $\mathcal{E}_1 \hat{=} (E_1, <_1, \#_1, h_1)$ is a pair of partial functions (η, λ) where $\eta : E_0 \multimap E_1$ and $\lambda : L_0 \multimap L_1$ satisfying:*

1. *$[\eta(e)] \subseteq \eta([e])$, if $\eta \downarrow e$;*
2. *$\eta(e) \Cup \eta(e')$ implies $e \Cup e'$, if $\eta \downarrow e, \eta \downarrow e'$ and*
3. *$\lambda \circ h_0 = h_1 \circ \eta$*

■

Example 7. *Consider the* PES*s in Example 6. We have that (id, id) is a morphism from $\mathcal{E}_1$ to $\mathcal{E}_2$. Note that the same pair of functions is not a morphism from $\mathcal{E}_2$ to $\mathcal{E}_1$.* ◁

2.3.3 Mazurkiewicz Trace Theory

Mazurkiewicz trace theory [Die95, Chapter 1] is based on the notion of *concurrent alphabet*. A concurrent alphabet is a pair $(\Sigma, \mathcal{I})$ where Σ is an alphabet and $\mathcal{I} \subseteq \Sigma \times \Sigma$ is a binary irreflexive relation called *independence*. The dependence relation $\mathcal{D}$ is the complement of independence, $\mathcal{D} \hat{=} (\Sigma \times \Sigma) \setminus \mathcal{I}$.

The *trace equivalence for I* is the least equivalence relation $\equiv_I$ in the monoid Σ^* such that for all $a, b \in \Sigma$ and $\sigma, \tau \in \Sigma^*$:

$$(a, b) \in I \implies \sigma ab\tau \equiv_I \sigma ba\tau.$$

The equivalence classes of $\equiv_I$ are called *traces* over I.

Let $\sigma \in \Sigma^*$ be a word.

We denote by $[\sigma]_{\equiv_I}$ the trace of σ. When no confusion arises, we refer to it as $[\sigma]$.

Given a language $L \subseteq \Sigma^*$, the trace language $[L]_{\equiv_I} \hat{=} \{[\sigma]_{\equiv_I} \mid \sigma \in L\}$ is defined as set of traces for each word in the language. If $L = (\bigcup [L]_{\equiv_I})$, we say that L is *existentially consistent* with I [Aal87].

It is well known that each trace can equivalently be seen as a partial order satisfying that a word belongs to the trace *if and only if* it is a linear order of the partial order associated to the trace [Maz87]. Such a partial order, called dependency graph, can be formally defined as follows (see [Die95, Chapter 2] for details).

The *dependency graph* of the trace $[\sigma]_{\mathcal{I}}$ is the unique labelled partial order $\mathcal{G}_{\mathcal{I},\sigma}$ inductively defined by:

- σ is the empty sequence and $\mathcal{G}_{\mathcal{I},\sigma}$ has an empty set of nodes; or

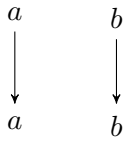
- $\sigma = \sigma' \cdot t$ and $\mathcal{G}_{\mathcal{I},\sigma}$ is the graph arising from extending $\mathcal{G}_{\mathcal{I},\sigma'}$ with one node e labelled by t and one arc from any node e' in $\mathcal{G}_{\mathcal{I},\sigma'}$ such that t and the label t' of e' satisfy $(t, t') \in \mathcal{D}$.

With this construction, one can check that for any two words $\sigma, \sigma' \in \Sigma^*$:

$$\sigma' \equiv_{\mathcal{I}} \sigma \quad \text{iff} \quad \sigma' \in [\sigma]_{\mathcal{I}} \quad \text{iff} \quad \sigma' \in \mathit{lin}(\mathcal{G}_{\mathcal{I},\sigma})$$

Example 8. *Consider the alphabet* $\Sigma \mathrel{\hat{=}} \{a, b\}$ *and the independence relation* $\mathcal{I} \mathrel{\hat{=}} \{(a, b), (b, a)\}$. *The two representations, equivalence class and dependence graph, of the trace* $[abab]_{\mathcal{I}}$ *are:*

$$\{aabb, abab, abba, bbaa, baba, baab\}$$



◁

2.3.4 True Concurrency Models

Event structures and Mazurkiewicz trace theory are examples of true concurrency models. Using process algebra terms [BK84], these are models that distinguish concurrency $a \parallel b$ from mutual exclusion $a \cdot b + b \cdot a$, where $\parallel$ denotes parallel composition, $\cdot$ denotes sequential composition and $+$ denotes non-deterministic choice. In the literature of semantics for concurrency we can find models that equip transition systems (a standard semantics for multithreaded programs) with additional information to derive notions of events, concurrency and conflict. Examples include *concurrent transition systems* [Sta89], *concurrent automata* [Dro90], *asynchronous transition systems* [Shi97], *labelled transition system with independence* [SNW96]. Similar ideas have also been applied in the context of process algebras [KP93] where the conflict relation is defined by induction from the syntax of CSP [BR84]. A desired property described in [SNW96, Section 3.2] is that these models should correspond to system representations of event structures.

2.4 Concurrent Programs and Semantics

Concurrent Programs. A *control-flow* graph $G \mathrel{\hat{=}} (L, A)$ is a finite, edge-labelled directed graph where a node $l \in L$ is a *location* and an edge $(l_s, S, l_t) \in A$ represents an *action* where S is a *statement* of a simple imperative language:

$$\text{Stat } S \quad ::= \quad x := e \mid \mathit{assume}(c) \mid \mathit{assert}(c) \mid \mathit{lock}(x) \mid \mathit{unlock}(x)$$

For simplicity of presentation, we assume that thread creation and join is modeled using mutexes. Furthermore, we assume expressions and conditions are standard and statements operate over *variables* of a set of symbols Var.

A concurrent program $P \mathrel{\hat{=}} (G_1, \ldots, G_n, \tilde{s})$ is a finite ordered collection of control-flow graphs $G_i \mathrel{\hat{=}} (L_i, A_i)$, where each G_i represents a thread, together with an *initial state* $\tilde{s}$. We identify each thread G_i by its position $i \in \mathbb{N}$ in the collection and assume that the sets of locations (and thus actions) are disjoint. Let $L \mathrel{\hat{=}} \bigcup L_i$ and $A \mathrel{\hat{=}} \bigcup A_i$ denote the set of locations and actions of all threads and $TId \mathrel{\hat{=}} \{1 \ldots n\}$ denote the set of thread identifiers. Since actions are disjoint, we define the function $tid : A \to TId$, $tid(a \in A_i) \mathrel{\hat{=}} i$, that retrieves the thread identifier of an action. For simplicity of presentation, we represent each thread in a standard basic imperative languages with conditionals, while loops and sequential composition, instead of the CFG representation.

Semantics. A *valuation* v: $\mathsf{Map}\ Var\ \mathsf{Type}$ is a map from variables in Var to values of a type $\tau \in \mathsf{Type}$. We let Val denote the set of all valuations.

A *state* of P is a tuple $s \mathrel{\hat{=}} (loc, val)$ where loc: $\mathsf{Map}\ TId\ L$ retrieves the current location $loc(i)$ of thread i and $val \in Val$ is a valuation. We denote by $State$ the set of all states of P.

Let $a \mathrel{\hat{=}} (l_s, S, l_t) \in A$ and $s \mathrel{\hat{=}} (loc, val) \in State$ be an action and state of P where $i \mathrel{\hat{=}} tid(a)$. The *denotation* of a, $[\![a]\!]$: $State \to \wp(State)$, defines the set of successor states from s according to the action a:

If $loc(i) \neq l_s$, then $[\![a]\!](s) \mathrel{\hat{=}} \varnothing$. Otherwise,$loc(i) = l_s$ and:

case S of

$$
\begin{array}{lcll}
x = e & \to & \{(loc(i) \mapsto l_t,\ val(x) \mapsto v) \mid v \in [\![e]\!](val)\} & \\
assume(c) & \to & \text{if } [\![c]\!](val) \text{ then } \{(loc(i) \mapsto l_t, val)\} & \text{else } \varnothing \\
assert(c) & \to & \text{if } [\![c]\!](val) \text{ then } \{(loc(i) \mapsto l_t, val)\} & \text{else } \bot \\
lock(x) & \to & \text{if } val(x) = 0 \text{ then } \{(loc(i) \mapsto l_t, val(x) \mapsto 1)\} & \text{else } \varnothing \\
unlock(x) & \to & \{(loc(i) \mapsto l_t, val(x) \mapsto 0)\} &
\end{array}
$$

where the denotation of expressions, $[\![e]\!]$: $Val \to \wp(\mathsf{Type})$, and boolean expressions, $[\![c]\!]$: $Val \to \mathbb{B}$, is standard.

We assume the existence of a error state for the assert statement denoted by $\bot$. We model the semantics of a concurrent program $P \mathrel{\hat{=}} (G_1, \ldots, G_n, \tilde{s})$ with the LTS

$M \hat{=} (State, Trans, \tilde{s})$ where $State$ is the set of states of P, $\tilde{s}$ is the *initial state* and the transition relation $Trans \subseteq State \times A \times State$ satisfies:

$(s, a, t) \in Trans$ iff $\exists$a $\in$ A. t $\in [\![$a$]\!]$(s)

We define the set of enabled transitions of a thread i at a state $s \in State$ as:

$$en_i(s) \hat{=} \{t \in Trans \mid t \in en(s) \text{ and } tid(act(t)) = i\}$$

The set of enabled transitions of an action a at a state $s \in State$ is defined as:

$$en_a(s) \hat{=} \{t \in Trans \mid t \in en(s) \text{ and } act(t) = a\}$$

When an action a is associated with a thread i, i.e. $tid(a) = i$, we have that for all $s \in State$, $en_a(s) \subseteq en_i(s)$.

Let P be a concurrent program and M its associated LTS.

We say that P is *fully-deterministic* if for all states at most one transition is enabled, i.e. $\forall s \in State. \mid en(s)| \leq 1$.

We say that P is *deterministic* if for all states at most one transition per thread is enabled, i.e. $\forall s \in State, i \in TId. \ |en_i(s)| \leq 1$. Otherwise, P is *non-deterministic*.

Furthermore, we distinguish between two types of *non-determinism*. Let $s \in State$ be a state where $|en_i(s)| > 1$ for some thread i. If $|en_a(s)| > 1$ for some action a of thread i, we say that P is *data non-deterministic*; if $|act(en_i(s))| > 1$, we say that P is *control non-deterministic*. Clearly, a program P can be *data* and *control non-deterministic*.

Chapter 3

Parametric Partial Order Semantics

In stateless model checking, dynamic PORs typically rely on Mazurkiewicz trace theory to compactly represent the interleaving space using independence of concurrent actions. Thus, from a semantics perspective, trace theory is a parametric partial order semantics, where independence is the parameter of the reduction.

The unfolding of a structure (e.g. a Petri net, labelled transition system, control flow graph, etc.) is a tree-like representation of its behavior typically known as the *computation tree*. In the case of Petri nets, the unfolding is a prime event structure (PES). Intuitively, a PES is a structure where common prefixes of its configurations (partial orders) are merged.

In this chapter, we present a parametric prime event structure semantics for multi-threaded programs that is equivalent to Mazurkiewicz trace theory. This is the first step towards a unification of the algorithmic benefits of DPORs and unfolding explorations under a common semantics. We now describe the key insights of our approach.

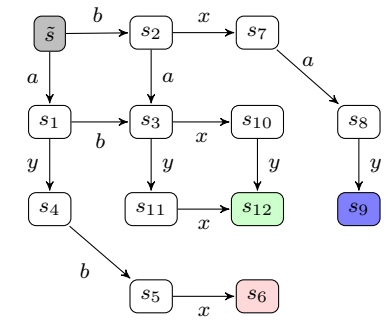
Consider the program in Figure 3.1. For simplicity, we do not represent the program as a set of CFGs since both threads are straight-line programs and the control locations do not play an important role in the discussion.

```
Thread A:          |  Thread B:
  a:=x+1;          |    b:=y+1;
  y:=a;            |    x:=b;
```

Figure 3.1: Multi-threaded program with $\tilde{s} \mathrel{\hat{=}} a, b, x, y \mapsto 0$.

The associated transition system with the program in Figure 3.1 is shown in the top section of Figure 3.2. Since all the statements write to a distinct variable, we abbreviate the statements to their associated written variable, e.g., $a := x + 1$ is represented as a. The interleaving space (set of executions of M) denoted $\mathit{Runs}(M)$

$M \mathrel{\hat{=}}$



$s_1 \mathrel{\hat{=}} a \mapsto 1; b, x, y \mapsto 0$

$s_2 \mathrel{\hat{=}} b \mapsto 1; a, x, y \mapsto 0$

$s_3 \mathrel{\hat{=}} a, b \mapsto 1; x, y \mapsto 0$

$s_4 \mathrel{\hat{=}} a, y \mapsto 1; b, x \mapsto 0$

$s_5 \mathrel{\hat{=}} a, y \mapsto 1; b \mapsto 2; x \mapsto 0$

$s_6 \mathrel{\hat{=}} a, y \mapsto 1; b, x \mapsto 2$

$s_7 \mathrel{\hat{=}} b, x \mapsto 1; a, y \mapsto 0$

$s_8 \mathrel{\hat{=}} a \mapsto 2; b, x \mapsto 1; y \mapsto 0$

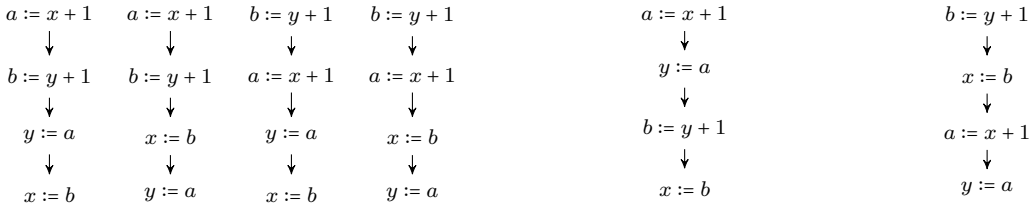
$s_9 \mathrel{\hat{=}} a, y \mapsto 2; b, x \mapsto 1$

$s_{10} \mathrel{\hat{=}} a, b, x \mapsto 1; y \mapsto 0$

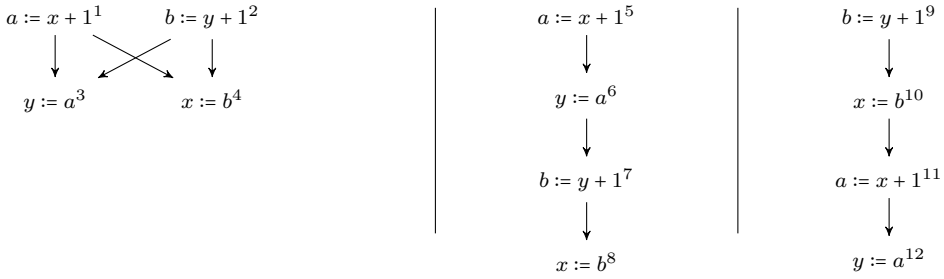
$s_{11} \mathrel{\hat{=}} a, b, y \mapsto 1; x \mapsto 0$

$s_{12} \mathrel{\hat{=}} a, b, x, y \mapsto 1$

$Runs(M) \mathrel{\hat{=}}$



$\mathcal{T}_\diamond \mathrel{\hat{=}}$



$\mathcal{U}_{M,\diamond} \mathrel{\hat{=}}$

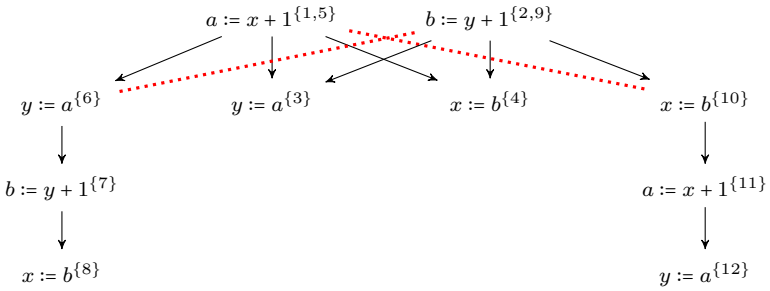


Figure 3.2: Transition system M of program in Figure 3.1 and three semantics: interleaving semantics $Runs(M)$, M-traces $\mathcal{T}_\diamond$ and the unfolding $\mathcal{U}_{M,\diamond}$.

is shown below the state graph. There are three deadlock states in the state graph: s_6, s_9 and s_{12}. From the state graph, it is clear that to reach state s_3 from the initial state $\tilde{s}$, the order of execution between the enabled transitions associated with the first action of each thread is irrelevant. When this situation occurs, we say that those transitions *commute* at $\tilde{s}$. If the transitions associated with two actions commute at every state, we say that the actions are independent. The set of actions with the independence relation forms a concurrent alphabet. In this example, we consider $a := x + 1 \diamond b := y + 1$ and $y := a \diamond x := b$.

In Figure 3.2, the first four executions are equivalent because we can transform one execution into another by swapping adjacent independent actions. By the properties of independence, every execution in this class will reach the same deadlock state s_{12}. The remaining two executions are each an equivalence class for the remaining two deadlock states. We now describe the process of merging this set of partial orders which are M-traces/dependence graphs into a particular PES that we refer to as the *unfolding*.

Each node in a dependence graph corresponds to an *event*. To make that explicit, we number the nodes of the dependence graphs. To each event e we associate the pair $(h(e), \lceil e \rceil)$ composed of the label and the causal history of e. For example, to the event number 1 we associate the pair $(a := x + 1, \varnothing)$ and to the event number 4 we have $(x := b, \{1, 2\})$. Using these pairs, we can merge common events across multiple partial orders. For example, events 1 and 5 are merged since their associated pair is the same. This procedure merges common prefixes of the partial orders and produces the last partial order in Figure 3.2. In order to *turn* this partial order into a PES we need to assign the conflict relation. For simplicity we only represent the immediate conflict denoted by the red dotted lines in Figure 3.2. Two events are conflicting when there is no partial order which contains both events. For example, consider the event number 6. There is no other event in any other partial order that can be merged with this event. The events compatible with 6 are the events $5, 7, 8$ which are part of the second partial order plus any event that can be merged with those events. In this case, we have that events 1 and 5 are merged, resulting in the set $\{1, 5, 7, 8\}$. Thus, in the PES representation, the event corresponding to the equivalence class for 6 is in immediate conflict with event 2 and 9. By hereditary conflict, it is also in conflict with $3, 4, 10, 11$ and 12. After applying this procedure to each event, we obtain the final PES of Figure 3.2.

The previous definition of the unfolding is not very useful from a practical perspective as it requires the entire set of partial orders to construct the final object. Once

we have the entire set of partial orders, we have already identified the set of deadlock states. Thus, our goal is to define the unfolding by extending the PES considered so far with a single event. As a result, we can potentially construct the unfolding as we are exploring each partial order. A key insight that we will show is that both causality and conflict are determined by the dependence relation. In particular, given an execution of the program the partial order is constructed with the dependence relation. For example the first execution presented in $Runs(M)$ of Figure 3.2 is converted to the first partial order considering that the first and the last two statements of each thread are independent. The conflict relation is also determined by the dependence relation by guaranteeing that events with dependent labels are not concurrent. Observe that in the PES of Figure 3.2, the immediate conflicts is between events labelled with dependent actions.

The main contributions of this chapter are:

- We define a new family of prime PES semantics parameterized by an independence relation and prove that it is equivalent to Mazurkiewicz trace theory (Section 3.3.1).
- We instantiate this domain of PES to multi-threaded programs and provide a fixpoint definition of the unfolding (Section 3.3.2).
- We present a formal relationship between the domain of unfoldings and the domain of independence relations (Section 3.4).

For the rest of this chapter, we consider the LTS $M \triangleq (State, Trans \subseteq State \times Act \times State, \tilde{s})$ associated with a *data-deterministic* concurrent program. Recall from Section 2.4 that a *data-deterministic program* satisfies that for all states, at most one transition per action is enabled.

3.1 The Independence Domain

In this chapter we use the independence relation based on [KP92, Def 2.2] and [God96, Def 3.1]. We start by recalling commutativity over actions.

Definition 12 (Commutativity of Actions)**.** *Given two actions $a, b \in Act$ and one state $s \in State$, we say that a, b* commute at *s iff*

- *if $a \in en(s)$ and $s \xrightarrow{a} t$, then $b \in en(s)$ iff $b \in en(t)$; and*
- *if $a, b \in en(s)$, then there is a state u such that $s \xrightarrow{ab} u$ and $s \xrightarrow{ba} u$.*

■

We define independence of concurrent actions based on commutativity.

Definition 13 (Independence of Concurrent Actions)**.** *An independence relation over the actions of M is any symmetric and irreflexive relation $\diamond \subseteq Act \times Act$ such that if $a \diamond b$, then a and b commute at* every *state $s \in Reach(M)$ and $tid(a) \neq tid(b)$. If a and b are not independent according to $\diamond$, then they are* dependent*, denoted by $a \diamondsuit b$.* ■

The independence relation in Definition 13 is also known as unconditional independence. Note that commutativity is a property of the system, i.e. there is only one commutativity relation, while there are multiple relations that can be considered independence relations according to Definition 13. For example, $\varnothing$ is always an independence relation.

Example 9. *Consider the program in Figure 3.1 extended with a third* `Thread C` *composed of the statement $x := 1$. Consider this statement and the statement $a := x+1$. At the initial state, $x, a \mapsto 0$, the statements do not commute since the final states of both orderings differ. Namely, $x := 1; a := x + 1(x, a \mapsto 0) = a \mapsto 2; x \mapsto 1$ and $a := x + 1; x := 1(x, a \mapsto 0) = a \mapsto 1; x \mapsto 1$. Hence, the statements do not commute at the initial state. The state after completely executing* `Thread B` *is $a, y \mapsto 0; b, x \mapsto 1$. At this state, the two statements commute because the assignment $x := 1$ does not modify the current state. By using independence relations, we are forced to declare these two statements dependent which limits the potential reduction of the state space.* ◁

A standard method to compute independence relations on the statements of a multi-threaded program is to consider the *read-write sets* of each statement. In particular, we associate for an action a a pair (R_a, W_a) of variables that are read and written by the action. Two actions a and b are dependent if $R_a \cap W_b$ or $W_a \cap W_b$ is not empty [God96]. Dynamic PORs use a more relaxed notion of independence where the read-write sets are dynamically generated. Thus, we can view this independence relation as being generated over a set of actions that is dynamically generated. This is particularly useful for statements that operate with arrays. For example, given the statements $a[i]$ and $a[j]$ a dynamically computed independence relation can consider both statements independence when $i \neq j$.

We denote by $\overline{\diamond}_M$ the set of independence relations of M which is naturally ordered by set-inclusion. The union and the intersection of a finite set of independence relations is an independence relation [Maz87]. Hence, $\langle \overline{\diamond}_M, \subseteq, \cup, \cap, \varnothing, \top^{\diamond} \rangle$ is a complete lattice where $\top^{\diamond}$ denotes the top element.

3.2 Mazurkiewicz Trace Semantics

Using the commutativity of concurrent actions, we define the following equivalence relation over the set of interleavings $Runs(M)$. Two permutation runs σ and σ' are *equivalent*, $\sigma \equiv \sigma'$, if they can be obtained by swapping transitions that commute. Formally, $\equiv$ is the transitive closure of the relation $\equiv^1$, where $\sigma \equiv^1 \sigma'$ iff there is $\sigma_1, \sigma_2 \in Act^*$ such that $\sigma = \sigma_1 ab \sigma_2$, $\sigma' = \sigma_1 ba \sigma_2$, $\tilde{s} \xrightarrow{\sigma_1} s$, and a, b commute at s. However, Mazurkiewicz trace theory requires a binary independence relation $\diamond$.

Similarly, independence of concurrent actions identifies an equivalence relation $\equiv_\diamond$ on the set $Runs(M)$. Formally, $\equiv_\diamond$ is the transitive closure of the relation $\equiv^1_\diamond$ where $\sigma \equiv^1_\diamond \sigma'$ iff there is $\sigma_1, \sigma_2 \in Act^*$ such that $\sigma = \sigma_1 ab \sigma_2$, $\sigma' = \sigma_1 ba \sigma_2$, and $a \diamond b$. From the properties of $\diamond$, one can immediately see that $\equiv_\diamond$ refines $\equiv$, i.e. if $\sigma \equiv_\diamond \sigma'$, then $\sigma \equiv \sigma'$.

The *Mazurkiewicz trace*, $\mathcal{T}_{\diamond,\sigma}$, of a run $\sigma \in Runs(M)$ is the set of runs equivalent to σ according to $\diamond$. Recall from Section 2.3.3 that given an independence relation $\diamond$ for every two runs $\sigma, \sigma' \in Runs(M)$:

$$\sigma' \equiv_\diamond \sigma \quad \text{iff} \quad \sigma' \in \mathcal{T}_{\diamond,\sigma} \quad \text{iff} \quad \sigma' \text{ is a linearization of } \mathcal{G}_{\diamond,\sigma}$$

In other words, the linearizations of the dependency graph $\mathcal{G}_{\diamond,\sigma}$ are *exactly* the executions in $\mathcal{T}_{\diamond,\sigma}$, and the collection of dependency graphs associated to (the runs of) M defines a partition of the set $Runs(M)$. Furthermore, all executions in the same equivalence class reach the same global state.

It is important to note that Mazurkiewicz trace semantics is a very general model since the independence relations used in the model can be parametric with respect to the choice of event labels. For example, consider the language $\{abc, cba\}$. If we assume that events are labelled by a single element of the set $\{a, b, c\}$, then it is clear that the action a is not independent from the action c resulting in set of equivalence classes $\{\{abc\}, \{cba\}\}$. However, by relaxing the naming of events, we can consider that the first and third events is either action a or c leading to a singleton equivalence class $\{\{abc, cba\}\}$. In this dissertation, we consider a simple correspondence between actions and labels of events, but in theory, there is no obstacle to consider independence relations parametric on the naming of events.

3.3 Prime Event Structure Semantics

In this section we define a labelled prime event structure $\mathcal{U}_{M,\diamond}$ where $\diamond$ is an input independence relation over the actions of M. This PES, known as the *unfolding*,

satisfies the following key property:

> Each configuration of $\mathcal{U}_{M,\diamond}$ corresponds to one dependence graph $\mathcal{G}_{\diamond,\sigma}$, for some $\sigma \in \mathit{Runs}(M)$.

Our key insight is to obtain a semantics equivalent to Mazurkiewicz traces by considering *the unfolding as a set of partial orders with conflict* [Eng91] where the conflict relation is defined by the dependence relation.

For the rest of this section, let $\diamond \in \overline{\diamond}_M$ be an independence relation of M. Recall the definitions from Section 2.3.2.

3.3.1 Prime Event Structures with Static Conflict

We now introduce a subclass of prime event structures where conflicts are determined by some symmetric binary relation on the event labels.

Consider a PES $\mathcal{E} \hat{=} (\mathit{Event}, <, \#, h)$. Recall the definition of immediate conflict $\#^i \hat{=} \{(e, e') \in \# \mid \lceil e \rceil \cup \lceil e' \rceil \in \mathit{conf}(\mathcal{E}) \text{ and } [e] \cup \lceil e' \rceil \in \mathit{conf}(\mathcal{E})\}$.

Since the events of $\mathcal{E}$ have a finite number of causes, the conflict between two events e, e' can be pinpointed to the two events in their past that are in immediate conflict [1]. The immediate conflict relation represents the source of non-determinism in the structure and can be seen as a generator for the conflict relation. We now show that the set of configurations of $\mathcal{E}$ can be defined with the structure $\mathcal{E}^i \hat{=} (\mathit{Event}, <, \#^i, h)$ that only considers the immediate conflict proving that $\#$ is the closure of $\#^i$ with respect to the axiom of hereditary conflict.

We start by showing that every conflict is associated to an immediate conflict (Lemma 1) and that if a set of events is not a configuration then it is because it is not causally closed or there is an immediate conflict between two events (Lemma 2).

Lemma 1. *Let $\mathcal{E} \hat{=} (\mathit{Event}, <, \#, h)$ be a PES and $C \subseteq E$ be a set of events that is causally closed. There exists $e, e' \in C$ such that $e \# e'$ iff exists $f, f' \in C$ such that $f \#^i f'$.*

Proof.

Let $\# e_{[f]} \hat{=} \{f' \in [f] \mid e \# f'\}$ be the set of conflicts of event e in the local configuration of event f.

[1] The event structures whose conflict relation is characterized by immediate conflict are know as *well branching event structures* [MT92].

($\Rightarrow$) Assume $e, e' \in C$ such that $e \# e'$. It is clear that $e' \in \# e_{[e']}$ and that $e \in \# e'_{[e]}$. Let $f' \in \min_{\#e_{[e']}}$ be a minimal event w.r.t. causality in the past of e' such that $e \# f'$. It follows that for all $f \in [e]$, $[f] \cup \lceil f' \rceil \subseteq C$ is contained in C (as it is causally closed) and that it is a configuration of $\mathcal{E}$. Let $f \in \min_{\#f'_{[e]}}$ be a minimal event w.r.t. causality in the past of e such that $f' \# f$. It follows that $[f'] \cup \lceil f \rceil$ is a configuration. Thus, $f \#^i f'$.

($\Leftarrow$) Assume $f, f' \in C$ such that $f \#^i f'$. By definition of $\#^i$ it follows that $f \# f'$. □

Lemma 2. *Let $\mathcal{E} \hat{=} (Event, <, \#, h)$ be a* PES *and $C \subseteq E$ be a set of events. If $C \notin conf(\mathcal{E})$ then:*

- *C is not causally closed or* *(1)*
- *exists $e, e' \in C$, such that $e \#^i e'$* *(2).*

Proof. Assume $C \notin conf(\mathcal{E})$. By Definition 8, either C is not causally closed or exists $e, e' \in C$, such that $e \# e'$. If C is not causally closed, by Equation (3.1) the lemma is trivially true. If exists $e, e' \in C$, such that $e \# e'$ then either C is not causally closed or two events in C are in immediate conflict. Assume that C is causally closed. Thus, if exists $e, e' \in C$, such that $e \# e'$ then two events in C are in immediate conflict. This holds by Lemma 1. □

Theorem 1 shows an alternative definition of configuration of a PES using the immediate conflict relation. Thus, we can represent a PES with the immediate conflict relation without loss of information.

Theorem 1. *Let $\mathcal{E} \hat{=} (Event, <, \#, h)$ be a* PES*. A set of events $C \subseteq E$ is a configuration, $C \in conf(\mathcal{E})$, iff:*

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

Proof.

($\Rightarrow$) Assume $C \in conf(\mathcal{E})$. Both conditions are trivially true as C is causally closed and $\#^i \subseteq \#$.

($\Leftarrow$) By contradiction. Assume C satisfies Equations (3.3) and (3.4) and that $C \notin conf(\mathcal{E})$. By Lemma 2, either C is not causally closed or there are two events in immediate conflict leading to a contradiction. □

We now define a subclass of PES where the immediate conflict relation is determined by a relation over the labels.

Definition 14 (R-Branching Prime Event Structure)**.** *Let $R \subseteq A \times A$ be a symmetric binary relation. A A-PES $\mathcal{E} \hat{=} (E, <, \#, h)$ is R-branching iff two events $e, e' \in E$ are in immediate conflict ($e \#^i e'$) iff:*

1. *$e \not\leqslant e'$,*
2. *$(h(e), h(e')) \in R$ and*
3. *For every $(\bar{e}, \bar{e'}) \in [e] \times [e'] \setminus \{(e, e')\}$, it holds that $(h(\bar{e}), h(\bar{e'})) \notin R$.*

■

A R-branching PES $(E, <, \#, h)$ is such that two events e, e' are in immediate conflict iff:

1. e and e' are not causally related
2. the labels of e and e' are related by R and
3. there is no other pair of events in the local configurations of e,e' besides (e, e') whose labels are related by R.

Lemma 3 shows that the immediate conflict relation of a R-branching PES defined in Definition 14 is well formed.

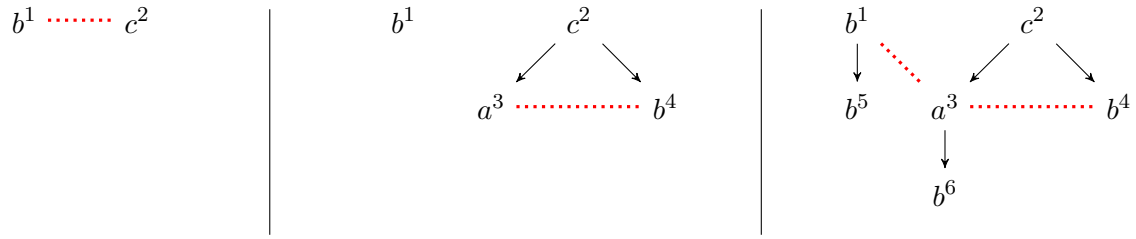
Lemma 3. *Let $\mathcal{E} \hat{=} (Event, <, \#, h)$ be a R-branching PES. For all $e, e', e'' \in Event$, if $(e, e') \in \#^i$ and $e < e''$, then $(e'', e') \notin \#^i$.*

Proof. By contradiction. Let $e, e', e'' \in Event$. Assume that (1) $(e, e') \in \#^i$, (2) $e < e''$ and (3) $(e'', e') \in \#^i$.

From (1) and Definition 14 it follows that $(h(e), h(e')) \in R$. From (2),(3) and Definition 14, it follows that $(e, e') \in [e''] \times [e'] \setminus \{(e'', e')\}$ and that $(h(e), h(e')) \notin R$. Thus, we reach a contradiction as $(h(e), h(e')) \notin R$ and $(h(e), h(e')) \in R$. □

Note that $\not\leqslant$ is irreflexive which implies that the conflict relation is also irreflexive.

Example 10. *Consider the set of actions $Act \hat{=} \{a, b, c\}$, the symmetry binary relation $R \hat{=} \{(a, b), (b, a)\}$ and the following prime event structures:*



The left one is not R-branching since $(b, c) \notin R$ but the events 1 and 2 are in immediate conflict. The middle PES *is also not R-branching since events 1 and 3 are concurrent but their labels are related by R, $(a, b) \in R$. Note that this constraint is satisfied for events 3 and 4 whose labels are a and b, respectively. The right* PES *is R-branching as any pair of events whose labels are related by R and are not causally related are in immediate conflict. Events 3 and 5 are not in immediate conflict even though they are not causally related and their labels are related by R because 1 and 3 are already in immediate conflict. Also, events 3 and 6 are not in immediate conflict since they are causally related. Observe that since R is irreflexive, it is possible for two events with the same label to be concurrent. For example, this is the case for events 1 and 4 in the middle and right prime event structures.* ◁

As the Example 10 demonstrates, the conflict relation in a R-branching PES is completely determined by a relation on the labels. We denote this subclass by $(Event, <, R, h)$ where R is a symmetric relation over the event labels. Hence, the name *prime event structures with static conflict*.

Given a PES without the conflict relation (imagine the structures in Example 10 without conflicts) we obtain a unique R-branching structure based on the labels and the relation on the labels. Clearly, not all prime event structures are R-branching since once we observe two events in immediate conflict, it means that any other events with the same labels cannot be concurrent. The middle PES in Example 10 is an example of a prime event structure whose conflict relation is not based on a relation over labels.

We now define deterministic R-branching PES, another subclass of prime event structures, that we will use to represent the interleaving semantics of a program via the set of M-traces. Recall from Section 2.3.2 that a PES $\mathcal{E}$ is *deterministic* iff for every configuration $C \in conf(\mathcal{E})$, if $e, e' \in en(C)$ and $h(e) = h(e')$, then $e = e'$.

The following lemma shows that in a deterministic PES any two events with the same label are not in immediate conflict nor concurrent [2].

[2]Deterministic PES in [SNW96] are known as *deterministically labelled* PES in [RT91].

Lemma 4. *A R-branching* PES $\mathcal{E} \hat{=} (Event, <, R, h)$ *is deterministic iff for all* $e, e' \in Event$*, if* $h(e) = h(e')$*, then* $\neg(e \#^i e')$ *and* $\neg(e \parallel e')$*.*

Proof. We will show that (for all $C \in conf(\mathcal{E})$, if $e, e' \in en(C)$ and $h(e) = h(e')$, then $e = e'$) iff (for all $e, e' \in Event$, if $h(e) = h(e')$, then $\neg(e \#^i e')$ and $\neg(e \parallel e')$).

($\Rightarrow$) By contradiction. Assume that for all $C \in conf(\mathcal{E})$, if $e, e' \in en(C)$ and $h(e) = h(e')$, then $e = e'$. Furthermore, assume that there is a pair of events $e, e' \in Event$ such that $h(e) = h(e')$ and ((1) $e \#^i e'$ or (2) $e \parallel e'$). The proof follows by cases:

1. It holds that $e \#^i e'$. By definition of $\#^i$ there is a configuration $C \hat{=} \lceil e \rceil \cup \lceil e' \rceil$ where $e, e' \in en(C)$. Thus we reach a contradiction since by the first assumption $e = e'$ but the relation $\#^i$ is irreflexive.

2. It holds that $e \parallel e'$. As previously, there is a configuration $C \hat{=} \lceil e \rceil \cup \lceil e' \rceil$ where $e, e' \in en(C)$. Thus we reach a contradiction since by the first assumption $e = e'$ but the relation $\parallel$ is irreflexive.

($\Leftarrow$) By contradiction. Assume that for all $e, e' \in Event$, if $h(e) = h(e')$, then $\neg(e \#^i e')$ and $\neg(e \parallel e')$. Let $C \in conf(\mathcal{E})$ and $e, e' \in en(C)$. Assume that $h(e) = h(e')$ and $e \neq e'$. By the first assumption it follows that $\neg(e \#^i e')$ and $\neg(e \parallel e')$. Since $e, e' \in en(C)$, it follows that $\lceil e \rceil \cup \lceil e' \rceil$ is a configuration C' and that $C \subseteq C'$. Since $e \neq e'$, $Event \times Event$ is partitioned by $(<, >, id, \#, \parallel)$ and by the definition of $\#^i$, either $e \#^i e'$ or $e \parallel e'$. Thus, we reach a contradiction.

□

Example 11. *Consider the R-branching* PES *in Example 10. This* PES *is not deterministic as events* 1 *and* 4 *have the same label and concurrent. Note that even if R is reflexive which would imply that those events would be in immediate conflict, the resulting* PES *is still not deterministic.* ◁

Recall Definition 11 of morphism between prime event structures. Clearly, this definition is also applicable to prime event structures with static conflict. Let $\mathcal{E} \hat{=} (Event, <, R, h)$ be a R-branching PES. We now define the name version of $\mathcal{E}$ where events are uniquely identified by their labels and their causes.

Definition 15 (Named prime event structure)**.** *Let* $\mathcal{E} \hat{=} (E_1, <_1, R, h_1)$ *be a R-branching* PES*. The named version of* $\mathcal{E}$ *is the R-branching* PES $n(\mathcal{E}) \hat{=} (\eta(E_1), <_2, R, h_2)$ *where:*

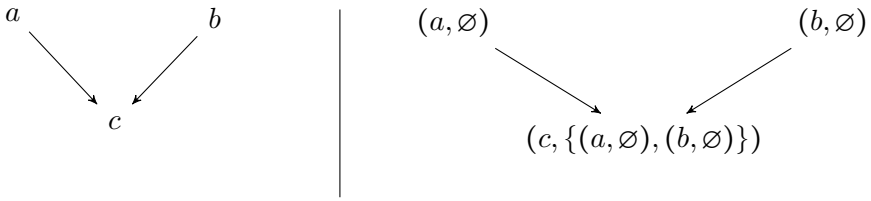
- $\eta(e) \hat{=} (h_1(e), \lceil e \rceil)$*;*
- $\eta(e) <_2 \eta(e')$ *iff* $e <_1 e'$ *and*

- $h_2(\eta(e)) \hat{=} h_1(e)$.

■

The structure $n(\mathcal{E})$ is a PES since: 1) the conflict relation is irreflexive as it is generated by the immediate conflict of Definition 14 and 2) the causality relation is a strict partial order as the renaming of events respects causality.

Example 12 (Naming Events)**.** *Consider the following* PES *on the left and its named version on the right:*



The minimal events in a named PES *are those with no causes,* $(a, \varnothing)$ *and* $(b, \varnothing)$*. Since the event labelled* c *depends on both, it has the name:* $(c, \{(a, \varnothing), (b, \varnothing)\})$*.* ◁

The next lemma shows that the *naming* of an event e as the pair $(h(e), \lceil e \rceil)$ is a morphism between R-branching prime event structures when R is a reflexive relation.

Lemma 5. *The pair* $n \hat{=} (\eta, id)$ *is a morphism between* $\mathcal{E} \hat{=} (E_1, <_1, R, h_1)$ *and its named version* $n(\mathcal{E})$ *if* R *is reflexive.*

Proof. We prove that n is a PES morphism by cases (note that for all $e \in E_1, \eta \downarrow e$):

1. We show that for all $e \in E_1$, $[\eta(e)] \subseteq \eta([e])$. Let $e \in E_1$ be an event. By definition of local configuration it follows that $[\eta(e)] = \{\eta(e') \mid \eta(e') <_2 \eta(e)\}$ and that $\eta([e]) = \{\eta(e) \mid e' <_1 e\}$. Since $\eta(e') <_2 \eta(e)$ iff $\{\eta(e) \mid e' <_1 e\}$, we have that $[\eta(e)] = \eta([e])$.

2. We show that for all $e, e' \in E_1$, if $\eta(e) \between_2 \eta(e')$, then $e \between_1 e'$. Proof by contradiction. Let $e, e' \in E_1$. Assume that $\eta(e) \between_2 \eta(e')$ and $\neg(e \between_1 e')$. It follows that $e \neq e'$ and $\neg(e \#_1 e')$. Since $\neg(e \#_1 e')$ it follows that: (1) $e \parallel e'$ or (2) ($e \in [e']$ or $e' \in [e]$). The proof continues by cases:

   (a) If $e \neq e'$ and $e \parallel e'$, it follows from Definition 14 that $(h_1(e), h_1(e')) \notin R$. Since R is reflexive, it follows that $h_1(e) \neq h_1(e')$ and $\eta(e) \neq \eta(e')$. Thus $\neg(\eta(e) \between_2 \eta(e'))$ and we reach a contradiction.

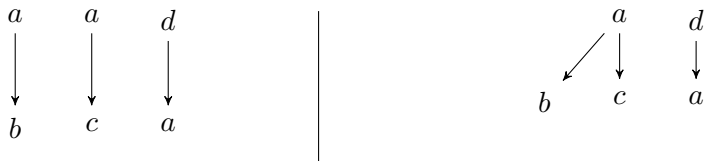
(b) If e and e' are causally related, then $\eta(e) <_2 \eta(e')$ or $\eta(e) <_1 \eta(e')$. Thus $\neg(\eta(e) \between_2 \eta(e'))$ and we reach a contradiction.

3. $id \circ h_0 = h_1 \circ \eta$. Follows directly from the definition.

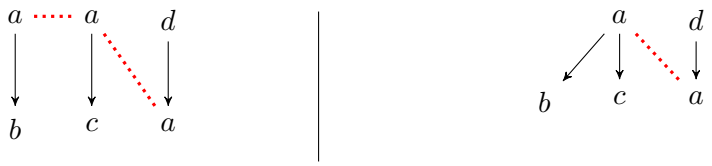
□

For the rest of this section, we assume that R is reflexive in the context of R-branching prime event structures.

Example 13 (Abstraction by Naming)**.** *In this example, we show a counter-example of when the naming operation is not a morphism if R is irreflexive and also exemplify the abstraction that occurs with the naming. Intuitively, the abstraction corresponds to merging events based on their histories and labels which is exactly how we merged events from multiple partial orders in Figure 3.2. Consider the following prime event structures:*



Assume that they are R-branching structures with $R \mathrel{\hat{=}} \varnothing$. Consider the PES *on the left. It contains two events labelled with a with no causes. These two events will be* merged *into a single event labelled by a in the named structure of this* PES *given on the right. Note that the event labelled with a after d is not merged with the minimal event labelled by a as they have a different set of causes. This merge operation is not an abstraction as the named version has fewer configurations than the original structure. For example, the word $aabcda$ is represented in the first but not the second. Consider now the structures where R is reflexive:*



The abstraction by merging events as defined in Definition 15 corresponds to removing the conflict between the two minimal events labelled by a by merging those events. It is clear that this is an abstraction, e.g. the word $abcd$ is represented in the second but not the first. ◁

Although this morphism seems to be a determinisation procedure, it is not true that $n(\mathcal{E})$ is a deterministic PES. For example, the second named event structure given in Example 13 has two events with the same label in immediate conflict. The following theorem shows that for deterministic R-branching prime event structures where R is reflexive, the named version is a complete abstraction. That is, both the original and the named structures represent exactly the same set of configurations.

Theorem 2. *Let $\mathcal{E} \hat{=} (Event, <, h)$ be a R-branching* PES *where R is reflexive. If $\mathcal{E}$ is deterministic, then for all $C \subseteq Event$, $C \in conf(\mathcal{E})$ iff $\eta(C) \in conf(n(\mathcal{E}))$.*

Proof. Assume that $\mathcal{E}$ is a deterministic R-branching PES. We will show that $C \in conf(\mathcal{E})$ iff $\eta(C) \in conf(n(\mathcal{E}))$.

($\Rightarrow$) Assume that $C \in conf(\mathcal{E})$. It follows that (1) C is causally closed and (2) for all $e, e' \in C, \neg(e \#^i e')$. From Lemma 5, we have that $\eta(C)$ is causally closed. We now show by contradiction that for all $e, e' \in C$, if $\neg(e \#^i e')$, then $\neg(\eta(e) \#^i \eta(e'))$. Let $e, e' \in C$ such that $\neg(e \#^i e')$ and $\eta(e) \#^i \eta(e')$. Since $\eta(e) \#^i \eta(e')$, it follows that $h(\eta(e), \eta(e')) \in R$. Note that $e, e' \in en(\lceil e \rceil \cup \lceil e' \rceil)$. Since $h(\eta(e)) = h(e)$ and $h(\eta(e')) = h(e')$ we reach the contradiction $h(e, e') \in R$ and $h(e, e') \notin R$.

($\Leftarrow$) Assume that $\eta(C) \in conf(n(\mathcal{E}))$. It follows that (1) $\eta(C)$ is causally closed and (2) for all $\eta(e), \eta(e') \in \eta(C), \neg(\eta(e) \#^i \eta(e'))$. From Lemma 5, we have that C is causally closed. We now show that for all $\eta(e), \eta(e') \in \eta(C)$, if $\neg(\eta(e) \#^i \eta(e')$, then $\neg(e \#^i e')$. Let $\eta(e), \eta(e') \in \eta(C)$. Assume that $\neg(\eta(e) \#^i \eta(e')$. It follows from the definition of $\#^i$ that: (1) $\eta(e) \lesseqgtr \eta(e')$ or (2) $h(\eta(e), \eta(e')) \notin R$ or (3) exists $\eta(\bar{e}) \in [\eta(e)], \eta(\bar{e'}) \in [\eta(e')]$ such that, $\eta(\bar{e}) \neq \eta(e)$ and $\eta(\bar{e'}) \neq \eta(e')$ and $h(\eta(\bar{e}), \eta(\bar{e'})) \in R$. The proof continues by cases:

1. It follows from $\eta(e) \lesseqgtr \eta(e')$ that: (1) $\eta(e) = \eta(e')$ or (2) $(\eta(e) < \eta(e')$ or $\eta(e') < \eta(e))$. It follows from (1) that $h(\eta(e)) = h(\eta(e'))$. It follows from $h(\eta(e)) = h(\eta(e')) = h(e) = h(e')$ and from Lemma 4 that $\neg(e \#^i e')$. It follows from (2) and by Lemma 5 that $e < e'$ or $e' < e$. Thus, $\neg(e \#^i e')$.

2. It follows from $h(\eta(e), \eta(e')) \notin R$, $h(\eta(e)) = h(e)$ and $h(\eta(e')) = h(e')$ that $h(e, e') \notin R$. Thus, $\neg(e \#^i e')$.

3. It follows from (3) and Lemma 5 that exists a $\bar{e} \in [e], \bar{e'} \in [e']$ such that, $\bar{e} \neq e$ and $\bar{e'} \neq e'$ and $h(\bar{e}, \bar{e'}) \in R$. Thus, $\neg(e \#^i e')$.

□

Thus, we can *represent* a deterministic R-branching PES via its named version.

The following Lemma shows how to define a PES from a set of events in a named R-branching event structure where both structures are isomorphic. In particular, the set of named events of the obtained PES is the starting set of named events.

Lemma 6. *Let $E \hat{=} \{e_1, e_2, \ldots, e_n\}$ be the set of named events of a named R-branching* PES. *The structure $[\![E]\!] \hat{=} (\{1, \ldots, n\}, <, R, h)$ defined as:*

- $h(i) \hat{=} \pi_1(e_i)$;
- $< \hat{=} \{(i, j) \mid e_i \in \pi_2(e_j)\}$

is a R-branching PES *and satisfies $n([\![E]\!]) = E$.*

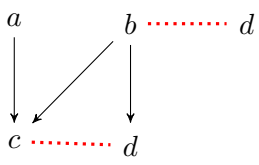
Proof.

($\mathcal{E}$ is R-branching). Since the events are from a well-formed R-branching named PES, it follows that $e_i = (a, H)$ where $a = h(e_i)$ and $\lceil e_i \rceil = H$. Thus, by definition, the causality relation is both PESs is equivalent. Since the causality is equivalent and the original PES is R-branching, it follows that $\mathcal{E}$ will necessarily be R-branching.

($n(\mathcal{E}) = E$). Directly holds from the observation that the causality relations are equivalent and by definition of h. □

As a consequence of Theorem 2 and Lemma 6, a deterministic R-branching PES $\mathcal{E} \hat{=} (Event, <, R, h)$ is equivalent to the tuple (R, E) where E is the set of named events of $n(\mathcal{E})$.

Example 14. *Consider the following deterministic R-branching* PES *where R is the symmetric reflexive closure of $\{(b, c), (b, d)\}$:*



$e_1 \hat{=} (a, \varnothing)$
$e_2 \hat{=} (b, \varnothing)$
$e_3 \hat{=} (d, \varnothing)$
$e_4 \hat{=} (c, \{e_1, e_2\})$
$e_5 \hat{=} (d, \{e_2\})$

The set E composed of five named events of this PES *is given on the right of the structure. The structure $[\![E]\!] \hat{=} (\{1, 2, 3, 4, 5\}, <, R, h)$ where events $1, 2$ and 3 are minimal and $1 < 4$, $2 < 4$ and $2 < 5$. Using the relation R to assign immediate conflicts, namely, $2 \#^i 3$ and $4 \#^i 5$, we obtain an isomorphic* PES *to the one above.* ◁

Let $R \subseteq A \times A$ be a reflexive symmetric binary relation over a set of actions A. The following Lemma shows that prefix-order on named deterministic R-branching prime event structures is simply subset ordering over the set of named events.

Lemma 7. *Let* $\mathcal{E}_1 \hat{=} (E_1, <_1, R, h_1)$ *and* $\mathcal{E}_2 \hat{=} (E_2, <_2, R, h_2)$ *be two named deterministic* R*-branching prime event structures. It holds that* $\mathcal{E}_1 \trianglelefteq \mathcal{E}_2$ *iff* $E_1 \subseteq E_2$.

Proof. It follows directly from Definition 9 and Lemma 5. □

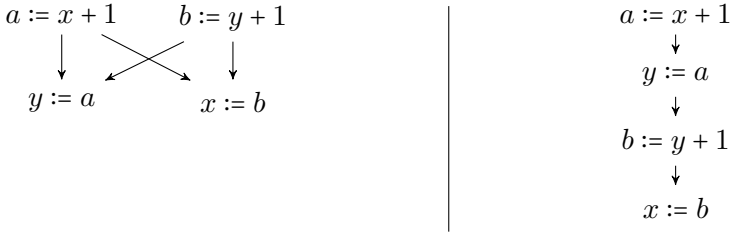
We denote by *Event* the set of named events that are possible in deterministic R-branching prime event structures. By Lemma 6, we can associate to each set of named events, a deterministic R-branching structure.

We define the lattice of deterministic R-branching PESs as $(\mathrm{dPES}_R, \trianglelefteq, \sqcup, \sqcap, \mathcal{E}^{\perp}, \top)$ where:

- $\sqcup(X) \hat{=} [\![\bigcup \{E \mid (E, <, \#, h) \in X\}]\!]$;
- $\sqcap(X) \hat{=} [\![\bigcap \{E \mid (E, <, \#, h) \in X\}]\!]$;
- $\mathcal{E}^{\perp} \hat{=} (\varnothing, \varnothing, \varnothing, \perp)$
- $\mathcal{E}^{\top} \hat{=} [\![Event]\!]$.

This lattice is isomorphic to $(\wp(Event), \subseteq, \cup, \cap, \varnothing, Event)$.

Example 15. *We can connect the results of this section to our initial example in Figure 3.2. Consider the first two M-traces in the figure represented as two prime event structures* [3]*:*



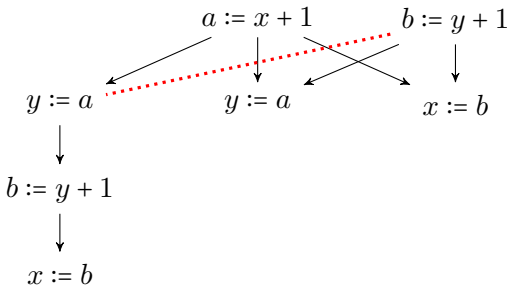
The sets of named events in the two structures are:

$$e_{11} \hat{=} (a := x + 1, \varnothing) \qquad e_{21} \hat{=} (a := x + 1, \varnothing)$$
$$e_{12} \hat{=} (b := y + 1, \varnothing) \qquad e_{22} \hat{=} (y := a, \{e_{21}\})$$
$$e_{13} \hat{=} (y := a, \{e_{11}, e_{12}\}) \qquad e_{23} \hat{=} (b := y + 1, \{e_{22}\})$$
$$e_{14} \hat{=} (x := b, \{e_{11}, e_{12}\}) \qquad e_{24} \hat{=} (x := b, \{e_{23}\})$$

Note that R *is a dependence relation defined as the reflexive symmetric closure of* $\{(a := x + 1, x := b), (b := y + 1, y := a), (b := y + 1, x := b), (a := x + 1, y := a)\}$. *Also, these*

[3] Since there is a bijection between M-traces and partial orders, a M-trace can be represented by a PES with the empty conflict relation. These prime event structures are known as *elementary prime event structures*.

two M-traces are named deterministic R-branching prime event structures. The result deterministic R-branching PES *from their join is:*



This PES *corresponds to building the structure from the union of their named events using Lemma 6. In particular, the union of both sets only merges the minimal events labelled by $a := x + 1$. The dependence relation sets the first events that are not part of the same partial order in immediate conflict.* ◁

3.3.2 Trace Semantics with Prime Event Structures

In this section we present a fixpoint definition of the unfolding that is parameterized by an independence relation from Section 3.1. The unfolding is a particular named deterministic $\diamond$-branching PES that is equivalent to the set of M-traces defined by the same dependence relation. That is, it will be a deterministic PES, as Mazurkiewicz trace theory is a linear model, where the conflict relation is static and determined by the dependence relation. As described in Section 3.3.1, equipping events with names is necessary to identify common events in the partial orders/dependence graphs. Since every configuration C of the unfolding corresponds to a M-trace, the unfolding is a compact representation of the interleaving semantics $\mathit{inter}(C)$.

Note that for an arbitrary PES, $\mathit{inter}(C)$ may contain sequences that are not executions of M. However, the definition of the unfolding ensures that $\mathit{inter}(C) \subseteq \mathit{Runs}(M)$. Also, since all sequences in $\mathit{inter}(C)$ belong to the same M-trace, all will reach the same state. Thus, we define the state of a configuration C as:

$$\mathit{st}(C) \mathrel{\hat{=}} \{s \in \mathit{st}(\sigma) \mid \sigma \in \mathit{inter}(C)\}$$

We will show that for all configurations C of the unfolding, $\mathit{st}(C)$ is a singleton. Since the program is data deterministic, it holds that $\mathit{st}(\sigma) = \{s\}$ for every run $\sigma \in \mathit{Runs}(M)$. For the rest of this section, assume a dependence relation $\diamond$ of M. We denote by $\mathcal{E}_\diamond$ the lattice of deterministic $\diamond$-branching prime event structures.

Given the current *version* of the unfolding $\mathcal{E} \hat{=} (Event, <, \otimes, h)$, we will compute the extension of the unfolding with an event named by an action a and some history $H \in conf(\mathcal{E})$. The set $\mathcal{H}_{\mathcal{E},a}$ of candidate *histories* for an action a in a PES $\mathcal{E}$ will contain all configurations H of $\mathcal{E}$ such that:

- the action a is enabled at $st(H)$, and
- for all events $e \in \max_H$, we have that $h(e) \otimes a$.

The enabling transformer, $en_{\rightarrow} :: \mathcal{E}_{\otimes} \rightarrow \wp(Event)$ is defined as:

$$en_{\rightarrow}(\mathcal{E} \hat{=} (E, <, h)) \hat{=} \{(a, H) \mid H \in \mathcal{H}_{\mathcal{E},a} \text{ and } a \in Act\} \setminus E \tag{3.5}$$

Once the event $e \hat{=} \langle a, H \rangle$ has been inserted into the unfolding, its associated action $h(e) = a$ may be dependent with $h(e')$ for some e' in the current prefix that is outside the history of e. Since their labels are dependent, in order to obtain a bijection between configurations and dependence graphs, we prevent their occurrence within a same configuration by introducing a conflict between e and e'. This is achieved by construction as $\mathcal{E}$ is $\otimes$-branching.

Definition 16 (Extension)**.** *The extension transformer, $ext_{\rightarrow} : \mathcal{E}_{\otimes} \rightarrow \mathcal{E}_{\otimes}$ extends a well-formed* PES *$\mathcal{E}$ with the events in $\mathcal{H}_{\mathcal{E},a}$ for some action a:*

$$ext_{\rightarrow}(\mathcal{E} \hat{=} (E, <, h)) \hat{=} [\![E \cup \{e \mid \ en_{\rightarrow}(\mathcal{E})\}]\!]$$

For simplicity, we assume that $ext_{\rightarrow}$ extends the current PES *with a single event if $en_{\rightarrow}$ returns a non-empty set.* ■

We now show that the extension transformer produces a deterministic $\otimes$-branching PES.

Lemma 8. *If $\mathcal{E}$ be a deterministic $\otimes$-branching* PES*, then $ext_{\rightarrow}(\mathcal{E})$ is a deterministic $\otimes$-branching* PES*.*

Proof. Follows directly from the definition of $en_{\rightarrow}$ that any event added to the current prefix is of the form $(a \in Act, C \in conf(\mathcal{E}))$. In particular, the causality relation is acyclic as every event introduced by $en_{\rightarrow}$ is a causal successor of only events that were already present in E. Furthermore, the insertion of an event does not change the causal relations already determined by E. The relation $<$ is transitive as the history of a configuration is causally closed. □

We now show that the extension transformer is monotone.

Lemma 9. *Let $\mathcal{E}_1, \mathcal{E}_2$ be deterministic $\otimes$-branching prime event structures. If $\mathcal{E}_1 \trianglelefteq \mathcal{E}_2$, then $ext_{\rightarrow}(\mathcal{E}_1) \trianglelefteq ext_{\rightarrow}(\mathcal{E}_2)$.*

Proof. Follows directly from the definition of $ext_{\rightarrow}$ and Lemma 7. □

We call each such deterministic $\otimes$-branching PES an unfolding prefix. Note that $\mathcal{E}^{\perp} \hat{=} [\![\varnothing]\!]$ is the bottom element of the lattice $\mathcal{E}_{\otimes}$. Each unfolding prefix intuitively represents the prefixes of dependency graphs associated to executions of M. One can view the construction of the unfolding from bottom at the start of the execution and then iteratively extending it for any transition until no more events can be added to the current unfolding prefix.

We now define *the* unfolding of the system. Since the lattice of unfolding prefixes is a complete lattice, by the Knaster-Tarski theorem, the set of fixed points of $ext_{\rightarrow}$ is also a complete lattice as $ext_{\rightarrow}$ is monotone. Furthermore, from the Kleene fixed-point theorem, we can obtain the least fixed point of $ext_{\rightarrow}$ by iteratively applying $ext_{\rightarrow}$ to the empty prefix until fixed point.

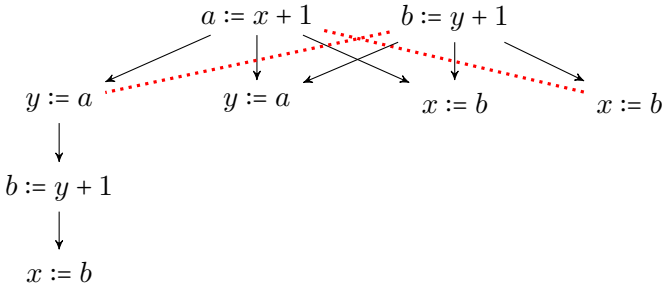
Definition 17 (Unfolding)**.** *The* unfolding $\mathcal{U}_{M,\diamond} \hat{=} \mathsf{lfp}\ \mathcal{E}.\ \mathcal{E}^{\perp} \sqcup ext_{\rightarrow}(\mathcal{E})$ *is the least fixed point of the extension transformer.* ■

The unfolding of M is the *unique* $\trianglelefteq$-maximal element in the ω-chain $\mathcal{E}^{\perp} \trianglelefteq ext_{\rightarrow}(\mathcal{E}^{\perp}) \trianglelefteq ext_{\rightarrow}{}^{2}(\mathcal{E}^{\perp}) \trianglelefteq \ldots \trianglelefteq ext_{\rightarrow}{}^{n}(\mathcal{E}^{\perp}) \trianglelefteq \ldots$.

Theorem 3. *The unfolding $\mathcal{U}_M$ exists and is unique.*

Proof. Follows directly from the fact that the set of unfolding prefixes is a complete lattice and $ext_{\rightarrow}$ is monotone. □

Example 16. *Consider the prefix presented in Example 15 of the unfolding in Figure 3.2. This prefix has only one enabled event $(x{:=}b, \{(b := y+1, \varnothing)\})$ corresponding to the event* 10 *in Figure 3.2. The extension of the prefix with this event generates the prefix:*



◁

We now prove that the configurations of every unfolding prefix $\mathcal{E}_i \hat{=} ext_{\rightarrow}{}^{i}(\mathcal{E}^{\perp})$ correspond to M-traces of the system. In particular, we need to verify that for every configuration $C \in conf(\mathcal{E}_i)$, every interleaving in $inter(C)$ is a run of M and that any two interleavings in $inter(C)$ reach the same state.

Lemma 10. *For every configuration C of an unfolding prefix $ext_{\rightarrow}{}^{i}(\mathcal{E}^{\perp})$ it holds that:*

1. *$inter(C) \subseteq Runs(M)$ and*
2. *for all $\sigma_1, \sigma_2 \in inter(C)$, we have $st(\sigma_1) = st(\sigma_2)$.*

Proof. Let C be a configuration of $ext_{\rightarrow}{}^{i}(\mathcal{E}^{\perp})$. Since C is finite, we will prove the lemma w.l.o.g. for the first finite $ext_{\rightarrow}{}^{h}(\mathcal{E}^{\perp})$, such that $h \leq i$ and C is a configuration of $ext_{\rightarrow}{}^{h}(\mathcal{E}^{\perp})$. The proof is by structural induction on the chain $\mathcal{E}^{\perp} \trianglelefteq ext_{\rightarrow}(\mathcal{E}^{\perp}) \trianglelefteq \ldots ext_{\rightarrow}{}^{h}(\mathcal{E}^{\perp})$. Let $\mathcal{E} \hat{=} ext_{\rightarrow}{}^{h}(\mathcal{E}^{\perp})$.

Base case. If $\mathcal{E} = \varnothing$, the lemma trivially holds.

Inductive step. Assume $\mathcal{E}$ has been produced by the extension transformer (Definition 16) to the unfolding prefix $\mathcal{E}'$, and let e be the event in $\mathcal{E}$ but not in $\mathcal{E}'$. Also, assume that the lemma holds for $\mathcal{E}'$. It holds that $e \in C$ or $e \notin C$. If $e \notin C$, then C is a configuration of $\mathcal{E}'$ and by the induction hypothesis the lemma holds. Otherwise, $e \in C$. Necessarily e is a $<$-maximal event in C.

We now prove both statements in the lemma separately.

(1.) Let $\sigma \in inter(C)$ be an interleaving of C, and let $C \hat{=} \{e_1, \ldots, e_n\}$. W.l.o.g., assume that σ is of the form

$$\sigma = h(e_1), \ldots, h(e_n)$$

and that $e_i = e$. Clearly, the causes of e is a subset of the events $\{e_1, \ldots, e_{i-1}\}$. Since, by definition of $inter(\cdot)$, $\{e_1, \ldots, e_{i-1}\}$ is a configuration and it does not include e, it is necessarily a configuration of $\mathcal{E}'$. Thus, by applying the induction hypothesis we know that the sequence

$$h(e_1), \ldots, h(e_{i-1})$$

is an execution of M and produces the same global state as another execution that first fires all events in $\lceil e \rceil$ and then all remaining events in $\{e_1, \ldots, e_{i-1}\}$. This means that σ is an execution of M iff the sequence

$$\sigma' \hat{=} \sigma'' \cdot h(f_1) \cdot \ldots \cdot h(f_k) \cdot h(e) \cdot h(g_1) \cdot \ldots \cdot h(g_l)$$

is an execution of M, where $\sigma'' \in \mathit{inter}(\lceil e \rceil)$, $\{f_1, \ldots, f_k\} = \{e_1, \ldots, e_{i-1}\} \setminus \lceil e \rceil$, and $g_1 = e_{i+1}, \ldots, g_l = e_n$.

Now we will show that the sequence $\sigma'' \cdot h(f_1) \ldots h(f_k) \cdot h(e)$, which is a prefix of σ', is an execution. From Definition 16, it follows that σ'' enables $h(e)$, and from the induction hypothesis it follows that σ'' enables $h(f_1)$. Since $\neg(f_1 \# e)$ and $f_1 \notin \lceil e \rceil$, from Definition 16, it follows that $h(f_1) \diamond h(e)$, i.e., the transitions associated to both events commute (at all states). Since both $h(f_1)$ and $h(e)$ are enabled at $st(\sigma'')$, then $\sigma'' \cdot h(f_1) \cdot h(e)$ is a run. Again, the run $\sigma'' \cdot h(f_1)$ enables both $h(e)$ and $h(f_2)$, and for the same reason $h(e) \diamond h(f_2)$. Thus, $\sigma'' \cdot h(f_1) \cdot h(f_2) \cdot h(e)$ is a run. Iterating this argument k times one can prove that

$$\tilde{\sigma} \doteq \sigma'' \cdot h(f_1) \cdot \ldots \cdot h(f_k) \cdot h(e)$$

is a run.

The next step is proving that the execution $\tilde{\sigma}$ can be continued by firing the sequence of transitions $h(g_1), \ldots, h(g_l)$. The argument is akin to the one above. Earlier we shown that $h(e) \diamond h(g_j)$ for $j \in \{1, \ldots, l\}$. Since $\tilde{\sigma}$ enables $h(e)$ and $h(g_1)$, and both commute at $st(\tilde{\sigma})$, then necessarily $\tilde{\sigma} \cdot h(e) \cdot h(g_1)$ is a run and reaches the same state as the execution $\tilde{\sigma} \cdot h(g_1) \cdot h(e)$. Iterating this argument l times one can show that, similarly, $\tilde{\sigma} \cdot h(e) \cdot h(g_1) \cdot \ldots \cdot h(g_l)$ is a run and reaches the same state as the execution $\tilde{\sigma} \cdot h(g_1) \cdot \ldots \cdot h(g_l) \cdot h(e)$. Thus, σ is a run.

(2). We show that any two executions in $\mathit{inter}(C)$ reach the same state. This is straightforward to show using the arguments we have introduced above. In particular, we have already shown that every linearization of C is h-labelled by a run of M that reaches the same state as the run that labels any other linearization of the same events that fires e last in the sequence. □

Lemma 10 shows that unfolding is *well-formed* in the sense that it does not represent runs that are not in $\mathit{Runs}(M)$. Also, it shows that $st(C)$ is a singleton when C belongs to an unfolding prefix.

A second important result is that the unfolding is *sound*, i.e. all runs are represented, and there is a bijection between configurations and Mazurkiewicz traces, i.e. every run of the system is uniquely represented by a configuration of the unfolding [4].

Theorem 4. *For every run $\sigma \in \mathit{Runs}(M)$, there exists a unique configuration $C \in \mathit{conf}(\mathcal{U}_M)$ such that $\sigma \in \mathit{inter}(C)$.*

[4] In [RSSK15a], we considered this property to be *completeness* while here we consider that a result based on the unfolding is *sound* iff all runs are represented.

Proof. The proof is by induction on the length $|\sigma|$ of the run.

Base Case. If $\sigma = \varepsilon$, then we have that σ is the only interleaving of the empty configuration $\varnothing$. Any non-empty configuration will contain an event labelled by an action. Thus, $\varnothing$ is the only configuration that represents ε.

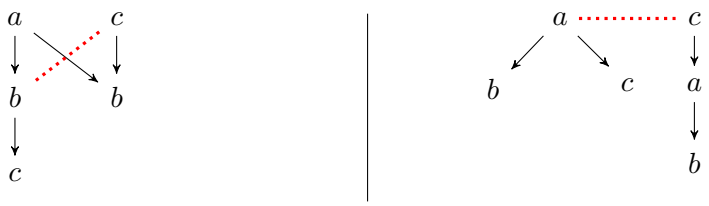
Inductive Step. Consider $\sigma = \sigma'.t_{k+1}$. By the induction hypothesis, we assume that there exist a unique configuration C' such that $\sigma' \in \mathit{inter}(C')$.

By Lemma 10, all runs in $\mathit{inter}(C')$ reach the same state s and σ' is such a run. Hence, t_{k+1} is enabled at state s. If for all events $e \in \max_{C'}$, it holds that $h(e) \diamond\!\!\!\!\diamond t_{k+1}$, then C' is a valid configuration H and by Definition 16 there is a configuration $C = C' \cup \{e'\}$ with $e' \hat{=} (t_{k+1}, C')$. Otherwise, we construct a valid H by considering subsets of C' removing a maximal event $e \in \max_{C'}$, such that $h(e) \diamond t_{k+1}$. Note that by the definition of independence, removing a maximal event whose label is independent with t_{k+1} does not disable t_{k+1}. We always obtain a valid H since C' is a finite set and $\varnothing$ is always a valid H. Since the unfolding is a $\diamond\!\!\!\!\diamond$-branching deterministic $\mathcal{E}$, it follows that $C \hat{=} H \cup \{e'\}$ with $e' \hat{=} (t, H)$ is a configuration and it is unique. □

3.4 The Unfolding Domain

In the previous section we defined the unfolding of a system M as the $\trianglelefteq$-maximal deterministic $\diamond\!\!\!\!\diamond$-branching PES in the lattice of prefixes $\mathcal{E}_{\diamond\!\!\!\!\diamond}$ where $\diamond\!\!\!\!\diamond$ is a dependence relation on the actions of M. Furthermore, we showed that the unfolding $\mathcal{U}_{M,\diamond}$ can be represented by the pair $(\diamond\!\!\!\!\diamond, E)$ where E is the set of *named* events. For each independence relation $\diamond \in \overline{\diamond}_M$ according to Definition 13, there is an unfolding $\mathcal{U}_{M,\diamond}$. In this section we identify the relationship between the domain of dependence relations and the associated domain of unfoldings as a Galois connection where the abstraction is the unfolding procedure of Definition 17. This result allows us to precisely understand the connection between unfoldings generated from multiple independence relations.

Example 17. *Consider a system M from a program with two threads where the first thread has actions a and b the second thread is the single action c. Assume that $\diamond_1 \hat{=} \{(a,c)\}^{\leftrightarrow}$ and that $\diamond_2 \hat{=} \{(b,c)\}^{\leftrightarrow}$ are independence relations in $\overline{\diamond}_M$. These two independence relations* generate *the following unfoldings (assuming that concurrent actions do not disable each other):*



These prime event structures are equivalent representations of the set of executions $\{abc, acb, cab\}$.

Consider the representation of these unfoldings as pairs $(E_1, \diamond_1)$, $(E_2, \diamond_2)$. *Some events such as* $e_1 \hat{=} (a, \varnothing)$ *and* $e_2 \hat{=} (c, \varnothing)$ *are elements of* E_1 *and* E_2. *However, it is clear that* E_1 *and* E_2 *are incomparable. For example, the event* $(b, \{e_1, e_2\})$ *is in* E_1 *and not in* E_2 *and the event* $(a, \{e_2\})$ *is in* E_2 *and not in* E_1. ◁

As shown by Example 17, there are multiple unfoldings associated to a system M. Furthermore, $\diamond_1 \cup \diamond_2$ and $\diamond_1 \cap \diamond_2$ are independence relations of M. A natural question to pose is what is the relationship between the resulting unfoldings ($\mathcal{U}_{M,\diamond_1\cup\diamond_2}$ and $\mathcal{U}_{M,\diamond_1\cap\diamond_2}$) with respect to $\mathcal{U}_{M,\diamond_1}$ and $\mathcal{U}_{M,\diamond_2}$?

Since all these unfoldings are equivalent in the sense that they represent the interleaving space $Runs(M)$, we want to explore the most efficient representation which should be considered the *best unfolding*. Not surprisingly, there can be exponential differences in the sizes of these unfoldings where the *worst* unfolding constructed with the empty independence relation is the computation tree of M. We will show that the *best* unfolding corresponds to the one associated with the *largest* independence relation and equivalently is the unfolding with the smallest set of events.

We denote by $\mathbb{U}_M$ the set of unfoldings of M. By Section 3.3.2, there is one per valid independence relation. Given two unfoldings $\mathcal{U}_{M,\diamond_1}$, $\mathcal{U}_{M,\diamond_2}$ where $\diamond_1, \diamond_2 \in \overline{\diamond}_M$ they are equivalent to pairs $(\circledast_1, E_1)$ and $(\circledast_2, E_2)$ such that $\mathcal{U}_{M,\diamond_1} = [\![E_1]\!]_{\circledast_1}$ and $\mathcal{U}_{M,\diamond_2} = [\![E_2]\!]_{\circledast_2}$. The following Lemma answers the question posed above by proving that the unfolding generated by the union (intersection) of independence relations is unfolding generated by the intersection (union) of their named events.

Lemma 11. *Let* $\diamond_1, \diamond_2 \in \overline{\diamond}_M$ *be independence relations over the actions of* M *with corresponding unfoldings* $\mathcal{U}_{M,\diamond_1} = [\![E_1]\!]_{\circledast_1}$ *and* $\mathcal{U}_{M,\diamond_2} = [\![E_2]\!]_{\circledast_2}$. *It holds that* $\mathcal{U}_{M,\diamond_1\cup\diamond_2} = [\![E_1 \cap E_2]\!]_{\circledast_1\cap\circledast_2}$ *and* $\mathcal{U}_{M,\diamond_1\cap\diamond_2} = [\![E_1 \cup E_2]\!]_{\circledast_1\cup\circledast_2}$.

Proof.

We first prove that $\mathcal{U}_{M,\diamond_1\cup\diamond_2} = [\![E_1 \cap E_2]\!]_{\circledast_1\cap\circledast_2}$.

($\subseteq$). Let $e \hat{=} (a, H) \in \mathcal{U}_{M,\diamond_1\cup\diamond_2}$. We will show that $e \in E_1 \cap E_2$. Consider an interleaving $\sigma \in inter(H)$. It holds by Theorem 4 that there is a unique configuration $C_1 \in$

$conf(\mathcal{U}_{M,\diamond_1})$ and $C_2 \in conf(\mathcal{U}_{M,\diamond_2})$ such that $\sigma \in inter(C_1)$ and $\sigma \in inter(C_2)$. The first step to show that e is in both unfoldings is to show that $H = C_1 = C_2$. We show this statement by structure induction on σ.

Base case. $\sigma = \varepsilon$. It holds that $H = C_1 = C_2 = \varnothing$.

Step. Assume that $\sigma = \sigma' \cdot t$ and that $H = H' \cup \{e'\}, C_1 = C_1' \cup \{e_1\}, C_2 = C_2' \cup \{e_2\}$ with $H' = C_1' = C_2'$. It holds that $h(\bar{e}) = h(e_1) = h(e_2) = t$ where $\bar{e} \hat{=} (t, \bar{H})$, $e_1 \hat{=} (t, H_1)$ and $e_2 \hat{=} (t, H_2)$. It is clear that t is enabled at $st(H)$, $st(C_1)$ and $st(C_2)$. By Equation (3.5), we know that for all maximals events m of H', $h(m) \diamond\!\!\!\!\!\!\diamond_1 \cap \diamond\!\!\!\!\!\!\diamond_2 t$. Then, we have that $H' = C_1' = C_2'$ since this set of maximal events is also dependent with one of the dependence relations. Thus, $H = C_1 = C_2$.

Since $e \in \mathcal{U}_{M,\diamond_1 \cup \diamond_2}$, we know that a is enabled at H. Therefore $e \in \mathcal{U}_{M,\diamond_1}$ and $e \in \mathcal{U}_{M,\diamond_2}$.

($\supseteq$). Let $e \hat{=} (a, H)$ such that $e \in \mathcal{U}_{M,\diamond_1}$ and $e \in \mathcal{U}_{M,\diamond_2}$. By Equation (3.5), we know that $a \in en(H)$ and that for all maximals events m of H, $h(m) \diamond\!\!\!\!\!\!\diamond_1 a$ and $h(m) \diamond\!\!\!\!\!\!\diamond_2 a$. Therefore,for all maximal events m of H, $h(m) \diamond\!\!\!\!\!\!\diamond_1 \cap \diamond\!\!\!\!\!\!\diamond_2 a$ and e is an event of $[\![E_1 \cup E_2]\!]_{\diamond\!\!\!\!\!\!\diamond_1 \cup \diamond\!\!\!\!\!\!\diamond_2}$.

We now prove that $\mathcal{U}_{M,\diamond_1 \cap \diamond_2} = [\![E_1 \cup E_2]\!]_{\diamond\!\!\!\!\!\!\diamond_1 \cup \diamond\!\!\!\!\!\!\diamond_2}$.

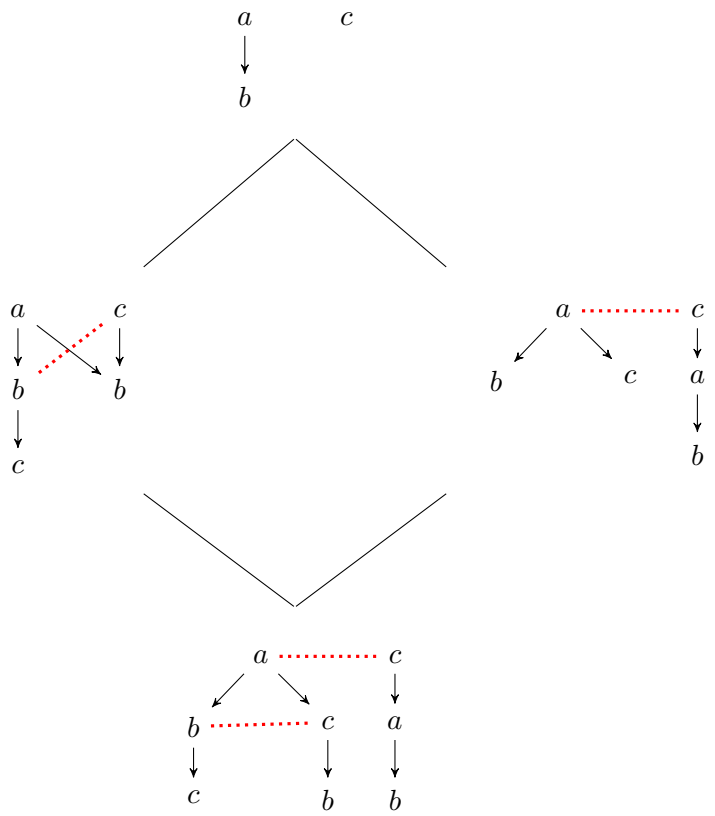
($\subseteq$). Let $e \hat{=} (a, H) \in \mathcal{U}_{M,\diamond_1 \cap \diamond_2}$. We will show that $e \in E_1 \cup E_2$. Consider an interleaving $\sigma \in inter(H)$. It holds by Theorem 4 that there is a unique configuration $C_1 \in conf(\mathcal{U}_{M,\diamond_1})$ and $C_2 \in conf(\mathcal{U}_{M,\diamond_2})$ such that $\sigma \in inter(C_1)$ and $\sigma \in inter(C_2)$. The first step to show that e is in one of the unfoldings is to show that $H = C_1$ or $H = C_2$. Without loss of generality, we will show that $H = C_1$. We show this statement by structure induction on σ.

Base case. $\sigma = \varepsilon$. It holds that $H = C_1 = \varnothing$.

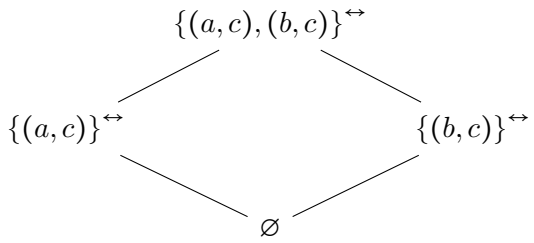
Step. Assume that $\sigma = \sigma' \cdot t$ and that $H = H' \cup \{e'\}, C_1 = C_1' \cup \{e_1\}$ with $H' = C_1'$. It holds that $h(\bar{e}) = h(e_1) = t$ where $\bar{e} \hat{=} (t, \bar{H})$, $e_1 \hat{=} (t, H_1)$. It is clear that t is enabled at $st(H)$ and $st(C_1)$. By Equation (3.5), we know that for all maximals events m of H', $h(m) \diamond\!\!\!\!\!\!\diamond_1 \cup \diamond\!\!\!\!\!\!\diamond_2 t$. Since $\diamond\!\!\!\!\!\!\diamond_1 \subseteq \diamond\!\!\!\!\!\!\diamond_1 \cup \diamond\!\!\!\!\!\!\diamond_2$ it also holds that $H \in \mathcal{H}_{\mathcal{U}_{M,\diamond_1},a}$. Since $e \in \mathcal{U}_{M,\diamond_1 \cap \diamond_2}$, we know that a is enabled at H. Therefore $e \in \mathcal{U}_{M,\diamond_1}$.

($\supseteq$). Let $e \hat{=} (a, H)$ such that $e \in \mathcal{U}_{M,\diamond_1}$ or $e \in \mathcal{U}_{M,\diamond_2}$. By Equation (3.5), we know that $a \in en(H)$ and that for all maximals events m of H, $h(m) \diamond\!\!\!\!\!\!\diamond_1 a$ or $h(m) \diamond\!\!\!\!\!\!\diamond_2 a$. Therefore, for all maximal events m of H, $h(m) \diamond\!\!\!\!\!\!\diamond_1 \cap \diamond\!\!\!\!\!\!\diamond_2 a$ and e is an event of $[\![E_1 \cap E_2]\!]_{\diamond\!\!\!\!\!\!\diamond_1 \cap \diamond\!\!\!\!\!\!\diamond_2}$. □

Example 18 (The Lattice of Unfoldings)**.** *We now show the complete lattice of Example 17.*



This lattice is obtained from the following lattice of independence relations:



◁

The set of sets of events of $\mathbb{U}_M$, denoted by $\mathbb{E}_M$, forms the lattice $\langle \mathbb{E}_M, \supseteq, \cap, \cup, E_\perp, E^\top \rangle$, where bottom $E_\perp$ is the largest set. Note that we are using the order dual here.

As a consequence, the lattice of independence relations and the lattice of events in the unfolding are connected by a Galois insertion.

Corollary 1. $(\overline{\diamond}_M, \subseteq) \underset{\alpha_{\mathbb{U}_M}}{\overset{\gamma_{\mathbb{U}_M}}{\leftrightarrows}} (\mathbb{E}_M, \supseteq)$ *where:*

$$\alpha_{\mathbb{U}_M} : \overline{\diamond_M} \to \mathbb{E}_M \qquad \gamma_{\mathbb{U}_M} : \mathbb{E}_M \to \overline{\diamond_M}$$
$$\alpha_{\mathbb{U}_M}(\diamond) \mathrel{\hat{=}} \pi_2(\mathcal{U}_{M,\diamond}) \qquad \gamma_{\mathbb{U}_M}(E) \mathrel{\hat{=}} \bigcup \left\{ \diamond \in \overline{\diamond_M} \mid \alpha_{\mathbb{U}_M}(\diamond) \supseteq E \right\}$$

is a Galois insertion.

The main observation from Corollary 1 is that every unfolding for the system has a common prefix that could be soundly used to represent the interleaving space. This common prefix can be considered to the *best* unfolding and is obtained with the smallest dependence. Furthermore, it may be the case that $\diamond_1 \subset \diamond_2$ and $\mathcal{U}_{M,\diamond_1} = \mathcal{U}_{M,\diamond_2}$.

We now investigate a constructive definition of the concretization function $\gamma_{\mathbb{U}_M}$. In particular, given the set of events $E \in \mathbb{E}_M$ of an unfolding, can we directly obtain a dependence relation $\diamond\!\!\!\diamond$ such that $\alpha_{\mathbb{U}_M}(\diamond) = E$?

For a general model of computation, the answer is negative as there is no way to check if two local configurations form a configuration. However, for programs as specified in Section 2.4 it is possible to obtain a sound approximation (because two incomparable independence relations can lead to the same unfolding).

Definition 18 (Dependence)**.** *Let $E \in \mathbb{E}_M$ be the set of events of an unfolding of M. The binary relation $D_E \subseteq Act \times Act$ over actions satisfies:*

$$
\begin{array}{l}
a\ D_E\ b \text{ iff exists } e, e' \in E \text{ such that} \\
\qquad (e <^i e' \text{ and } h(e) = a \text{ and } h(e') = b) \text{ or} \\
\qquad (h(e) = a = lock(x) \text{ and } h(e') = b = lock(x) \text{ and } \lceil e \rceil = \lceil e' \rceil)
\end{array}
$$

We define the binary relation $\diamond\!\!\!\diamond_E \hat{=} D_E^{\leftrightarrow}$ as the symmetric closure of D_E. ■

3.5 Discussion

In this section we discuss related work and present several open questions that arise in our semantics.

Comparison to Petri Net Unfoldings In contrast to our parametric semantics, classical unfoldings of Petri nets [ERV02] (or synchronized state machines [EH08], or process algebras [LB99]) use a fixed independence relation. This relation, valid only for safe nets, is the complement of the following dependence relation. Given two transitions t and t',

$$t \diamond\!\!\!\diamond_n t' \text{ iff } (post(t) \cap pre(t') \neq \varnothing) \text{ or } (post(t') \cap pre(t) \neq \varnothing) \text{ or } (pre(t') \cap pre(t) \neq \varnothing),$$

where $pre(t)$ and $post(t)$ are respectively the *preset* and *postset* of t. Classic Petri net unfoldings (of safe nets) are therefore an instance of our semantics. A well known

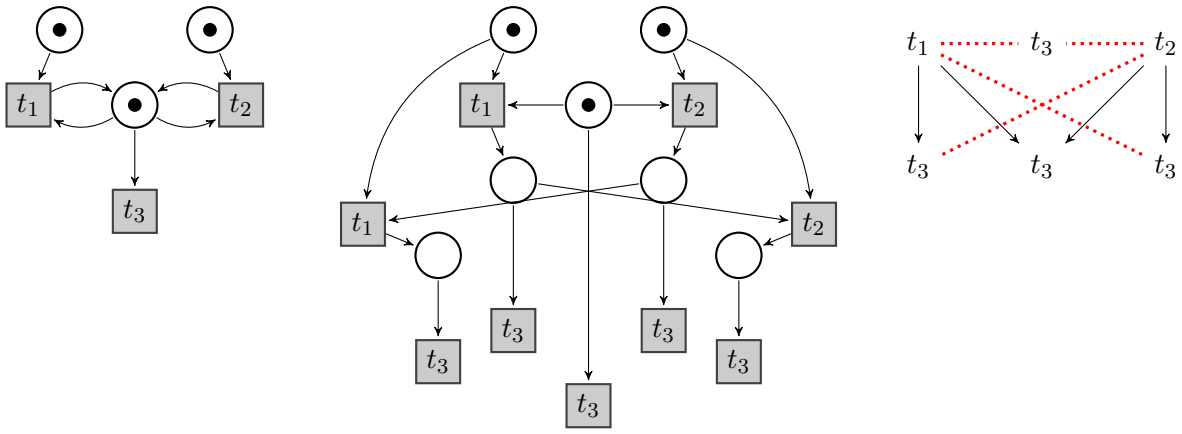


Figure 3.3: (a) A Petri net; (b) its classic unfolding; (c) our parametric semantics.

limitation of classic unfoldings are transitions that *read* places, e.g., t_1 and t_2 in Figure 3.3 (a). Since $t_1 \otimes\!\!\!\otimes_n t_2$, the classic unfolding, Figure 3.3 (b), sequentializes all their occurrences. A solution to this is the so-called *place replication (PR) unfolding* [MR95], or alternatively *contextual unfoldings* (which have internally the same asymptotical size as the PR-unfolding).

This problem vanishes with our parametric unfolding. It suffices to use a dependency relation $\otimes\!\!\!\otimes'_n \subset \otimes\!\!\!\otimes_n$ that makes transitions that *read* common places independent. The result is that our unfolding, Figure 3.3 (c), can be of the same size as the PR-unfolding, i.e., exponentially more compact than the classic unfolding. For instance, when Figure 3.3 (a) is generalized to n *reading* transitions, the classic unfolding would have $\mathcal{O}(n!)$ copies of t_3, while ours would have $\mathcal{O}(2^n)$. The point here is that our semantics naturally accommodates a more suitable notion of independence without resorting to specific ad-hoc tricks.

Furthermore, although this work is restricted to *unconditional* independence, we conjecture that an adequately restricted *conditional* dependence would suffice, e.g., the one of [KP92]. Gains achieved in such setting would be difficult with classic unfoldings. In [KSH14], it is presented an unfolding-based semantics for programs by conversion to Petri nets which corresponds to an instance of our general framework.

Prime Event Structures vs Mazurkiewicz Trace Theory In this chapter, we showed a correspondence between a subclass of prime event structures and Mazurkiewicz trace theory. The closest related work in this direction is presented in [RT91] where the main result is a representation theorem between a subclass of prime event structures and trace theory is presented. Our work is more than an instance of such

representation theorem applied for programs as the class defined in Section 3.3.1 is alternative characterization of the restrictions of prime event structures in [RT91]. In particular, we show that the algebraic structure required for event structures to be equivalent to trace semantics is essentially a concurrent alphabet where actions are replaced by events. This in turn, emphasizes the key novel aspect in the fixpoint definition of the unfolding which is the definition of the set of histories of an event. This is also novel with respect to [RSSK15a] where this subclass of event structures is not characterized.

The Prime Unfolding The relationship between unfoldings and independence relations in Section 3.4 answers the basic question between the differences of multiple unfoldings of a program. However, it also points to the fact that an unfolding is not *per se* a useful object for verification as it can contain a lot of redundant events. This is quite tied to the fact that we use an unconditional independence relation while more expressive independence relations could achieve significant more reduction. In the context of programs, an interesting verification question concerns fireability of actions or local state reachability. Ideally we would like to construct a prime event structure such that for each reachable statement in the program, the number of events associated with it is minimal. It is clear that this would be a prefix of the best unfolding and consequently a prefix of any unfolding of the program. We call such prefix the *prime unfolding*. Defining and study the properties of such prefix could be of practical interest. Basic experiments show that because of unconditional independence and the properties of deterministic R-branching event structures, there is a tilling pattern in the unfolding. For example, considering a program with n threads each composed of a write to the same variable. After constructing the first configuration which a linear ordering of these n writes, we can immediately identify the rest of the unfolding. This is because the prime unfolding in this case is simply a prime event structure with n conflicting write events.

Conclusions In this chapter, we presented a new family of *true concurrency* semantics based on prime event structures that unifies the underlying semantics of the two algorithmic approaches of interest: DPORs and net unfoldings. As a main consequence, we have showed that both algorithms can be described over a common data-structure where the source of reduction is the same: independence of concurrent actions.

Chapter 4

Algorithms for Independence-based Explorations

DPORs [FG05, GFYS07, YWY06, YCGK08, AAJS14, AAA+15b] and Petri net unfoldings [EH08, McM93a] are two widely used approaches to overcome state explosion in algorithmic verification of concurrent systems. In particular, DPORs have been widely applied to programs while unfoldings have been mainly studied in the context of Petri nets. In both approaches, the independence of concurrent actions is a fundamental parameter that drives the reduction. Stateless DPORs use independence to dynamically compute a provably-sufficient subset of enabled transitions at every state without keeping states of previous executions. In contrast, Petri net unfoldings use independence to represent the set of executions as a prime event structure.

Despite impressive advances in the field, both unfoldings and DPORs have shortcomings. We now describe six of them.

Current unfolding algorithms (1) need to solve an NP-complete problem when adding events to the unfolding [McM93a], which seriously limits the performance of existing unfolders as the structure grows. They also (2) suffer from space scalability issues as they cannot selectively discard visited events from memory. DPORs, on the other hand, explore linearizations of Mazurkiewicz traces which (3) can outnumber the events in the corresponding unfolding by an exponential factor. Furthermore, DPORs often (4) explore the same states repeatedly [YCGK08], and combining them with stateful search is difficult because of the dynamic nature of DPOR [YCGK08, YWY06]. The same holds when extending DPORs to (5) cope with non-terminating executions. Lastly, (6) existing stateless PORs do not exploit additional available memory for any other purpose.

Available solutions or promising directions to address these six problems can be found in, respectively, the opposite approach. DPORs inexpensively add events to

the current execution, contrary to unfoldings (1). They easily discard events from memory when backtracking, which addresses (2). On the other hand, while PORs explore Mazurkiewicz traces (*maximal configurations*), unfoldings explore events (*local configurations*), thus addressing (3). Explorations of repeated states and pruning of non-terminating executions is elegantly achieved in unfoldings by means of cutoff events solving (4) and (5).

Example 19. *We illustrate problems (3), (4), and (5), and explain how our DPOR deals with them. The following code is the skeleton of a producer-consumer program. Two concurrent producers write in, resp.,* `buf1` *and* `buf2`*. The consumer accesses the buffers in sequence.*

```
Thread P1:                Thread P2:                Thread C:
  while (1):                while (1):                while (1):
    lock(m1)                  lock(m2)                  lock(m1)
    if (buf1 < MAX):          if (buf2 < MAX)           if (buf1 > MIN):
       buf1++                    buf2++                    buf1--
    unlock(m1)                unlock(m2)                unlock(m1)
                                                        // same for m2, buf2
```

Lock and unlock operations on both mutexes `m1` *and* `m2` *create many Mazurkiewicz traces. However, most of them have isomorphic* suffixes*, e.g., producing two items in* `buf1` *and consuming one reaches the same state as only producing one. After the common state, both traces explore identical behaviors and only one needs to be explored. We use cutoff events, inherited from unfolding theory [ERV02, BHK+14], to dynamically stop the first trace and continue only with the second. This addresses (4) and (5), and partially deals with (3). Observe that cutoff events are a form of semantic pruning, in contrast to the syntactic pruning introduced by, e.g., bounding the depth of loops, a common technique for coping with non-terminating executions in DPOR. With cutoffs, the exploration can build unreachability* proofs*, while depth bounding renders DPOR incomplete, i.e., it can only* find bugs. ◁

This chapter lays out a generic DPOR algorithm on top of an unfolding as defined in Chapter 3. Our main result is a novel optimal DPOR that explores at most once every M-trace, and potentially achieves super-optimality owing to cutoff events. It also copes with non-terminating systems and exploits all available RAM with a *cache memory* of events, speeding up revisiting events.

This provides a solution to (4), (5), (6), and a partial solution to (3). Our algorithm can alternatively be used as a stateless unfolding exploration, partially addressing (1) and (2).

Our result reveals DPORs as algorithms exploring an object that has richer structure than a plain directed graph. In particular, unfoldings provide a solid notion of event *across multiple executions*, and a clear notion of conflict which is hidden in DPOR algorithms.

To summarize, the main contributions of this chapter are:

1. Reformulation of POR theory using event structures (Section 4.1).
2. Unfolding exploration that visits the set of Mazurkiewicz traces using an optimal strategy (Section 4.2).
3. Application of the theory of cutoffs to support non-terminating executions and stateful explorations (Section 4.3).
4. Implementation of the exploration algorithm for multi-threaded C programs (Section 4.4).
5. Empirical evaluation against state-of-the-art DPOR tools (Section 4.5).

For the rest of this chapter, we fix a system $M \mathrel{\hat{=}} (\mathit{State}, \mathit{Trans}, \tilde{s})$ and assume that: (1) M is the system of a data deterministic program and (2) $\mathit{Reach}(M)$ is finite. Let $\mathcal{U}_{M,\diamond} \mathrel{\hat{=}} \langle E, <, \#, h\rangle$ be the unfolding of M under $\diamond$, which we abbreviate as $\mathcal{U}$.

4.1 Dynamic Partial Order Reductions

In this section, we present several advantages of guiding a DPOR exploration using a prime event structure. Consider the program in Figure 4.1.

```
Thread W:
  z:=1;
```

```
Thread R1:
  x:=1;
  a:=z;
```

```
Thread R2:
  y:=1;
  b:=z;
```

Figure 4.1: Simple multi-threaded program with initial state $\tilde{s} \mathrel{\hat{=}} x \mapsto 0, y \mapsto 0, z \mapsto 0$. Assume that the variables x, y, z are global and the variables a, b are local. For simplicity, we do not consider program counters.

The state graph in Figure 4.2 compactly represents the 30 executions of the program. The initial state of the program denoted with the gray box. The transitions of `Thread W` are represented with the diagonal edges. The transitions of `Thread`

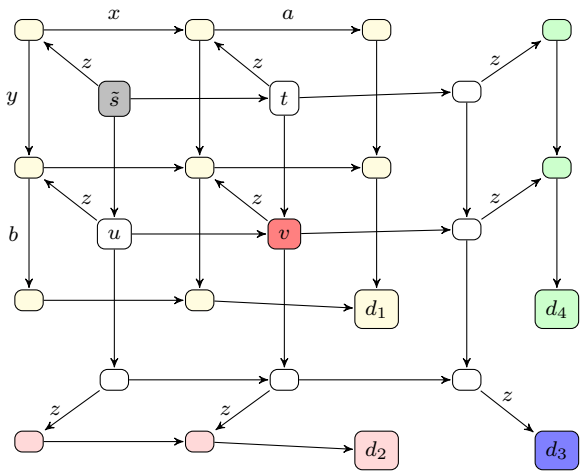


Figure 4.2: State graph of program in Figure 4.1. Since all the statements write to a distinct variable, we abbreviate the statements to their associated written variable, e.g., $x := 1$ is represented as x.

`R1` are represented with the edges from left to right and the ones from `Thread R2` from top to bottom. There are four deadlock/final states in this program: d_1: $x, y, z, a, b \mapsto 1$; d_2: $x, y, z, a \mapsto 1, b \mapsto 0$; d_3: $x, y, z \mapsto 1, a, b \mapsto 0$ and d_4: $x, y, z, b \mapsto 1, a \mapsto 0$.

It is clear that one does not need to explore the complete graph to reach the four deadlock states. Intuitively, the main idea in a partial order reduction (POR) algorithm is to explore a subgraph of Figure 4.2 where a property of interest is preserved. This subgraph is typically known as the *reduced graph*. As we mentioned before, we focus on reachability of deadlock states.

The basic approach of a POR is to discard transitions from the original graph $\mathcal{G} \hat{=} (State, Trans)$ without compromising the reachability of the deadlock states in the original state graph. The states that become unreachable in this process can be safely removed and we obtain the reduced graph $\mathcal{G}_R \hat{=} (State' \subseteq State, Trans' \subseteq Trans)$. This process is effectively removing paths that lead to the same final state of a path already represented in the reduced graph. Note that the reduced graph is usually not unique. More formally, consider the kernel of the state function where two maximal runs σ and σ' are *equivalent*,i.e. $\sigma \equiv_{st} \sigma'$ iff $st(\sigma) = st(\sigma')$. To explore any deadlock state d such that $st(\sigma) = d$, it is *sufficient* to explore an arbitrary run σ' such that $\sigma' \equiv_{st} \sigma$.

A search strategy, known as *selective search*, iteratively visits the states of $\mathcal{G}$ by selecting at every state s a *subset* $P \subseteq en(s)$, and recursively exploring every state s'

with $t \in P$ and $s \xrightarrow{t} s'$. The selected set P must satisfy that if $\sigma \in Act^*$ is the sequence leading to s and there is a continuation $\sigma' \in Act^*$ such that $\sigma \cdot \sigma'$ is a maximal run, then P must contain an action t satisfying that $\sigma \cdot t$ is a prefix of at least one execution $\sigma'' \equiv \sigma.\sigma'$ for some equivalence relation $\equiv$.

By considering a POR as backtracking search strategy that explores executions as elements of equivalence classes, a natural notion of *optimal exploration* arises in this context. In particular, given an equivalence relation over the set of maximal executions of a program, we say that an exploration is *optimal* when it explores only one representative per equivalence class. Furthermore, the exploration does not *get stuck* by backtracking before it reaches a deadlock state. It is important to note that optimality is tied to an equivalence relation. Thus, it is possible for an exploration to be super-optimal for some equivalence if it is optimal for a coarser relation. In order to perform a faithful comparison between DPOR explorations, it is crucial that they rely on the same set of equivalence classes.

We are interested in PORs where the equivalence between runs is based on the independence of concurrent actions. In particular, we consider dynamic PORs [FG05] where the independence relation is computed dynamically. This is of practical relevance as the lack of alias information can limit the reduction obtained by static PORs. Recent DPORs techniques rely on more abstract notions of independence which can potentially lead to fewer equivalence classes [CCP+17, AJLS18].

In Figure 4.2, it is clear that several concurrent actions (interpreted as statements in the threads) are independent as they commute in all states of the program (according to Definition 12). For example, $x := 1$ are $y := 1$ are independent.

It is widely known that the reduction obtained from an advanced DPOR involves some form of global reasoning – it is unsound to not explore both enabled transitions of a state s simply because their actions commute at s. For example, consider the red state v in Figure 4.2. Even though the actions a and b are independent, we have to explore both states reachable from a and b to not risk missing a deadlock state – if we discard the a-labelled transition, we could miss the deadlock state d_4.

However, there is a simple type of reduction called *sleep sets* [God96] that prevents the exploration from revisiting the final states in concurrency diamonds. This reduction will not remove enough transitions to reduce the number of visited states.

We now present the intuition and an informal definition of sleep sets (see [God96, Chapter 5] for details). Intuitively, a sleep set T at a state s is a set of actions from s that provably do not need to be explored because the transition associated with those actions have already been explored and form concurrency diamonds.

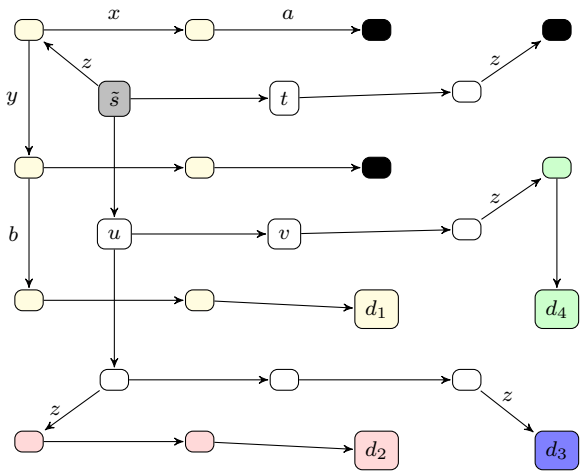


Figure 4.3: State graph explored using Sleep Sets.

Let s be a state of the graph and *sleep(s)* be the current sleep set based on the exploration so far where initially $sleep(s) = \varnothing$ for all states. Consider two enabled actions a, b at s which are not in the sleep set of s. Suppose that we first consider a. The sleep set at the state reached by a is the sleep set at s minus the actions dependent with a. When the exploration returns at the state s, we add a to the sleep set of s. Thus, when we consider the extension with b, if a is independent with b, the sleep set at $b(s)$ will contain a effectively disabling any transition labelled with this action.

In Figure 4.3, we present a possible reduced graph using sleep sets. Assume that we explore the original state graph using a DFS that will reach the deadlock states in lexicographical order. In the first execution, we explore $z \cdot y \cdot b \cdot x \cdot a$ reaching d_1. Then, the exploration will backtrack until a state where multiple transitions are enabled. In this case, it would be state reached by $z \cdot y$ where x is also enabled. The sleep set at this state will now be the singleton $\{b\}$. The sleep set at the state reached by $z \cdot y \cdot x$ will continue to be $\{b\}$ since b is independent to x. Thus, at this state, sleep sets prevent the algorithm from taking the enabled transition labelled with b. This is represented by the absence of that transition in Figure 4.3. Consequently, the exploration is forced to take the transition labelled with a and the sleep set at the state reached by $z \cdot y \cdot x \cdot a$ is still the singleton $\{b\}$. We now reach a state s where $en(s) = sleep(s)$ and the exploration has no choice but to backtrack. These states are represented by the black nodes in Figure 4.3. This is a situation known as *sleep-set blocked execution* (SSB) [AAJS14] and intuitively means that the algorithm tried to explore a M-trace already visited.

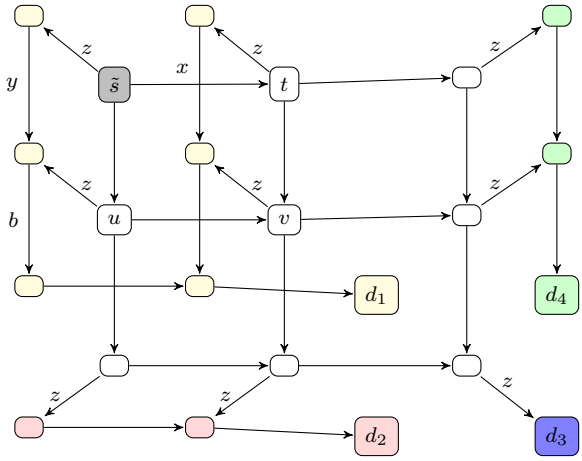


Figure 4.4: State graph explored using Persistent Sets.

Observe that the exploration using sleep sets visits all states and that no concurrency diamond is present in the reduced graph.

An orthogonal method to obtain a reduction is to postpone concurrent actions if we can guarantee that they stay enabled and their transitions form concurrency diamonds with the transitions taken. This form of *forward* reasoning can be used to identify at every state a subset of enabled transitions sufficient for a sound exploration. A widely know technique that employs this reasoning is the persistent set technique [FG05].

Intuitively, a set $P \subseteq Act$ of actions enabled at a state s is a *persistent set* if for every run σ starting at s that does not contain actions from P satisfies that every action in σ is independent with every action in P. This set must be non-empty if the set of enabled actions at s is non-empty. It is clear the set of enabled actions is a persistent set. Also, if an action a is independent with all other actions, then the set $\{a\}$ is trivially persistent at a state where a is enabled.

Example 20. *The set $\{y\}$ is persistent at the initial state $\tilde{s}$ of the state space in Figure 4.2, since regardless of the which execution we pick, the first action dependent with y is y itself. Clearly, $\{z\}$ is not persistent at $\tilde{s}$ since if we consider the execution $x \cdot a$, then a is dependent with z and $a \notin \{z\}$.* ◁

The original DPOR algorithm [FG05] computes persistent sets using a DFS over the state graph assuming that it is acyclic. The algorithms associate to each state a special set of enabled actions, called *backtrack set*, and guarantees that once the exploration terminates, the backtrack sets are persistent. Thus, the algorithm only

explores the actions in the backtrack set. In this section, assume that actions do not disable each other which nevertheless does not poses a fundamental problem.

A DPOR explores executions of the system which are linearizations of M-traces. Once a M-trace has been completely explored we obtain a maximal run $\sigma \in \mathit{Runs}(M)$. To each state reached by a prefix of σ, there is a backtrack set with at least the action taken. Independence is now used to achieve a form of non-chronological backtracking. That is, the algorithm *searches* for possible pairs of concurrent actions in σ that are dependent and can be reversed. More formally, we are looking for actions a, b in $\sigma \hat{=} \pi \cdot a \cdot \tau \cdot b \cdot \omega$ such that $\sigma' \hat{=} \pi' \cdot b \cdot \tau'$ is a run and $\sigma \not\equiv \sigma'$. The algorithm is able to determine this situation using with the independence relation and by constructing π' using only actions from π and τ such that $\pi \prec \pi'$. In order to guarantee that the backtrack set P at $s \hat{=} \mathit{st}(\pi)$ will be a persistent set we add to P the first action in $(\pi' \cdot b) - \pi$ dependent with any action in P.

Although persistent sets can achieve enough reduction for certain states to not be explored, the following example shows some of the limitations.

Example 21. *In Figure 4.4, we present the state graph obtained with DPOR using persistent sets in an exploration that visits the deadlock states in lexicographical order. Furthermore, when multiple actions are enabled, it first chooses actions from* `Thread W`*, then* `Thread R2` *and finally* `Thread R1`*.*

The first observation to note is that even though $\{y\}$ *is a persistent set at* $\tilde{s}$*, the algorithm computed a persistent set at* $\tilde{s}$ *that does not yield any reduction of the state space. Consider the first execution explored by the algorithm to be* $z \cdot y \cdot b \cdot x \cdot a$ *reaching* d_1*. At this stage, the algorithm identifies that there are two pairs of dependent actions* (z, a) *and* (z, b) *as* z *writes to the same variable read by both* a *and* b*. Consider the pair* (z, b)*. In order to find a second execution where the order between these actions is different the algorithm needs to needs to backtrack to the initial state and identify that in order to enable* b *it needs to explore first* y*. Thus, it adds* y *to the backtrack set of the initial state. Similarly, because of the other pair, it will also add* x *to this backtrack set. Note however, that the backtrack sets of the intermediate states up to the initial state are singletons. Therefore, will backtrack to the initial state and we refer to it as non-chronological backtracking.*

The exploration will continue and eventually all deadlock states will be explored. Note that it is possible for the exploration to visit the final state of a concurrency diamond multiple times. ◁

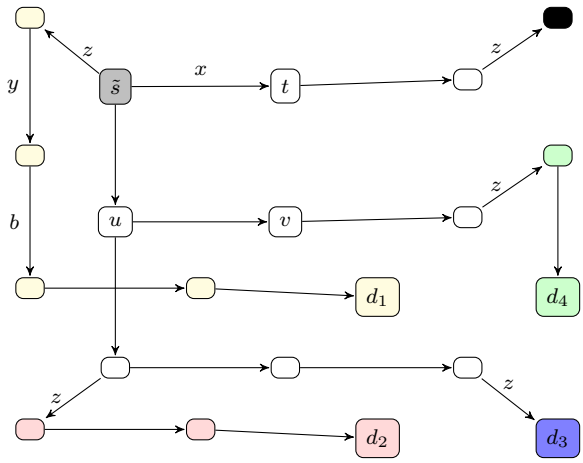


Figure 4.5: State graph explored using Persistent and Sleep sets.

It is know that persistent sets and sleep sets are orthogonal techniques. While persistent sets attempt to aggressively prune the state space by postponing transitions, sleep sets prevent the exploration from being redundant with respect to concurrency diamonds. Both techniques can be trivially combined [FG05] and achieve better reduction than each individual technique. In Figure 4.5, we show a reduced graph explored by a DPOR using persistent and sleep sets. Observe that this reduced graph is strictly smaller than the previous reduced graphs. However, it is still possible to reach a sleep-set blocked execution – this occurs when the exploration backtracks to the initial state a second time having already visited all deadlock states. At this stage, it will have both z and y in the sleep set. Since y is independent with all concurrent actions, this execution is bound to reach a SSB execution.

A possible solution to mitigate SSB executions in a DPOR is to use an improvement of persistent sets called *source sets* [AAJS14, AAJS17a]. Intuitively, a set $P \subseteq Act$ of actions enabled at a state s is a *source set* if for every run σ from s, there is a run $\sigma' \mathrel{\hat{=}} a \cdot \tau$ such that $a \in P$ and $\sigma \equiv \sigma'$. This set must be non-empty if the set of enabled actions at s is non-empty.

Example 22. *The set $\{z, y\}$ is a source set at the initial state of Figure 4.2 since for every execution, there is an equivalent execution starting with y.* ◁

We now briefly explain the main difference between persistent and source sets in a DPOR when updating the backtrack set. Consider the maximal execution $z \cdot y \cdot b \cdot x \cdot a$ in Figure 4.2. As we have seen before, there are two pair of actions that are dependent

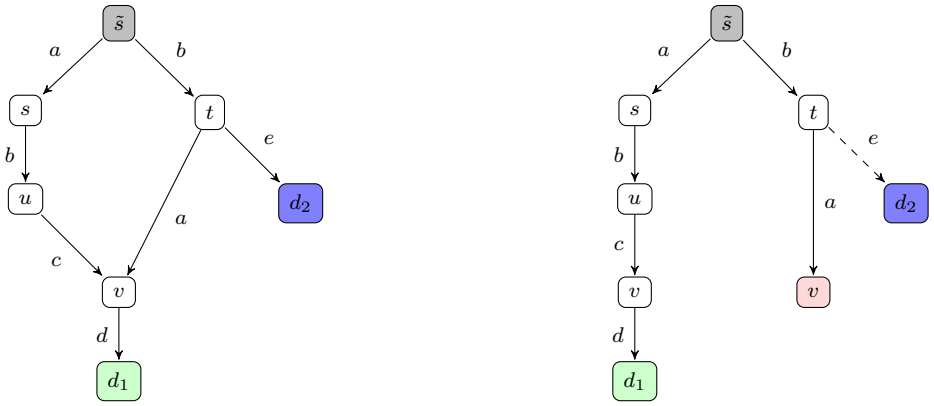
and can be reversed, (z, a) and (z, b). Thus, we need to update the backtrack set at the initial state to reverse both races. Consider the first pair (z, a). We need to update the backtrack set in order to enable the exploration of an execution of the form $\sigma \cdot a \cdot \tau \cdot z$. While persistent sets try to *force* this execution by adding x to the backtrack set as x is required to enable a, source sets are able to add some other action that does not prevent this execution from being reached. In particular, we can add y to the backtrack set. Similarly for the other race we can also add y thus obtaining the set $\{z, y\}$.

Example 23. *A DPOR exploration using source and sleep sets can be optimal for the state graph in Figure 4.2. In particular, with respect to Figure 4.5, the exploration would not explore the x-labelled transition from the initial state.* ◁

However, as shown in [AAJS14], source sets combined with sleep sets are not powerful enough to prevent the exploration of SSB executions.

A practical feature of DPOR is that it is able to perform a stateless exploration of the state space. In particular, this feature is useful in the context of multi-threaded programs where storing states can be expensive if the state involves the heap. However, a stateful exploration is potentially able to stop certain explorations that reach states already visited and yield better reductions.

Example 24. *Consider the following state graph on the left:*



This state graph has two deadlock states d_1 and d_2. It is clear that the sleep sets will not yield any reduction as the state graph does not exhibit concurrency diamonds. Consider the DFS tree explored by a DPOR using persistent or source sets presented on the right. In this case, the explored started by exploring the run $a{\cdot}b{\cdot}c{\cdot}d$ reaching d_1. The race analysis detects that the concurrent actions a and b are dependent and adds b to the backtrack set of the initial state $\tilde{s}$. Then, the exploration backtracks to the initial

state and starts exploring an execution starting with b. It is possible that will revisit the state v again marked with the red node in the tree. A stateful exploration would be able stop the exploration of this execution as all deadlock states from this state have been visited. However, we face the main difficulty in adapting this type of exploration with a stateful search: how to soundly update the backtrack sets of the current states in the stack? If we only consider the current exploration, we would not update the backtrack of the state t with the dashed transition and miss the deadlock state d_2. We can trivially solve this problem by considering all enabled transitions from the states in the stack. However, in that case the gain obtained by stopping the execution at the red state is potentially irrelevant for practical purposes. ◁

4.1.1 Persistent and Source Sets on PES

The DPORs techniques described in the previous section fundamentally use the independence relation to compute sleep, persistent or source sets. In particular, DPORs explore executions which are linearizations of the M-traces that represent the set of executions $Runs(M)$. We have shown in Chapter 3, we can represent these M-traces using a PES. In that context, the notion of state of the system corresponds to a configuration of the PES. Since the PES is deterministic, a set of enabled actions at a state s will correspond to a set of enabled events at a configuration C such that $st(C) = s$.

We now formalize the notion of persistent and source sets as subsets of enabled events of a configuration C of a PES $\mathcal{U}$.

Intuitively, a set of enabled events P at a configuration C is persistent if the events reachable from enabled events not in P are concurrent with all events in P.

Definition 19 (Persistent Set)**.** *A set P of events is a persistent set at a configuration C if it is a set of enabled events of C and for every configuration $C' \supset C$:*

$$\text{If } C' \cap P = \varnothing, \text{ then for all } e \in P, e' \in C' \setminus C \text{ it holds that } e \parallel e' \tag{4.1}$$

Furthermore, P is non-empty if the set of enabled events at C is non-empty. ■

Source sets are more relaxed than persistent sets in the sense that we only require having one minimal event per maximal configuration. This is enough to guarantee that from the current state we are not excluding any maximal configuration.

Definition 20 (Source Set)**.** *A set P of events is a source set at a configuration C if it is a set of enabled events of C and for every maximal configuration $C' \supset C$:*

$$min_{<}(C' \setminus C) \cap P \neq \varnothing \tag{4.2}$$

Furthermore, P is non-empty if the set of enabled events at C is non-empty. ■

Example 25. *We now present some examples of persistent and source sets in the perspective of M-traces and* PES *from the program in Figure 4.1. We consider the independence relation* $\diamond \hat{=} \{(x,y),(x,z),(z,y),(a,b)\}^{\leftrightarrow}$.

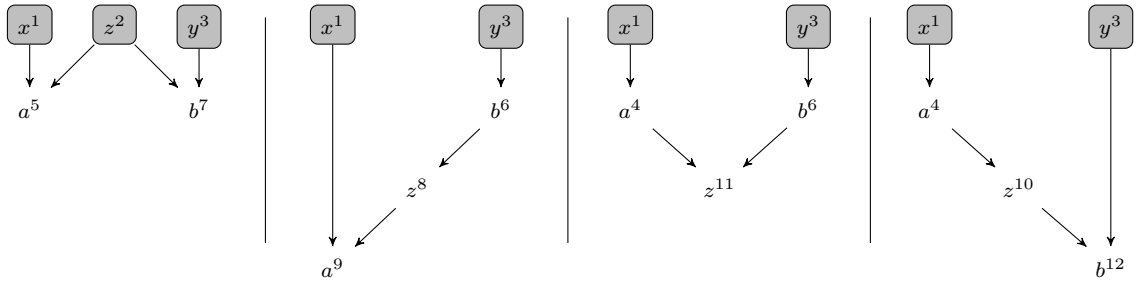


Figure 4.6: M-traces generated from executions in Figure 4.2.

The set of runs correspond to the four M-traces presented in Figure 4.6 reaching d_1, d_2, d_3 *and* d_4. *These represent to the various ways one can reverse the order between dependent actions that are co-enabled at some state. For example, in the leftmost M-trace, we represent the runs where both reads of the global variable x happen after the write action of* `Thread W`.

The four M-traces correspond the four maximal configurations of the PES *that compactly represents these M-traces. The minimal events are represented with the gray nodes. Intuitively, a source set at the initial state, i.e., empty configuration, is a set of* gray events *that contains at least one event per maximal configuration. Thus, the possible source sets are* $\wp(\{x^1,y^3,z^2\}) \setminus \{\varnothing,\{z^2\}\}$. *That is, only the empty set and the set* $\{z^2\}$ *are not source sets.*

Not all source sets are persistent sets. For example, $\{x^1,z^2\}$ *is a source set but not a persistent set at the empty configuration.*

This is clearer in the PES *representation presented in Figure 4.7.*

By considering the first event z^2 *in the persistent set, we need to ensure that the enabled events that are causal predecessors of events in immediate conflict with* z^2 *are also in the persistent set. It is immediate to see from the* PES *that the gray* z^2 *event is in immediate conflict with the red events labelled with* a^4 *and* b^7. *This implies all gray events must be part of the persistent set. The set* $\{x^1\}$ *is a persistent set as it is not in immediate conflict with any other event of the* PES.

Note that the transition system associated with the configuration structure of this PES *is exactly the state graph in Figure 4.2.* ◁

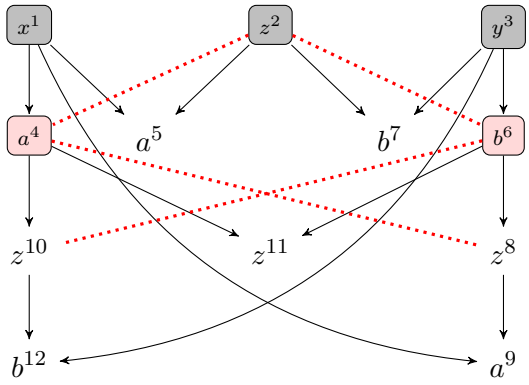


Figure 4.7: Unfolding of Figure 4.2.

We now adapt the selective search strategy over the configurations of a prime event structure.

Let $\mathcal{U} \mathrel{\hat{=}} (E, <, \#, h)$ be a PES. Assume that we are given a map $S \colon \mathit{conf}(\mathcal{U}) \to \wp(E)$ that maps configurations of $\mathcal{U}$ to source sets. By definition, the state graph associated with $\mathcal{U}$ (see Definition 6) where states are configurations and edges represent enabled events, is acyclic. Consider the procedure `Explore`(C) that receives a configuration and recursively calls `Explore`$(C \cup \{e\})$ for all $e \in S(C)$ where $S(C)$ is a source set at configuration C.

We now show that source sets are *sound* as all maximal configurations of $\mathcal{U}$ are eventually an argument to a call of `Explore`.

Theorem 5. *Let $C \in \mathit{conf}(\mathcal{U})$ be a maximal configuration of $\mathcal{U}$. There is a call to* `Explore`(C) *starting from* `Explore`$(\varnothing)$.

Proof. Let $C' \subseteq C$ be a configuration contained in C. By the definition of source sets, $S(C')$ contains an event $e \in C$. From the definition of `Explore`$(\cdot)$, it follows that `Explore`(C') will eventually explore `Explore`$(C' \cup \{e\})$. Since $\varnothing \subseteq C$ and C has size n, it follows that by applying this argument n times, there exists a chain of calls `Explore`(C_1), `Explore`(C_2), ..., `Explore`(C_n) where $C_1 = \varnothing$ and $C_i \subseteq C$. Since $|C| = n$, it follows that $C_n = C$. □

4.2 Unfolding-based DPOR Explorations

In this section, we present several algorithms to explore the unfolding $\mathcal{U}$. In fact, these algorithms can explore an arbitrary PES. For now, we assume that $\mathcal{U}$ is finite, i.e., that all executions of M terminate. We will relax this assumption in Section 4.3.

As opposed to standard unfolding explorations that extend the prefix with a single event (similarly to the semantics presented in Chapter 3) we will explore $\mathcal{U}$ one configuration at a time. Thus, the exploration resembles a DFS-based DPOR exploration with the major difference that it uses the prime event structure to guide the exploration. In particular, our algorithms are backtracking and explore a single execution of the program at a time. We also incorporate multiple constructions such as sleep sets, source sets and wake-up trees present in advanced DPORs [AAJS14]. Intuitively, the algorithm presented in Section 4.2.3 and main contribution of this section is an optimal DPOR over the state graph corresponding to the configuration structure of the input event structure.

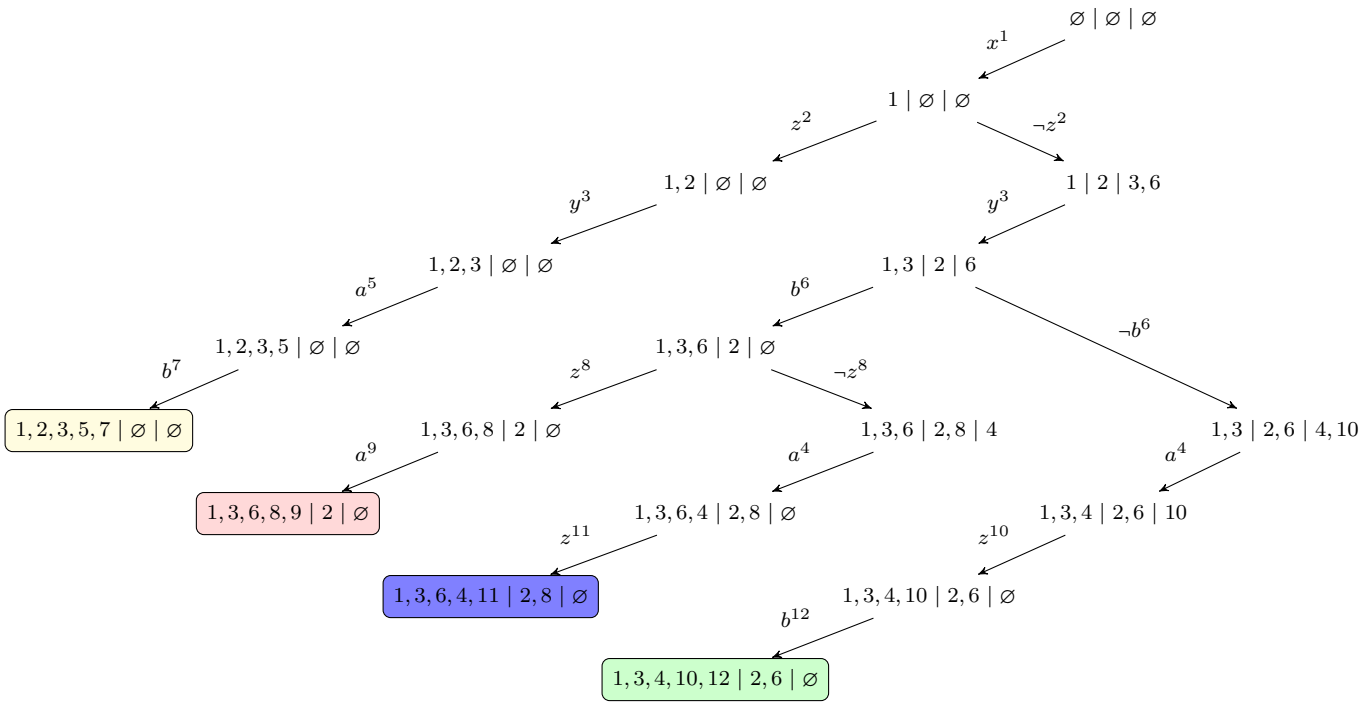


Figure 4.8: Call tree of an optimal unfolding-based exploration.

Example 26. *In Figure 4.8 we present a call tree of our optimal unfolding-based exploration algorithm. The exploration of the prime event structure is represented by a binary tree whose nodes are triples (C, D, A) where C is a configuration, and D and A are sets of events. Intuitively, C represents the current execution being explored. Observe that at any time, the algorithm only explores a single configuration/execution similarly to DPORs. For those familiar with the DPOR algorithms [FG05, AAJS14], the intuition for the events in D is that they are events which have been disabled (i.e. their associated transitions have been added to the sleep sets) and the events in A*

Algorithm 1: Basic Unfolding-based DPOR Exploration.

```
1 Initially call Explore(∅, ∅).
2 Procedure Explore(C, D)
3   if ena(C) = ∅
4     return
5   Choose e from ena(C,D)
6   Explore(C ∪ {e}, D)
7   Explore(C, D ∪ {e})
```

correspond to branches of the wake-up tree in [AAJS14]. These are vital to guarantee that the exploration is optimal. The root node is the triple $(\varnothing, \varnothing, \varnothing)$ *on the top right. This tree corresponds to an exploration of the unfolding in Figure 4.7 that explores the M-traces in the order presented in Figure 4.6. At a given node, following the left child always adds events to the current configuration. This corresponds to the process of executing the program to obtain a maximal execution. A right child of a node denotes that the event previously added to the configuration at this node is conflicting with some other event. Intuitively, this means that there is a race which can be reversed and will lead to a new M-trace previously unexplored.*

The algorithm is optimal in the sense that all leaves of the tree are distinct maximal configurations. Note that all configurations explored in a node's left and right subtrees are different. ◁

4.2.1 Basic Unfolding-based DPOR Exploration

A simple exploration algorithm is given in Algorithm 1. This algorithm will serve as the basis for our main contribution.

The idea of Algorithm 1 is to explore from a configuration C all maximal configurations that avoid a set D of extensions of C. Intuitively, this algorithm should be seen as DPOR where D is the analogue of sleep sets on event structures. The exploration avoids visiting events in D which can be enabled at the configuration C by first checking if $\texttt{ena}(C,D) \mathrel{\hat{=}} en(C) \setminus D$ contains an event enabled at C and not in D (Line 4). No such event exists when:

1. C is a maximal configuration, i.e., there is no enabled event, or
2. C is not maximal but $en(C) \subseteq D$, i.e., all possible events that could be added to C have already been explored.

If such event e exists, we recursively explore all maximal configurations C' such that $C \cup \{e\} \subseteq C'$ and $C' \cap D = \varnothing$ (Line 6). Finally, we assume e is a conflicting extension of some configuration that contains C. Thus, we explore the maximal configurations C' such that $C \subseteq C'$ and $C' \cap (D \cup \{e\}) = \varnothing$ (Line 7). Since we start from the empty configuration and we do not exclude any event, we are bound to explore all maximal configurations. An important assumption in Algorithm 1 is that the extensions of a configuration are known during the exploration. The situation where the algorithm backtracks when C is not a maximal execution corresponds to the *sleep-set blocked executions* presented in [AAJS14].

We now informally argue why all maximal configurations can be reached before formally proving this result and other properties that will be useful for the rest of the section. Suppose we want to find a maximal configuration $C \hat{=} \{e_1, e_2, \ldots, e_n\}$ and we are in some call to `Explore` (C_1, D_1). It can be that $D_1 \cap C = \varnothing$ or $D_1 \cap C \neq \varnothing$.

If $D_1 \cap C = \varnothing$, either $C_1 \subseteq C$ or $C_1 \nsubseteq C$. If $C_1 \subseteq C$, we can pick an event $e \in C$. If that is the case, we will call `Explore` $(C_1 \cup \{e\}, D_1)$ where $C_1 \cup \{e\}$. Since configurations are finite, we will reach such C. If we do not pick an event $e \in C$, then we will eventually call `Explore` $(C_1, D_1 \cup \{e\})$. However, some $e \in C$ is still enabled and can be picked in this call. Since, we can only add a finite number of events to D, we will reach some call to `Explore` (C_1, D'_i) where the only event enabled not in D'_i will be $e \in C$. Thus, C will be found from the call to `Explore` $(C_1 \cup \{e\}, D'_i)$. If $C_1 \nsubseteq C$, let $C_i = C_1 \cap C$. Since $C_i \subseteq C_1$, it follows that $ext(C_i) \subseteq ext(C_1)$. Therefore there was a call to `Explore` (C_i, D_i) where $D_i \subseteq D_1$ and $D_i \cap C = \varnothing$. At this call, we are in the situation where $C_i \subseteq C$ and $D_i \cap D = \varnothing$ which we have already considered.

If $D_1 \cap C \neq \varnothing$, it means that at some point in the past there was a call to `Explore` $(C'_1, D'_1 \cup \{e\})$ where $e \in C$ and $D'_1 \cap C = \varnothing$. Now, it clear that there was a previous call to `Explore` (C'_1, D'_1). We have already showed above that from such call we can reach the configuration C. Therefore, the configuration C has already been explored previously to `Explore` (C_1, D_1) or it will be in its future.

Correctness We now formally state the main properties of Algorithm 1. Informally, for a given unfolding $\mathcal{U}$, Algorithm 1 terminates and it is sound, i.e. explores all maximal configurations. Furthermore, we will show that the algorithm explores every maximal configuration only once. This does not mean that the algorithm is optimal [AAJS14], as it can get *stuck* by returning in Line 4 when its first argument is not a maximal configuration.

A key idea in our exploration strategy is based on the following fact between maximal configurations.

Lemma 12. *Let C_1, C_2 be two maximal configurations of $\mathcal{U}$. If $C_1 \neq C_2$, then $C_1 \cap \mathit{cext}(C_2) \neq \emptyset$.*

Proof. Assume that C_1 and C_2 are two maximal configurations such that $C_1 \neq C_2$. It is clear that $C_1 \not\subseteq C_2$ as if that was the case then $\mathit{en}(C_1) \neq \emptyset$ and by definition C_1 would not be a maximal configuration. Let $C = C_1 \cap C_2$. It holds that there are two events $e_1, e_2 \in \mathit{en}(C)$ such that $e_1 \in C_1$ and $e_2 \in C_2$. Furthermore, it holds that $e_1 \notin C_2$ and $e_2 \notin C_1$. It follows that $e_1 \#^i e_2$ and $e_1 \in \mathit{cext}(C_2)$. Thus $C_1 \cap \mathit{cext}(C_2) \neq \emptyset$. □

A consequence of Lemma 12 is that the conflicting extensions of two distinct maximal configurations are incomparable.

Lemma 13. *Let C_1, C_2 be two maximal configurations of $\mathcal{U}$. If $C_1 \neq C_2$, then exists $e_1 \in \mathit{cext}(C_1)$ and $e_2 \in \mathit{cext}(C_2)$ such that $e_1 \notin \mathit{cext}(C_2)$ and $e_2 \notin \mathit{cext}(C_1)$.*

Proof. Assume that C_1 and C_2 are two maximal configurations such that $C_1 \neq C_2$. By Lemma 12, we have that $C_1 \cap \mathit{cext}(C_2)$ and $C_2 \cap \mathit{cext}(C_1)$ are non-empty. Let $e_1 \in C_2 \cap \mathit{cext}(C_1)$ and $e_2 \in C_1 \cap \mathit{cext}(C_2)$. Since an event $e \in C$ in some configuration cannot be part of its extensions, it holds that $e_1 \notin \mathit{cext}(C_2)$ and $e_2 \notin \mathit{cext}(C_1)$. □

We start by formalizing the call graph explored by the `Explore` procedure which is fundamental in the proofs. We will show that it is a binary tree where the child relations are given by the recursive calls.

Every call to `Explore` (C, D) issues two recursive calls. We define the *call graph* generated by Algorithm 1 as a directed graph $\mathcal{G} \mathrel{\hat{=}} (N, \triangleright)$ representing the exploration of the algorithm. As the algorithm is non-deterministic, different executions can generate different graphs.

The nodes N of the call graph are pairs of the form (C, D), where C, D are the parameters of the function `Explore` $(\cdot, \cdot)$. More formally, N contains exactly all tuples (C, D) satisfying:

- C, D are sets of events of the unfolding $\mathcal{U}$;
- during the execution of `Explore` $(\emptyset, \emptyset)$, the function `Explore` $(\cdot, \cdot)$ has been recursively called with C, D as, respectively, first and second argument;

The edge relation of the call graph, $\triangleright \subseteq N \times N$, represents the sequence of calls made by `Explore(·,·)`. Formally, it is the union of two disjoint relations $\triangleright \mathrel{\hat{=}} \triangleright_l \uplus \triangleright_r$, defined as follows:

$$(C, D) \triangleright_l (C^l, D^l) \qquad \text{and that} \qquad (C, D) \triangleright_r (C^r, D^r)$$

iff the execution of `Explore(`C, D`)` issues a recursive call to, resp., `Explore(`C^l, D^l`)` at Line 6 and `Explore(`C^r, D^r`)` at Line 7. Observe that C^l and C^r will necessarily be different (as $C^l = C \cup \{e\}$, where $e \notin C$, and $C^r = C$). Therefore, the two relations are disjoint sets. We distinguish the node

$$n_0 \mathrel{\hat{=}} (\varnothing, \varnothing)$$

as the *root node*.

Since $\mathcal{U}$ contains a finite number of events and the recursive calls monotonically increase either the first or the second argument, it follows that all paths in the graph are finite.

Lemma 14. *Let $n \mathrel{\hat{=}} (C, D)$ and $n' \mathrel{\hat{=}} (C', D')$ be two nodes of the call graph such that $n \triangleright n'$. Then*

- $C \subseteq C'$ *and* $D \subseteq D'$; *(3)*
- *if* $n \triangleright_l n'$, *then* $C \subset C'$; *(4)*
- *if* $n \triangleright_r n'$, *then* $D \subset D'$. *(5)*

Proof. Let e be the event chosen at Line 5 in the call `Explore(`C,D`)`. If $n \triangleright_l n'$, then $C' = C \cup \{e\}$ and $D' = D$. If $n \triangleright_r n'$, then $C' = C$ and $D' = D \cup \{e\}$. In both cases, all three statements hold. □

Lemma 15. *Any path $n_0 \triangleright n_1 \triangleright n_2 \triangleright \ldots$ in the call graph starting from n_0 is finite.*

Proof. For $i \geq 0$, let $n_i \mathrel{\hat{=}} (C_i, D_i)$. Note that $\mathcal{U}$ has a finite number of events. Also, whenever `Explore(·,·)` makes a recursive call at Line 6 it adds one event to C_i, as stated by Equation (4.4). Thus, the number of times that C_i and C_{i+1} are related by $\triangleright_l$ is finite. More formally, the set

$$L \mathrel{\hat{=}} \{i \in \mathbb{N} \mid C_i \triangleright_l C_{i+1}\}$$

is finite. As a result it has a maximum, and its successor $k \mathrel{\hat{=}} 1 + \max_<(L)$ is an index in the path such that for all $i \geq k$ we have $C_i \triangleright_r C_{i+1}$, i.e., the function only makes recursive calls at Line 7.

It follows that for $i \geq k$, we have that $C_i = C_k$, Observe that, as a result of Equation (4.5), the sequence

$$D_k \subset D_{k+1} \subset D_{k+2} \subset \dots$$

is also finite as there is a finite number of events to add to D. Hence, every path in the graph is finite. □

Lemma 15 is used to formally prove termination of Algorithm 1.

Theorem 6 (Termination)**.** *Let $\mathcal{U}$ be the finite unfolding of a system M with independence relation $\diamond$. Algorithm 1 on $\mathcal{U}$ terminates.*

Proof. Let $\mathcal{G} \triangleq (N, \triangleright)$ be the call graph generated by Algorithm 1. `Explore(C,D,A)` two recursively calls to `Explore`. Observe that there is no loop in Algorithm 1. Therefore, any non-terminating execution of Algorithm 1 must perform a non-terminating sequence of recursive calls, which entails the existence of an infinite path in $\mathcal{G}$. It is clear that $\mathcal{G}$ is finite as there is a finite number of events and each node of $\mathcal{G}$ is a tuple of sets of events. Since $\mathcal{G}$ is finite, for a path to be infinite it must contain a cycle. Since, by Lemma 15, no infinite path exist in the call graph, Algorithm 3 always terminates. □

We now show that the call graph is a finite binary tree where $\triangleright_l$ and $\triangleright_r$ are, respectively, the *left-child* and *right-child* relations. The following general lemma states useful invariants towards that result.

Lemma 16. *Let $(C, D) \in N$ be a node of the call graph. It holds that:*

- *C is a configuration;* *(6)*
- *$D \subseteq ext(C)$;* *(7)*

Proof. We will show Equations (4.6) and (4.7) by induction on the length $k \geq 0$ of any path

$$n_0 \triangleright n_1 \triangleright \dots \triangleright n_{k-1} \triangleright n_k$$

on the call graph, starting from the initial node and leading to $n_k \triangleq (C, D)$. For $i \in \{0, \dots, k\}$, let $n_i \triangleq (C_i, D_i)$.

We now show Equation (4.6).

Base case. $k = 0$. The empty set which is the element C at the root node is a configuration.

Step. Assume C_{k-1} is a configuration.

If $n_{k-1} \triangleright_l n_k$, then $C = C_{k-1} \cup \{e\}$ for some event $e \in en(C_{k-1})$, as specified in Line 5. By definition of $en(\cdot)$, C is a configuration.

If $n_{k-1} \triangleright_r n_k$, then $C = C_{k-1}$. By the induction hypothesis, C is a configuration. Thus, C is a configuration.

We now show Equation (4.7).

Base case. $k = 0$. Then $D = \varnothing$ and $D \subseteq ext(C)$ clearly holds.

Step. Assume that $D \subseteq ext(C)$ holds for (C_i, D_i) with $i \in \{0, \ldots, k-1\}$. We show that it holds for n_k. As before, we have two cases.

- Assume that $n_{k-1} \triangleright_l n_k$. We have that $D = D_{k-1}$ and that $C = C_{k-1} \cup \{e_{k-1}\}$. We need to show that for all $e' \in D$ we have $\lceil e' \rceil \subseteq C$ and $e' \notin C$. By induction hypothesis we know that $D = D_{k-1} \subseteq ext(C_{k-1})$, so clearly $\lceil e' \rceil \subseteq C_{k-1} \subseteq C$. We also have that $e' \notin C_{k-1}$, so we only need to check that $e' \neq e_{k-1}$. It follows directly from the definition of $\texttt{ena}(\cdot,\cdot)$ that $e_{k-1} \notin D$.

- Assume that $n_{k-1} \triangleright_r n_k$. We have that $D = D_{k-1} \cup \{e_{k-1}\}$, and by hypothesis we know that $D_{k-1} \subseteq ext(C_{k-1}) = ext(C)$. As for e_{k-1}, from Line 5 we know that $e_{k-1} \in en(C_{k-1}) = en(C) \subseteq ext(C)$. As a result, $D \subseteq ext(C)$.

□

From Lemma 16, we also know that for a node $(C, D) \in N$ where C is a maximal configuration we have that $D \subseteq cext(C)$.

Theorem 7 (Call Tree)**.** *The call graph $(N, \triangleright)$ is a finite binary tree, where $\triangleright_l$ and $\triangleright_r$ are respectively the* left-child *and* right-child *relations.*

Proof. It follows from Theorem 6 that the call graph is a finite directed acyclic graph. We now show that for every node $n \in N$, the nodes reached after the left child are different from those reached after the right one. Formally, we show that if

$$(C, D) \triangleright_l (C \cup \{e\}, D) \triangleright^* (C_1, D_1) \quad \text{and}$$
$$(C, D) \triangleright_r (C, D \cup \{e\}) \triangleright^* (C_2, D_2),$$

where e is the event chosen in Line 5, then $C_1 \neq C_2$. It follows from Equation (4.3), that $e \in D_2$ and $e \in C_1$. Since $D_2 \subseteq ext(C_2)$, by Equation (4.7), we have that $e \in ext(C_2)$, so $e \notin C_2$. □

The following result is the main lemma in proving that all maximal configurations are visited during the exploration.

Lemma 17. *For any node $n \mathrel{\hat{=}} (C, D) \in N$ in the call graph and any maximal configuration $\hat{C} \subseteq E$ of $\mathcal{U}$, if $C \subseteq \hat{C}$ and $D \cap \hat{C} = \varnothing$, then there is a node $n' \mathrel{\hat{=}} (C', \cdot) \in N$ such that $n \rhd^* n'$, and $\hat{C} = C'$.*

Proof. Consider the set $X = \hat{C} \setminus C$ of events in the maximal configuration and not in C. If $X = \varnothing$, then $\hat{C} = C$, we have trivially found such node. If $X \neq \varnothing$, consider an event $e \in \min_X$. It follows that $e \in en(C)$, $e \in \hat{C}$ and therefore, $e \notin D$. Thus, $e \in \texttt{ena}\,(C,D)$ and can be selected at Line 5. Since Algorithm 1 is non-deterministic we have to consider both cases. If e is chosen, there is a path $n \rhd_l n_l$ where $n_l = (C \cup \{e\}, D)$. In such node n_l, we are in the same scenario as in the node n but closer to $\hat{C}$ by one event. Since $\hat{C}$ is finite, by applying the same argument we will reach the desired node. If e is not chosen, there is a path $n \rhd_r n_r$ where $n_r = (C, D \cup \{e'\})$ for some $e' \in \texttt{ena}\,(C,D)$ such that $e' \notin \hat{C}$. In such node n_r, we are in the same scenario as in node n where $e \in \texttt{ena}\,(C,D \cup \{e'\})$. Since all paths in the tree are finite, we cannot go right indefinitely and so we will eventually select such $e \in \hat{C}$. By applying the same reasoning a finite number of times, we will reach the desired node $(C', \cdot) \in N$ such that $\hat{C} = C'$. □

Theorem 8 (Soundness)**.** *Let C be a maximal configuration of $\mathcal{U}$.* `Explore(·,·)` *is called* once *with its first parameter being equal to C.*

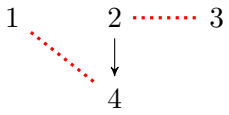
Proof. We first show that `Explore(·,·)` is called *at least once* with its first parameter being equal to C. This amounts to show that there is a path $n_0 \rhd n$ in the call graph between the root node n_0 and a node $n = (C, \cdot)$. This result follows immediately from Lemma 17 as initially we call `Explore(∅,∅)` and $\varnothing \subseteq C$, $C \cap \varnothing = \varnothing$.

We now show by contradiction that `Explore(·,·)` is called *at most once* with its first parameter being equal to C. Assume that there are two distinct paths from the root node n_0 to a node $n_1 \mathrel{\hat{=}} (C_1, D_1)$ and to node $n_2 \mathrel{\hat{=}} (C_2, D_2)$ where $C_1 = C_2$ and C_1 is a maximal configuration. It follows from Theorem 7 there n_1 and n_2 are distinct nodes and thus $D_1 \neq D_2$. Consider the set $D \mathrel{\hat{=}} D_1 \cap D_2$. It holds $D \cap C_1 = \varnothing$ and by Theorem 7 there is a unique path between n_0 and a node $n' \mathrel{\hat{=}} (C', D)$ where $C' \subseteq C_1$. This corresponds to the shared prefix between the two paths above. It follows that the left successor of n' will add an event $e \in C_1$ (or $C' = C_1$) and the right successor such event to D. Therefore the second path from such right successor will not visit C_1 and we reach a contradiction as $C_1 \neq C_2$. □

4.2.2 Stateful Optimal Exploration

In Section 4.2.1, we have shown that Algorithm 1 visits every maximal configuration only once. However, a call to `Explore(`C`,`D`)` is not optimal as it can return in Line 4 for a non-maximal configuration, i.e. the algorithm tries to explore a sleep-set blocked exploration. This happens when we add the event e chosen in Line 5 to D in the second recursive call but this event in part of every maximal configuration C' such that $C \subseteq C'$ and $C' \cap D = \varnothing$. However, as the next example shows, it is not sufficient for optimal explorations to know when to issue a second recursive call.

Example 27. *Consider the following prime event structure:*



Consider the call of `Explore(`$\varnothing$`,`$\varnothing$`)` *from Algorithm 1 that chooses event* 1 *at Line 5. After, it will recursively call* `Explore(`$\{1\}$`,`$\varnothing$`)` *which will visit all maximal configurations that contain the event* 1*. In this example, this corresponds to the configurations* $\{1,2\}$ *and* $\{1,3\}$*. Consider the second maximal configuration visited* $\{1,3\}$*. This will be represented by the node* $(\{1,3\},\{2\})$ *in the call graph whose predecessor is the node* $(\{1\},\{2\})$*. It is clear that at this stage* going right *in the call* `Explore(`$\{1\}$`,`$\{2,3\}$`)` *will return before finding another maximal configuration. Thus, the exploration does not always need to issue a second recursive call. At this stage, it remains to explore a third configuration* $\{2,4\}$ *which does not contain the event* 1*. Therefore, we must issue the second recursive call* `Explore(`$\varnothing$`,`$\{1\}$`)`*. However, at this call we are not guaranteed to visit such maximal configuration as we can choose the event* 3 *in Line 5.* ◁

As demonstrated in Example 27 to achieve an optimal exploration based on Algorithm 1 we need to understand: 1) when to issue a second recursive call to *avoid* the chosen event (together with D) and 2) guarantee that the next maximal configurations will necessarily avoid $D \cup \{e\}$. The main insight to achieve optimal explorations is based on two observations. First, once the exploration returns from the first recursive call `Explore(`$C \cup \{e\}$`,`D`)`, it has explored all maximal configurations $X \triangleq \{C_1, C_2, \ldots, C_n\}$ that contain $C \cup \{e\}$ with events that conflict with events in D. Consider some maximal configuration $C_i \in X$ already explored and a maximal configuration $\hat{C}$ not yet explored. We need to verify if the algorithm should issue a second recursive call by checking if $\hat{C}$ contains events that are conflicting with events

Algorithm 2: Optimal Exploration.

```
1  Initially call Explore(∅, ∅, ∅).
2  Procedure Explore(C, D, A)
3      if ena(C,D) = ∅ return
4      if A = ∅
5          Choose e from ena(C, D)
6      else
7          Choose e from A ∩ ena(C,D)
8      Explore(C ∪ {e}, D, A ∖ {e})
9      if ∃J ∈ Alt(C, D ∪ {e})
10         Explore(C, D ∪ {e}, J ∖ C)
```

in the set $D \cup \{e\}$. Second, We know (from Lemma 12) there exists $\hat{e} \in \hat{C}$ such that $\hat{e} \in \mathit{cext}(C_i)$. We already know that C_i is able to avoid D. That means that for each event $e' \in D$, there is an immediate conflict to e' in the set $C \cup A_i$.

Formally, we call such continuation an *alternative*:

Definition 21 (Alternatives)**.** *Given a set of events $U \subseteq E$, a configuration $C \subseteq U$, and a set of events $D \subseteq U$, an* alternative *to D after C is any configuration $J \subseteq U$ satisfying that*

- *$C \cup J$ is a configuration* *(8)*
- *for all events $e \in D$, there is some $\hat{e} \in C \cup J$ such that $\hat{e} \in \#^i_U(e)$.* *(9)*

■

In this subsection, we consider the set U in Definition 21 to be the set of events of the unfolding $\mathcal{U}$. Informally, alternatives correspond to the branches of the wake-up trees in [AAJS14].

We present an optimal exploration in Algorithm 2 using alternatives. As in Algorithm 1, `Explore`(C, D, A), the main procedure, is given the configuration that is to be explored as the parameter C and a set of D of events to avoid. The set A (for *add*) is occasionally used to force the exploration in a certain direction and avoid the events in D. Similarly to Algorithm 1, we assume that the entire unfolding is visible.

In Algorithm 2, a call to `Explore`(C, D, A) visits all maximal configurations of $\mathcal{U}$ which contain C and do not contain D; and the first one explored will contain $C \cup A$. As before, it first checks if `ena`$(C,D) \hat{=} \mathit{en}(C) \setminus D$ contains an event enabled at C and not in D. If no such event exists, it returns in Line 3. Otherwise (Line 4), `ena`$(C) \neq \emptyset$ and it selects an event enabled at C and not in D. If A is empty,

any event in $\texttt{ena}\,(C,D)$ can be taken (Line 5). If not, A needs to be explored first and e must come from the intersection $\texttt{ena}\,(C,D) \cap A$ (Line 7). Observe that as in Algorithm 1, this is a source of non-determinism in the exploration.

In Line 8, it recursively calls $\texttt{Explore}\,(C \cup \{e\}, D, A \setminus \{e\})$, where it explores *all* configurations that contain $C \cup \{e\}$ without events from D. After that, it remains to visit those containing C and not $D \cup \{e\}$ if and only if there exists one. Function `Alt` returns a set of alternatives that witness the existence of such configuration.

Definition 22 (`Alt` function)**.** *Function* `Alt` *denotes any function such that* $\texttt{Alt}\,(C, D)$ *returns a set* $X \subseteq \wp(\mathit{Event})$ *satisfying:*

1. *For all* $J \in X$*,* J *is an alternative to* D *after* C *and*
2. *If exists* $C' \in \mathit{conf}(\mathcal{U})$ *such that* $C \subseteq C'$ *and* $C' \cap D = \varnothing$*, then* $X \neq \varnothing$*.*

■

Note that in Line 9, it calls $\texttt{Alt}\,(C, D \cup \{e\})$. That is, J is a configuration that witnesses a maximal configuration that contains C without events from $D \cup \{e\}$. If such J exists, we issue the second recursive call $\texttt{Explore}\,(C, D \cap \{e\}, J \setminus C)$ in Line 10.

Correctness We now formally show that for a given unfolding $\mathcal{U}$, Algorithm 2 terminates and it is optimal [AAJS14], i.e., it explores every maximal configuration only once and it never gets *stuck*. Specifically, *not getting stuck* means that: 1) when returning at Line 3, the parameter C is a maximal configuration (there are no SSB explorations) and 2) whenever the algorithm reaches the Line 5 or Line 7 it is able to select an event e.

Similarly to the proofs of correctness for Algorithm 1, we start by formalizing the call graph explored by the `Explore` procedure. Every call to $\texttt{Explore}\,(C, D, A)$ issues up to two recursive calls. We define the *call graph* generated by Algorithm 2 as a directed graph $\mathcal{G} \mathrel{\hat{=}} (N, \triangleright)$ representing the exploration of the algorithm. Formally, N contains exactly all tuples (C, D, A) satisfying:

- C, D, and A are sets of events of the unfolding $\mathcal{U}$;
- during the execution of $\texttt{Explore}\,(\varnothing, \varnothing, \varnothing)$, the function $\texttt{Explore}\,(\cdot, \cdot, \cdot)$ has been recursively called with C, D, A as, respectively, first, second, and third argument;

The edge relation of the call graph, $\triangleright \subseteq N \times N$, represents the sequence of calls made by $\texttt{Explore}\,(\cdot,\cdot,\cdot)$. Formally, it is the union of two disjoint relations $\triangleright \mathrel{\hat{=}} \triangleright_l \uplus \triangleright_r$, defined as follows:

$$(C, D, A) \triangleright_l (C^l, D^l, A^l) \qquad \text{and that} \qquad (C, D, A) \triangleright_r (C^r, D^r, A^r)$$

iff the execution of $\texttt{Explore}\,(C, D, A)$ issues a recursive call to, resp., $\texttt{Explore}\,(C^l, D^l, A^l)$ at Line 8 and $\texttt{Explore}\,(C^r, D^r, A^r)$ at Line 10. Observe that C^l and C^r will necessarily be different (as $C^l = C \cup \{e\}$, where $e \notin C$, and $C^r = C$). Therefore, the two relations are disjoint sets. We distinguish the node

$$n_0 \mathrel{\hat{=}} (\varnothing, \varnothing, \varnothing)$$

as the *root node*. Note that as the algorithm is non-deterministic, different executions can generate different graphs.

We now show that if $\texttt{ena}\,(\cdot,\cdot)$ makes the same choices in Algorithm 1 and Algorithm 2, the graph generated by Algorithm 1 simulates the graph generated by Algorithm 2.

Lemma 18. *Let $\mathcal{G}_1 \mathrel{\hat{=}} (N_1, \triangleright_1)$ be the graph generated from Algorithm 1 and $\mathcal{G}_2 \mathrel{\hat{=}} (N_2, \triangleright_2)$ be the graph generated from Algorithm 2 over the unfolding $\mathcal{U}$ using the same result for $\texttt{ena}\,(\cdot,\cdot)$. It holds that if $(\varnothing, \varnothing, \varnothing) \triangleright_2^* (C, D, A)$, then $(\varnothing, \varnothing) \triangleright_1^* (C, D)$.*

Proof. Assume that there is a path $\pi \mathrel{\hat{=}} (\varnothing, \varnothing, \varnothing) \triangleright_2^* (C, D, A)$. The proof continues by induction on the length $k \geq 0$ of the path π.

Base case. $k = 0$. The node $(\varnothing, \varnothing) \in N_1$ as it is the root node.

Step. Assume $(C_{k-1}, D_{k-1}, A_{k-1}) \in N_2$ and $(C_{k-1}, D_{k-1}) \in N_1$. We have that either $(C_{k-1}, D_{k-1}, A_{k-1}) \triangleright_{l2} (C, D, A)$ or $(C_{k-1}, D_{k-1}, A_{k-1}) \triangleright_{r2} (C, D, A)$. Let e_1, e_2 be the event picked by both algorithms. We have that $e_1 = e_2$ as $\texttt{ena}\,(\cdot,\cdot)$ is called with the same arguments and $e_1, e_2 \in \texttt{ena}\,(C_{k-1}, D_{k-1})$ in both algorithms.

Assume that $(C_{k-1}, D_{k-1}, A_{k-1}) \triangleright_{l2} (C, D, A)$. Then we have that $C = C_{k-1} \cup \{e_2\}$ and $D = D_{k-1}$. In Algorithm 1, we always issue two recursive calls. Thus we also have $(C_{k-1}, D_{k-1}) \triangleright_{l1} (C_{k-1} \cup \{e_2\}, D_{k-1})$.

Finally, assume that $(C_{k-1}, D_{k-1}, A_{k-1}) \triangleright_{r2} (C, D, A)$. Then we have that $C = C$ and $D = D_{k-1} \cup \{e_2\}$. In Algorithm 1, we always issue two recursive calls. Thus we also have $(C_{k-1}, D_{k-1}) \triangleright_{lr} (C_{k-1}, D_{k-1} \cup \{e_2\})$.

Therefore, $(C, D) \in N_1$. □

Using Lemma 18, we know that the call graph obtained from a run of Algorithm 2 is a subtree of the tree obtained from Algorithm 1 considering considering the nodes

to be the first two elements. Thus, we know that corresponding versions of Lemmas 14 to 16 and Theorems 6 and 7 also hold for Algorithm 2.

We now focus on showing soundness and optimality of Algorithm 2 which relate to the new parameter A of `Explore` and the use of alternatives. The following lemma presents useful invariants regarding the parameter A.

Lemma 19. *Let $(C, D, A) \in N$ be a node of the call graph. It holds that:*

- *$C \cup A$ is a configuration and $C \cap A = \varnothing$;* *(10)*
- *for all $e \in D$ there is some $\hat{e} \in C \cup A$ such that $e \#^i \hat{e}$;* *(11)*
- *if $A = \varnothing$, then $D \subseteq cext(C)$;* *(12)*
- *$A \cap D = \varnothing$.* *(13)*

Proof. We show Equations (4.10) and (4.11) by induction on the length $k \geq 0$ of any path

$$n_0 \rhd n_1 \rhd \ldots \rhd n_{k-1} \rhd n_k$$

on the call graph, starting from the initial node and leading to $n_k \mathrel{\hat{=}} (C, D, A)$. For $i \in \{0, \ldots, k\}$, let $n_i \mathrel{\hat{=}} (C_i, D_i, A_i)$.

We now show Equation (4.10).

Base case. $k = 0$. It follows that $C = \varnothing$ and $A = \varnothing$. Clearly $C \cup A$ is a configuration and $C \cap A = \varnothing$.

Step. Assume that $C_{k-1} \cup A_{k-1}$ is a configuration and that $C_{k-1} \cap A_{k-1} = \varnothing$. The proof continues by cases:

- Assume $n_{k-1} \rhd_l n_k$. If A_{k-1} is empty, then A is empty as well. Hence, $C \cup A$ is a configuration and $C \cap A$ is empty. If A_{k-1} is not empty, then $C = C_{k-1} \cup \{e\}$ and $A = A_{k-1} \setminus \{e\}$, for some $e \in A_{k-1}$, and we have

  $$C \cup A = (C_{k-1} \cup \{e\}) \cup (A_{k-1} \setminus \{e\}) = C_{k-1} \cup A_{k-1},$$

  which is a configuration. We also have that $C \cap A = C_{n-1} \cap A_{n-1}$, so $C \cap A$ is empty.

- Assume $n_{k-1} \rhd_r n_k$. It follows that $C = C_{k-1}$ and also $A = J \setminus C_{k-1}$ for some $J \in \texttt{Alt}\,(C_{k-1}, D \cup \{e\})$. From Equation (4.8) we know that $C_{k-1} \cup J$ is a configuration. As a result,

  $$C \cup A = C_{k-1} \cup (J \setminus C_{k-1}) = C_{k-1} \cup J,$$

and therefore $C \cup A$ is a configuration. Finally, by construction of A, we clearly have $C \cap A = \varnothing$.

We now show Equation (4.11).

Base case. $k = 0$. It follows that $D = \varnothing$ and the result holds.

Step. Assume Equation (4.11) holds for $(C_{k-1}, D_{k-1}, A_{k-1})$. We show that it holds for n_k. The proof continues by cases:

- Assume $n_{k-1} \rhd_l n_k$. It follows that $D = D_{k-1}$. As a result, from the induction hypothesis for any $e \in D$ there is some $\hat{e} \in C_{k-1} \cup A_{k-1}$ such that $e \#^i \hat{e}$. Since $C_{k-1} \cup A_{k-1} \subseteq C \cup A$, it follows that such e is also in $C \cup A$. Thus, Equation (4.11) holds for n_k.

- Assume $n_{k-1} \rhd_r n_k$. It follows that $C = C_{k-1}$ and $D = D_{k-1} \cup \{e_{k-1}\}$ where e_{k-1} is the event chosen in the conditional of Line 4. Let $J \in \texttt{Alt}(C, D)$ be the alternative such that $A = J \setminus C$. Partition D into the pair (D_b, D_a) where $D_b = \{e \in D \mid e \in \mathit{cext}(C)\}$ and $D_a = D \setminus \mathit{cext}(C)$. It is clear that events in D_b will be in immediate conflict with events in $C \cup A$ for any set A as they are already in immediate conflict with some event in C. Thus it remains to show that for all $e \in D_b$ there is an event $\hat{e} \in A$ such that $e \#^i \hat{e}$. This follows immediately from Definition 21, since $e \in A$ only if $e \in J$ and we have that for $e \in D_b$, exists $\hat{e} \in J$ with $\hat{e} \#^i e$.

We now show Equation (4.12).

By Equation (4.7) we know that $D \subseteq \mathit{ext}(C)$. Assume $A = \varnothing$. For each $e \in D$ we need to prove the existence of some $\hat{e} \in C$ with $\hat{e} \#^i e$. This follows immediately from Equation (4.11).

Finally, we show Equation (4.13) by contradiction.

Assume that exists $e \in A \cap D$. It follows from Equation (4.10) that $C \cup A$ is a configuration. Furthermore, we know from Equation (4.11) that there is an event $\hat{e} \in C \cup A$ such that $e \#^i \hat{e}$. Therefore, we reach a contradiction as $C \cup A$ is not a configuration. □

We now show the analog of Lemma 17 for Algorithm 2.

Lemma 20. *For any node $n \hat{=} (C, D, A) \in N$ in the call graph and any maximal configuration $\hat{C} \subseteq E$ of $\mathcal{U}$, if $C \subseteq \hat{C}$ and $D \cap \hat{C} = \varnothing$, then there is a node $n' \hat{=} (C', \cdot, \cdot) \in N$ such that $n \rhd^* n'$, and $\hat{C} = C'$.*

Proof. Consider the set $X = \hat{C} \setminus C$ of events in the maximal configuration and not in C. If $X = \varnothing$, then $\hat{C} = C$, we have trivially found such node. If $X \neq \varnothing$, consider an event $e \in \min_X$. It follows that $e \in en(C)$, $e \in \hat{C}$ and therefore, $e \notin D$.

Thus, $e \in \texttt{ena}\,(C,D)$ and can be selected in the conditional Line 4. The proof continues by cases.

- Assume $e \in \hat{C}$ is chosen. This could be either when $A = \varnothing$ or $e \in A$. If e is chosen, there is a path $n \rhd_l n_l$ where $n_l = (C \cup \{e\}, D, A \setminus \{e\})$. In such node n_l, we are in the same scenario as in the node n but closer to $\hat{C}$ by one event.

- Assume that $e \in \hat{C}$ is not chosen but some other event $e' \in \texttt{ena}\,(C,D)$ and $e' \notin \hat{C}$. Again, this could occur when $A = \varnothing$ or $e \notin A$. We will now show that there is a path $n \rhd_r n_r$ where $n_r = (C, D \cup \{e'\}, A')$. This immediately holds by the Definition 22 as there exists at least one $J \in \texttt{Alt}\,(C, D \cup \{e'\})$ because $\hat{C}$ is a configuration such that $\hat{C} \subseteq C$ and $\hat{C} \cap (D \cup \{e'\}) = \varnothing$. Note that $\texttt{Alt}\,(C, D \cup \{e'\})$ can return more than one J to reach other maximal configurations. Nevertheless, in such node n_r, we are in the same scenario as in node n where $e \in \texttt{ena}\,(C,D \cup \{e'\})$ but we know that we cannot indefinitely *go right* and we will eventually pick an alternative that forces $\hat{C}$.

Since $\hat{C}$ is finite, by applying this argument a finite number of times, we will reach the desired node $(C', \cdot, \cdot) \in N$ such that $\hat{C} = C'$. □

Theorem 9 (Soundness)**.** *Let C be a maximal configuration of $\mathcal{U}$.* $\texttt{Explore}\,(\cdot,\cdot,\cdot)$ *is called* at least once *with its first parameter being equal to C.*

Proof. This amounts to show that there is a path $n_0 \rhd n$ in the call graph between the root node n_0 and a node $n = (C, \cdot, \cdot)$. This result follows immediately from Lemma 20 as initially we call $\texttt{Explore}\,(\varnothing,\varnothing,\varnothing)$ and $\varnothing \subseteq C$, $C \cap \varnothing = \varnothing$. □

We now show that Algorithm 2 is optimal.

Theorem 10. *Algorithm 2 is optimal in the sense that:*

1. *If C be a maximal configuration of $\mathcal{U}$, then* $\texttt{Explore}\,(C,\cdot,\cdot)$ *is called* at most once. *(14)*

2. *Only returns when it reaches a maximal configuration. Let (C, D, A) be a node of the call tree.* $\texttt{ena}\,(C,D) = \varnothing$ *iff C is a maximal configuration. (15)*

3. *If it reaches Line 4, then it will reach Line 8. Let* (C, D, A) *be a node of the call tree. If* $\texttt{ena}(C) \neq \varnothing$ *and* $A \neq \varnothing$*, then* $A \cap \texttt{ena}(C) \neq \varnothing$. *(16)*

Proof. We start by showing Equation (4.14) by contradiction. Assume that there are two distinct paths from the root node n_0 to a node $n_1 \hat{=} (C_1, D_1, A_1)$ and to node $n_2 \hat{=} (C_2, D_2, A_2)$ where $C_1 = C_2$ and C_1 is a maximal configuration. It follows from Lemma 18 and Theorem 7 there n_1 and n_2 are distinct nodes and thus $D_1 \neq D_2$. Consider the set $D \hat{=} D_1 \cap D_2$. It holds $D \cap C_1 = \varnothing$ and by Theorem 7 there is a unique path between n_0 and a node $n' \hat{=} (C', D, A)$ where $C' \subseteq C_1$. This corresponds to the shared prefix between the two paths above. It follows that the left successor of n' will add an event $e \in C_1$ to reach C_1 (or $C' = C_1$) and the right successor will add e to D to reach C_1 again. Therefore the second path from such right successor will not visit C_1 and we reach a contradiction as $C_1 \neq C_2$.

We now show Equation (4.15).

($\Rightarrow$) If $\texttt{ena}(C, D) = \varnothing$, then either $en(C) = \varnothing$ or $en(C) \subseteq D$. The proof continues by cases:

- Assume $en(C) = \varnothing$. By definition of maximal configuration, it follows that C is maximal.
- Assume $en(C) \subseteq D$ and that $en(C) \neq \varnothing$. We now show that we reach a contradiction since $en(C) \nsubseteq D$. Note that $D \subseteq ext(C)$ by Equation (4.7). The proof continues by cases:
  - Assume $A = \varnothing$. It follows from Equation (4.12) that $D \subseteq cext(C)$. Thus,there is an event $e \in en(C)$ such that $e \notin D$.
  - Assume $A \neq \varnothing$. Let $A' \hat{=} A \cap en(C)$ and $D' \hat{=} D \cap en(C)$. We know from Equation (4.10) that $C \cup A$ is a configuration, thus $A' \neq \varnothing$, and from Equation (4.13) that $A \cap D = \varnothing$. Thus, $en(C) \nsubseteq D$.

($\Leftarrow$) If C is maximal, then $en(C) = \varnothing$. It follows that $\texttt{ena}(C, D) = \varnothing$.

Finally, we show Equation (4.16). Assume that $\texttt{ena}(C) \neq \varnothing$ and $A \neq \varnothing$. Let $A' \hat{=} A \cap en(C)$. We know from Equation (4.10) that $C \cup A$ is a configuration, thus $A' \neq \varnothing$. Furthermore, we know from Equation (4.13) that $A \cap D = \varnothing$. Therefore, there exists an event $e \in A'$ such that $e \in en(C)$ and $e \notin D$. Thus, $A \cap \texttt{ena}(C) \neq \varnothing$. □

Parameter A of `Explore` plays a central role in the enforcing the second item in Theorem 10. It is necessary to ensure that, once the algorithm decides to explore some alternative J, such an alternative is visited first. Not doing so makes it possible

Algorithm 3: Stateless Optimal Exploration.

```
1  Initially, let U ≔ ∅, G ≔ ∅, and call Explore(∅, ∅, ∅).
2  Procedure Explore(C, D, A)
3      Extend(C)
4      if ena(C) = ∅
5          Inspect(C)
6          return
7      if A = ∅
8          Choose e from ena(C)
9      else
10         Choose e from A ∩ ena(C)
11     Set C(e), D(e), A(e) to C, D, ∅
12     Explore(C ∪ {e}, D, A ∖ {e})
13     if ∃J ∈ FilterAlt(C, D ∪ {e})
14         Explore(C, D ∪ {e}, J ∖ C)
15     Remove(C, D)
16 Procedure Extend(C)
17     Add ext(C) to U
18 Procedure Inspect(C)
19     foreach e ∈ C and e′ ∈ #^i_{cext(C)}(e)
20         Let J ≔ C ∖ #(e′) ∪ {e′}
21         if J ∈ Alt(C(e), D(e))
22             Add J to A(e)
23 Procedure Remove(C, D)
24     Move U ∖ Q_{C,D} from U to G
25 Procedure FilterAlt(C, D)
26     foreach e ∈ D
27         foreach J ∈ A(e)
28             if J ∈ Alt(C,D)
29                 return J
30     return ∅
```

to extend C in such a way that no maximal configuration can ever avoid including events in D as shown in Example 27.

4.2.3 Stateless Optimal Exploration

In Section 4.2.2, we have shown that Algorithm 2 performs an optimal exploration of the unfolding $\mathcal{U}$. However, the algorithm assumes that $\mathcal{U}$ is pre-computed and visible during the exploration as `Alt` can identify alternatives based on the entire unfolding. In this section, we remove such assumption and present a stateless exploration of the unfolding that only keeps a small portion of the structure that has already been explored. The main technical contributions of this algorithm are related to procedures that extend prime event structures with events based on the execution being explored and compute the set of alternatives to guarantee optimality. We consider the algorithm to be stateless as it does not require keeping the entire set of events already explored despite having to keep a potential large number of alternatives. This is similar to the claim made in [AAJS14] regarding the ODPOR algorithm and the potential size of the wake-up trees. Our experimental observations show however, that in practice, the number of alternatives to achieve optimality is low.

The stateless exploration algorithm is given in Algorithm 3. `Explore`(C, D, A), the main procedure, has the same parameters as in Algorithm 2 where:

- C is the configuration to be explored;
- D is the set of set of events that have already been explored and we would like to avoid;
- A is a set of events that is used to force the exploration in a certain direction and avoid the events in D.

The main difference between both algorithms is that now `Explore`(C, D, A) operates over a global set U that represents all events of $\mathcal{U}$ in scope, i.e., already discovered and not discarded. Whenever some event is shown not to be necessary for the exploration to continue, `Remove` will move it from U to G (for *garbage*). Once in G, it can be discarded at any time, or be preserved in order to save work when it is re-inserted in U. Note that an event can be discovered multiple times.

The key intuition in Algorithm 3 is still the same as in Algorithm 2 – a call to `Explore`(C, D, A) visits all maximal configurations of $\mathcal{U}$ which contain C and do not contain D; and the first one explored will contain $C \cup A$.

Example 28. *Consider the execution of algorithm presented in Figure 4.8. Each node $C \mid D \mid A$ represents a call to* `Explore`(C,D,A) *where C is a configuration. The terminal nodes represent calls where the configuration is maximal. Given some non-terminal node, $C \mid D \mid A$ we can always reach a maximal configuration $C' \supset C$ by a taking finite sequence of left children. A key property of the algorithm is that we will always visit configurations that do not contain events in D. The set A is fundamental to achieve this property since the algorithm chooses the exploration to consider events in A first to guarantee that we will avoid D. Thus, if A is not-empty, the left child of the node will be forced to take an event from A. An invariant of the algorithm is that A will always contain enabled events so there is no possible way to get stuck when A is not empty. Furthermore, once A is empty, it will guarantee that all events in D are conflicting extensions of C. From that point it will not be able to add any event in D as the exploration only adds enabled events to the current configuration.*

In summary, given a node $C \mid D \mid A$ the left sub-tree will explore configurations $C \subset C'$ satisfying $C' \cap D = \varnothing$ starting with some event e. The right sub-tree will explore configurations $C \subset C'$ satisfying $C' \cap (D \cup \{e\}) = \varnothing$. Note that an event e can be explored multiple times. ◁

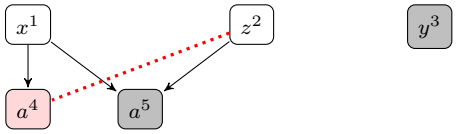
The algorithm first extends U with respect to the configuration C (Line 3). For now, this amounts to inserting $ext(C) = en(C) \cup cext(C)$ (Line 17). Note that this involves inserting multiple events in U.

Example 29 (Extension)**.** *We now present several examples of the extensions of a configuration in Figure 4.8.*

The call `Extend(`$C_1 \hat{=} \varnothing$`)` *returns the prefix:*

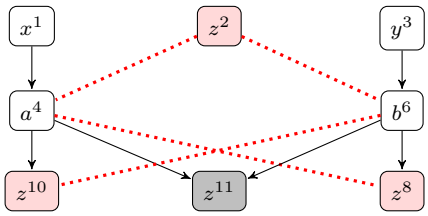
where the gray events are enabled events at C_1*.*

The call to `Extend(`$C_2 \hat{=} \{x^1, z^2\}$`)` *returns the prefix:*



where the gray events are enabled events and the red events are conflicting extensions at C_2*.*

The call to `Extend(`$C_3 \hat{=} \{x^1, y^3, a^4, b^6\}$`)` *returns the prefix:*



where the gray event z^{11} *is the only enabled event and the red events are conflicting extensions at* C_3*. The extension of a maximal configuration, e.g.* $C_3 \cup \{z^{11}\}$ *only adds the conflicting extensions, the red events.*

The numbering of events in Figure 4.8 is given by their order of discovery in the calls of `Extend` *in Algorithm 3.* ◁

Next, it checks if `ena(`C,D`)` $\hat{=} en(C) \setminus D$ contains an event enabled at C and not in D. Recall that no such event exists when:

1. C is a maximal configuration, i.e., there is no enabled event, or
2. C is not maximal but $en(C) \subseteq D$, i.e., all possible events that could be added to C have already been explored.

Before returning at Line 6, it calls the procedure `Inspect`. This procedure analyses the conflicting extensions of the input configuration to compute alternatives to other maximal configurations not yet explored.

Otherwise (Line 7), $\texttt{ena}(C) \neq \varnothing$ and it selects an event enabled at C and not in D. If A is empty, any event in $\texttt{ena}(C)$ can be taken (Line 8). If not, A needs to be explored first and e must come from the intersection $\texttt{ena}(C) \cap A$ (Line 10). Again, note that this is a source of non-determinism in the exploration.

In Line 12, it recursively calls $\texttt{Explore}(C \cup \{e\}, D, A \setminus \{e\})$ (left subtrees in Example 26), where it explores *all* configurations that are supersets of $C \cup \{e\}$ without events from D. After that, it remains to visit those containing C and not $D \cup \{e\}$ if and only if there exists one. In Line 9 of Algorithm 2, this situation was resolved by calling $\texttt{Alt}(C, D \cup \{e\})$ which had access to the entire unfolding i.e.,$U = E$. Furthermore, for each event in D, there might be many conflicting events in U which cannot be part of an alternative, so this procedure can be costly.

To avoid this problem, whenever the algorithm visits one maximal configuration C, the function `Inspect` will record the alternatives for events e of C. When it later backtracks to e, the alternatives found so far by `Inspect` are sufficient to decide whether the second recursive call is needed. We now explain this mechanism in more detail.

For a call $\texttt{Explore}(C, D, A)$, if the algorithm reaches Line 11, it associates to the chosen event e the following sets:

- $C(e)$ that represents the configuration C where e was inserted to U;
- $D(e)$ that represents the set D for the event e in the current call and
- $A(e)$ that represents a set of alternatives such that every $J \in A(e)$ satisfies $J \in \texttt{Alt}(C(e), D(e) \cup \{e\})$.

Specifically, we equip every event with a precomputed set of alternatives $\texttt{A}(e)$, generated by the function `Inspect` when visiting a maximal configuration containing e. Now, when Algorithm 3 evaluates the need of a second recursive call at Line 13, it calls $\texttt{FilterAlt}(C, D \cup \{e\})$. In this function, we select among the alternatives pre-computed for the events in D if there is one alternative that satisfies all events in D Line 28. If that is the case, we return it Line 29.

If an alternative is returned, a second recursive call at Line 14 (right subtrees in Example 26) uses the alternative to initially guide the exploration.

When Algorithm 3 returns from the second recursive call at Line 15, it has explored:

1. all maximal configurations that contain $C \cup \{e\}$ avoiding D and

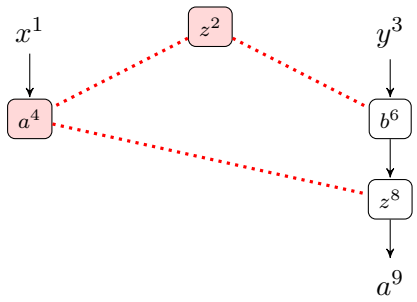
2. all maximal configurations that contain C avoiding $D \cup \{e\}$.

It has also pre-computed all necessary alternatives for the events $e' \in C$ to decide if `Explore`$(C(e'), D(e'), \cdot)$ needs to make a second recursive call.

Example 30 (Alternatives)**.** *Consider the node* $1,3,6 \mid 2 \mid \varnothing$ *in the exploration tree of Figure 4.8. Since* $A = \varnothing$*, we know that* z^2 *is a conflicting extension of* $C \doteq \{x^1, y^3, b^6\}$*. This is the case as* $b^6 \#^i z^2$*.*

The chosen event at Line 8 is z^8*. When the algorithm reaches Line 13, it needs to decide if there is a maximal configuration* $C' \supset C$ *such that* $C' \cap \{z^2, z^8\} = \varnothing$*. This maximal configuration starts with an alternative to* $\{z^2, z^8\}$ *after* C*.*

The algorithm only relies on the alternatives computed at the maximal configuration reached on the left sub-tree of C*. In this case, there is only one maximal configuration* $C_m \doteq \{x^1, y^3, b^6, z^8, a^9\}$ *whose state is* d_2*. Thus, if there is an alternative to* $\{z^2, z^8\}$ *after* C*, the* `Inspect` *procedure must find it at* C_m*. The prefix observed by* `Inspect` *is:*



`Inspect` *will consider events in* C_m *that have immediate conflicts. We have marked those events,* z^8 *and* b^6*, in the prefix with a box. We now analyze each event separately:*

- $e \doteq z^8, e' \doteq a^4$*: the next step, Line 20, constructs a potential alternative* $J \doteq C_m \setminus \#(e') \cup \{e'\}$*. In this case, we have* $J \doteq \{x^1, y^3, a^4\}$*. Then,in Line 21, it checks if* $J \in$`Alt`$(C(e), D(e))$*. In this case,* $C(e) \doteq \{x^1, y^3, b^6\}$ *and* $D(e) \doteq \{z^2\}$*. Therefore, we want to check if* J *is an alternative to* $\{z^2\}$ *after* $\{x^1, y^3, b^6\}$*. Since this is true,* `Inspect` *adds* J *to* $A(z^8)$ *(Line 22).*

- $e \doteq b^6, e' \doteq z^2$*: the potential alternative is* $J \doteq \{x^1, y^3, z^2\}$*. In the next step, it checks if* $J \in$ `Alt`$(\{x^1, y^3\}, \{z^2\})$*. This is not true as* $z^2 \in J$*. Therefore,* `Inspect` *does not add any alternative to* $A(b^6)$*.*

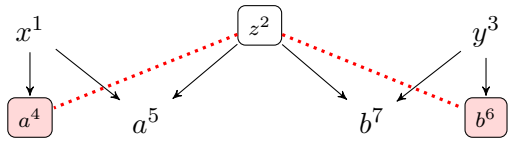
When the algorithm backtracks to the node $1,3,6 \mid 2 \mid \varnothing$*, it finds that it has an alternative to* z^8 *as* $A(z^8) = \{x^1, y^3, a^4\}$*. Therefore, it recursively calls* `Explore` $(C, D \cup \{e\}, J \setminus C)$ *which produces the right child* $1,3,6 \mid 2,8 \mid 4$ *in Figure 4.8.* ◁

Since `Alt` only considers any pre-computed alternative added to $\mathtt{A}(e')$ for $e' \in C$, `Remove` will preserve in U the following events:

$$Q_{C,D} \triangleq C \cup \bigcup_{e \in D} [e] \cup \bigcup_{e \in C} \mathtt{A}(e) .$$

Note that the chosen e can be part of those alternatives so the algorithm cannot always remove e when backtracking.

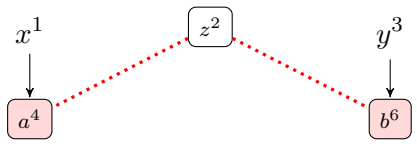
Example 31. *Consider the prefix* U *when the exploration reaches the first maximal configuration represented by the node* $1,2,3,5,7 \mid \varnothing \mid \varnothing$ *in Figure 4.8:*



At this point, `Inspect` *is going to add two possible alternatives to* $A(z^2)$*:*

1. $\{x^1, y^3, a^4\}$ *and*
2. $\{x^1, y^3, b^6\}$.

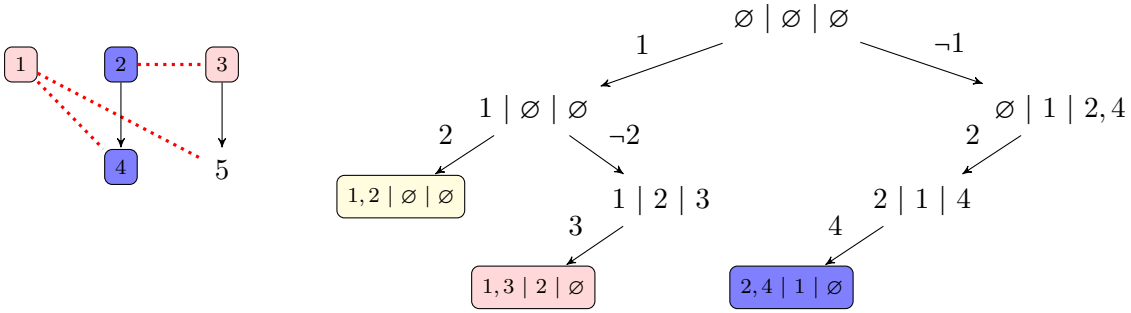
When the algorithm starts backtracking, the events b^7 *and* a^5 *can be safely removed from* U *as they are not part of any alternative. However, when backtracking from the node* $1,2,3 \mid \varnothing \mid \varnothing$ `Remove` *cannot remove the event* y^3 *as it is a part of an alternative. Thus, when the algorithm backtracks to the node* $1,2 \mid \varnothing \mid \varnothing$ *we have the following prefix:*



◁

Before we consider the correctness of Algorithm 3, we provide an example of why it is necessary to collect a set of alternatives for an event e in $A(e)$. In other words, we will show that it is possible that we need to issue a second recursive call adding the chosen event e to the set D even when $A(e) = \varnothing$.

Example 32. *Consider the following unfolding on the left and part of the call tree for Algorithm 3 on the right.*



This part of the call tree presented above corresponds to the exploration of the first three maximal configurations when the algorithm chooses enabled events with the smallest number. Once the algorithm reaches the first maximal configuration denoted in yellow, it adds $\{3\}$ *to* $A(2)$ *and also* $\{2,4\}$ *to* $A(1)$*. This is the case since at this stage we have not discovered event* 5 *(as it is not a conflicting extension of* $\{1,2\}$*). Next, the algorithm backtracks and finds that* $A(2) \in \texttt{Alt}\,(\{1\},\{2\})$*. Thus, there is a second recursive call which will lead to the discovery of the second maximal configuration in red. At this stage, there is another alternative for event* 1*. Namely,* `Inspect` *adds* $\{3,5\}$ *to* $A(1)$*. Once the algorithm backtracks to the root node, it finds that there are two possible values for* A *to issue a second recursive call. This is another source of non-determinism in the algorithm. Assume that calls* $\texttt{Explore}\,(\varnothing,\{1\},\{2,4\})$*. This will lead to the discovery of the third maximal configuration in blue. At this stage, the algorithm calls* `Inspect` *which only uses the maximal configuration and the conflicting extensions to find alternatives. In particular,* `Inspect` *sees that both events* 2 *and* 4 *are in immediate conflict with some conflicting extension:*

- $e \mathrel{\hat{=}} 4$ *and* $e' \mathrel{\hat{=}} 1$*. However,* $\{1\} \notin \texttt{Alt}\,(\{2\},\{1\})$*, so nothing is added to* $A(4)$ *and;*
- $e \mathrel{\hat{=}} 2$ *and* $e' \mathrel{\hat{=}} 3$*. However,* $\{3\} \notin \texttt{Alt}\,(\varnothing,\{1\})$ *since* 3 *is concurrent with* 1*.*

Thus, once we return from `Inspect` *we have not added any alternative to any event in the maximal configuration. However, we know that there is an alternative to* $\{1,2\}$ *after* $\varnothing$ *even though we have that* $A(2) = \varnothing$*! It is at this stage that we rely on previous computed alternatives to find such alternative. In particular, we have that* $\{3,5\} \in A(1)$ *which is such alternative.* ◁

Correctness In this section, we show that Algorithm 3 has the same properties than Algorithm 2, i.e. it terminates and performs an optimal exploration. In the sequel, we consider the *call graph* (as defined for Algorithm 2) generated by Algorithm 3 as a directed graph $\mathcal{G} \hat{=} (N, \triangleright)$ representing the exploration of the algorithm.

The only result that does not hold from Algorithm 2 is *soundness* as it is not clear if we keep enough information to visit all maximal configurations. This is because alternatives are computed only using maximal configurations instead of the complete unfolding.

We now show that once Algorithm 3 reaches Line 13, it holds that $\texttt{Alt}\,(C, D \cup \{e\}) \neq \varnothing$ iff $\texttt{FilterAlt}\,(C, D \cup \{e\}) \neq \varnothing$ regardless of the contents of G.

Lemma 21. *Let $n \hat{=} (C, D, A) \in N$ be a node of the call graph. If $(C, D, A) \triangleright_r (C, D \cup \{e\}, A')$ then $A' \in \texttt{Alt}\,(C, D \cup \{e\})$.*

Proof. Assume that $\texttt{Explore}\,(C, D, A) \triangleright_r \texttt{Explore}\,(C, D \cup \{e\}, A')$. It follows that $A' = J \setminus C$ where $J \neq \varnothing$. Furthermore, we know that if $J \neq \varnothing$, then it was returned at Line 29. Thus, $J \in \texttt{Alt}\,(C, D \cup \{e\})$. It follows from Definition 21 that we also have that $J \setminus C \in \texttt{Alt}\,(C, D \cup \{e\})$. Thus, $A' \in \texttt{Alt}\,(C, D \cup \{e\})$. □

Lemma 22. *Let $n \hat{=} (C, D, A) \in N$ be a node of the call graph. It holds that $(C, D, A) \triangleright_r (C, D \cup \{e\}, A')$ iff $\texttt{Alt}\,(C, D \cup \{e\}) \neq \varnothing$.*

Proof.

($\Rightarrow$). Assume $(C, D, A) \triangleright_r (C, D \cup \{e\}, A')$. From Lemma 21, we have that if $(C, D, A) \triangleright_r (C, D \cup \{e\}, A')$ then $A' \in \texttt{Alt}\,(C, D \cup \{e\})$. Thus, $\texttt{Alt}\,(C, D \cup \{e\}) \neq \varnothing$.

($\Leftarrow$). Proof by structural induction.

Base case. C is a maximal configuration and $\texttt{Alt}\,(C, D \cup \{e\}) = \varnothing$. Then the statement is trivially true.

Step. Assume that the implication holds for every successor of n. Furthermore, assume that $\texttt{Alt}\,(C, D \cup \{e\}) \neq \varnothing$. We now show that $(C, D, A) \triangleright_r (C, D \cup \{e\}, A')$ is an edge of the call graph. It follows from the induction hypothesis and Lemma 21, that we have explored all maximal configurations $\mathcal{C}$ of the unfolding such that for all $C_i \in \mathcal{C}$, it holds that $C \cup \{e\} \subseteq C_i$ and $D \subseteq \mathit{cext}(C_i)$. This is the case since Algorithm 2 and Algorithm 3 will produce the same subtree assuming the same choice for the enabled event and alternatives. We also know that an alternative to $D \cup \{e\}$ after C is a continuation X of C,i.e. $C \cup X$ is a configuration, that only needs contains immediate

conflicts with events $\hat{e} \in D_C \cup \{e\}$ where $D_C \triangleq C \setminus cext(C)$. Furthermore, we know from Equation (4.7) that for all $e_1, e_2 \in D_C \cup \{e\}$ either $e_1 \parallel e_2$ or $e_1 \#^i e_2$.

Let $\hat{C}$ be a maximal configuration such that $C \subseteq \hat{C}$ and $D \cup \{e\} \subseteq cext(\hat{C})$. We know that such $\hat{C}$ exists since $\texttt{Alt}\,(C, D \cup \{e\}) \neq \varnothing$. Let $\hat{C}' = \hat{C} \setminus \#\,(e)$ be the configuration that removes conflicts of e from $\hat{C}$. At such configuration $\hat{C}'$ we know that: 1) e is an enabled event, 2) $C \subseteq \hat{C}'$, 3) $\hat{C}' \cap D = \varnothing$. It follows that all events $\bar{e} \in \hat{C}' \setminus C$ are concurrent with e. Thus, it will be the case that there will be at least one maximal configuration $C_i \in \mathcal{C}$ such that $\hat{C}' \cup \{e\} \subseteq C_i$. Denote that subset of $\mathcal{C}$ as $\mathcal{K}$.

At the configuration $\hat{C}'$, we only need to avoid $D_{\hat{C}'} \triangleq D_C \cap en(\hat{C}')$. Let $F \triangleq en(\hat{C}') \setminus D_{\hat{C}'}$ and $\hat{F} \triangleq F \setminus \{e\}$. We know that: 1) $e \in F$; 2) $\hat{F} \neq \varnothing$; 3)for all $e' \in \hat{F}$, $e' \#^i e$ and 4)for all $e_1, e_2 \in \hat{F}$, it holds that $e_1 \parallel e_2$. The proof continues by cases.

- If $D_{\hat{C}'} = \varnothing$, then we are done as it will be the case that $\hat{C}' \cup \{e'\} \in A(e)$.
- If $D_{\hat{C}'} \neq \varnothing$, then it holds that: 1) in the maximal configurations $\mathcal{K}$, the events that are in immediate conflict with events in $D_{\hat{C}'}$ are successors of e or e itself and 2) in the maximal configuration $\hat{C}$, the events that are in immediate conflict with events in $D_{\hat{C}'}$ are successors of some event $e' \in \hat{F}$ or is some event $e' \in \hat{F}$. If there is one event $e' \in \hat{F}$ that is in conflict with all events in $D_{\hat{C}'}$ we are also done as $\hat{C}' \cup \{e'\} \in A(e)$. Otherwise, consider the sequence σ of events added to $D_{\hat{C}'}$. We know that at some point i in this sequence there is some event e_i that is not in conflict with some event $\hat{e}_i \in \hat{F}$. However, we know that there is some alternative in $A(e_j)$ for $j \leq i$ that will avoid all events up to and including $\hat{e}_i$. This is because we know that e_j is not e and there is a right edge that adds e. Since $e_i \parallel \hat{e}_i$, we know that some maximal configurations will contain $\hat{e}_i$. Therefore, for the next event e_k after e_i, we know that there is some alternative that contains this event $\hat{e}_i$ which is also in conflict with e. Since there is a finite number of events in $D_{\hat{C}'}$ we are guaranteed to have $\hat{e}_i$ is one of alternatives to D. Thus, the algorithm will have seen enough conflicting extensions to compute alternatives which is exactly what we wanted to show.

□

Theorem 11 (Soundness)**.** *Let C be a maximal configuration of $\mathcal{U}$ and $\mathcal{G}$ be the call tree generated by Algorithm 3.* $\texttt{Explore}(\cdot,\cdot,\cdot)$ *is called* at least once *with its first parameter being equal to C.*

Proof. By Lemmas 21 and 22, whenever Algorithm 3 executes, it generates a call tree isomorphic to one that Algorithm 2 can generate. Thus, by Theorem 9, it follows that every maximal configuration will be visited at least once. □

4.3 Cutoff Theory

In this section, we remove the assumption that $\mathcal{U}$ is finite and show how Algorithm 3 can be turned into a stateful exploration using the notion of cutoff events [McM93a] without losing soundness and optimality of the exploration.

Example 33. *Consider the consumer-producer program in Example 19. Since the state space of this program contains cycles, there are infinite executions. For this reason, the unfolding of this program will be infinite. However, there events in the unfolding which have isomorphic* suffixes. *For example, producing two items in* `buf1` *and consuming one reaches the same state as only producing one. After the common state, both traces explore identical behaviors. If we consider only local state reachability, only one needs to be explored.* ◁

Note that we still assume that the state space is finite. While cutoffs are a standard tool for obtaining complete prefixes of Petri net unfoldings, their application to Algorithm 3 is not immediate when we allow the removal of certain key events from the unfolding prefix.

Preventing Algorithm 3 from getting stuck in an infinite exploration is not a fundamental problem in our setting. Since the state space is finite, we are bound to find a previous state already in the stack. The core question is how to combine a stateful exploration that keeps a record of the states already explored with an exploration that removes events.

We now formally define the notion of complete prefix.

Definition 23 (Complete Prefix)**.** *Let $\mathcal{U}$ be an unfolding of a system M with independence $\diamond$. We say that the prefix $\mathcal{P}$ of $\mathcal{U}$ is complete if:*

1. *For every reachable state $s \in Reach(M)$, there is a configuration $C \in conf(\mathcal{P})$ such that $st(C) = s$ and*
2. *If $t \in Trans$ is fireable, then there exists an event $e \in \mathcal{P}$ such that $h(e) = act(t)$.*

■

Definition 23 states that a complete prefix is a PES where all reachable states can be retrieved and every fireable transition is represented. In general, these two properties are orthogonal: if we one represents the set of reachable states we can miss some transition $t_b \hat{=} (s, b, t)$ if we already represent the transition $t_a \hat{=} (s, a, t)$. Furthermore, it is known that we can reduce global state reachability to the fireability of a transition by introducing a transition that is only enabled at the global state in question. Thus, we are aiming for an algorithm that explores a sufficient set of configurations where all fireable transitions of the system are represented via a set of events and using those events we can build a configuration for every reachable state.

We follow the standard practice in the application of cutoffs and create the *illusion* that maximal configurations are finite. We achieve this by substituting procedure `Extend` with another procedure `ExtendCut` that operates as `Extend` except that it only adds to U an event from $e \in ext(C)$ if the predicate `iscutoff(e,U,G)` evaluates to false.

For the rest of this section, assume that Algorithm 3 uses `ExtendCut` instead of `Extend`. We refer to this version as the *updated algorithm*.

We use *adequate orders* (from [ERV02]) between configurations, in the definition of `iscutoff`.

Let $\mathcal{U}$ be the unfolding of $M \hat{=} (State, Trans, \tilde{s})$. Consider a configuration $C \in conf(\mathcal{U})$. We know that the state of C is a singleton $\{s\}$ where $s \in State$. We denote the unfolding of the system $M_s \hat{=} (State, Trans, s)$ as $\Uparrow C$. Consider the isomorphism I_1^2 between $\Uparrow C_1$ and $\Uparrow C_2$ where $st(C_1) = st(C_2)$. Recall from Section 2.3.2 that $C \oplus E$ denotes that E is an extension of C.

Definition 24 (Adequate Order)**.** *A partial order $\prec \subseteq conf(\mathcal{U}) \times conf(\mathcal{U})$ on the finite configuration of $\mathcal{U}$ is an adequate order if:*

1. *$\prec$ is* well-founded*, i.e. every non-empty $S \subseteq conf(\mathcal{U})$ has a minimal element w.r.t. $\prec$;*
2. *$C_1 \subset C_2$ implies $C_1 \prec C_2$ and*
3. *$\prec$ is preserved by finite extensions; more formally, if $C_1 \prec C_2$ and $st(C_1) = st(C_2)$, then the isomorphism I_1^2 satisfies $C_1 \oplus E \prec C_2 \oplus I_1^2(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

■

For the rest of this section, let $\prec$ be an adequate order on the configurations of $\mathcal{U}$.

We define `iscutoff(`e,U,G`)` to hold iff there exists some event $e' \in U \cup G$ such that:

$$st([e]) = st([e']) \quad \text{and} \quad [e'] \prec [e]. \tag{4.17}$$

We refer to e' as the *corresponding* event of e, when it exists.

We now define the canonical prefix associated with $\prec$. Given an event $e \in E$, we call it $\prec$-*cutoff* iff there exists some other event $e' \in E$ such that Equation (4.17) holds. Observe that we now search e' in E and not in $U \cup G$. We denote by $\mathcal{P}_\prec$, the unique $\trianglelefteq$-maximal unfolding prefix that contains no $\prec$-cutoff.

It is known from [EH08] that:

1. the prefix $\mathcal{P}_\prec$ exists and is unique and
2. it is *complete* according to Definition 23.

Consider the set of terminal configurations explored by the updated algorithm, and let us denote them by

$$C_1, C_2, \ldots, C_n.$$

Let $\mathcal{P}_u \mathrel{\hat{=}} [\![E_u]\!]$ be the unique prefix of $\mathcal{U}$ whose set of events $E_u = \bigcup C_i$.

By definition of $\mathcal{P}_\prec$ we have that `iscutoff(`e,U,G`)` always returns *false* for every $e \in \mathcal{P}_\prec$. We make the following two key observations:

1. The set of maximal configurations explored by the original Algorithm 3 and the updated algorithm is the same for $\mathcal{P}_\prec$.
2. If the updated algorithm does not remove any event in G, then $\mathcal{P}_u = \mathcal{P}_\prec$.

Example 34. *In this example, we present two consequences from the fact that* `iscutoff` *is a function of U and G.*

- *Every configuration C_i is not necessarily a maximal configuration of $\mathcal{U}$ when we assume that $\mathcal{U}$ is the unfolding of a finite acyclic state space. For example, consider the system composed of two threads where the first two statements are considered dependent when in fact they are independent. There will be two configurations that reach the same state. The updated algorithm will be able to stop one of them and consider it maximal while it is not maximal in the original unfolding.*

- *Every configuration C_i is not necessarily a maximal configuration of $\mathcal{P}_\prec$. This can happen because an event e could be declared cutoff while exploring one maximal configuration and non-cutoff while exploring the next, as the corresponding event found in* `iscutoff` *might have been removed from $U \cup G$. This is in stark contrast to the classic unfolding construction, where events are declared cutoffs* once and for all.

◁

Example 34 exemplifies the scenarios where the standard proofs of using cutoffs to obtain complete prefixes fail. Thus, if we do not remove any event from $U \cup G$ or more loosely do not remove any corresponding event it follows that the updated algorithm is exploring a complete prefix.

A class of cutoffs called *causal cutoffs* satisfy the condition that the updated algorithm will not remove any corresponding event because they will be in the stack C. In particular, a *causal cutoff* is any event e for which there is some $e' \in [e]$ satisfying Equation (4.18). It is known that causal cutoffs define a finite prefix of $\mathcal{U}$ as per the classic saturation definition [BHK+14]. Also, `iscutoff(`e,U,G`)` always holds for causal cutoffs, regardless of the contents of U and G.

Nevertheless, we can show that the prefix explored by the updated algorithm is complete as per Definition 23. This means that the set G can be cleaned at discretion.

Theorem 12 (Complete Prefix)**.** *The prefix explored by Algorithm 3 updated with the cutoff mechanism described above is complete.*

Proof. We need to show that:

1. If $t \in \mathit{Trans}$ is fireable, then there exists an event $e \in \mathcal{P}_u$ such that $h(e) = \mathit{act}(t)$ and

2. For every reachable state $s \in \mathit{Reach}(M)$, there is a configuration $C \in \mathit{conf}(\mathcal{P}_u)$ such that $\mathit{st}(C) = s$.

Both items follow directly from 1) $\mathcal{P}_\prec \trianglelefteq \mathcal{P}_u$ and 2) $\mathcal{P}_\prec$ is complete. □

In [RSSK15b], we show that every maximal configuration of $\mathcal{P}_\prec$ is a subset of some configuration C_i. Although this result is stronger, it has no practical consequence.

4.4 Implementation

In this section, we present an implementation of Algorithm 3 in a prototype explicit-state model checker baptized POET (Partial Order Exploration Tool) [Sou].

Written in Haskell, a lazy functional language, POET receives a multi-threaded program in C with POSIX threads. POET accepts deterministic integer programs where all threads are created in the main function and correspond to different methods. Thread creation and join are modeled using mutexes and we only support thread synchronisation with mutexes. We assume that all method calls are inlined which restricts the support to non-recursive methods. We allow enabled statements to disable each other, namely, lock statements over the same mutex.

POET translates every statement f of the program to a lambda function λ_f: $State \to \mathsf{Maybe}\ (State, Act)$ where $\mathsf{Maybe}\ \tau$ is the standard Haskell optional type $\mathsf{Nothing} \mid \mathsf{Just}\ \tau$. We define states as pairs (k, v) where k is a map from thread identifier to control location and v is a map from variable to integer value. Given a state s and a lambda λ_f, we say that the statement is enabled at s iff $\lambda_f(s) \neq \mathsf{Nothing}$. Whenever λ_f is enabled, its execution returns a pair (s, a) where s is another state and a is an action that denotes what variables were read and written. Since the program is a deterministic integer program, the lambda associated with a statement is also deterministic. We define the standard independence relation using *read-write sets*. In particular, two statements λ_f, λ_g are dependent if one writes to a variable accessed by the other. Note that if they are from the same thread, both will write to the variable associated to the thread identifier.

POET implements Algorithm 3 with the support for cutoffs. The unfolding prefix is represented using an hash table, $\mathsf{HashTable\ EvID\ Ev}$, where an event identifier is a natural number and an event is defined as:

$$\mathsf{Ev} \triangleq TId \times L \times Act \times \mathsf{IPred} \times \mathsf{ISucc} \times \mathsf{ICfl} \times \mathsf{Disa} \times \mathsf{Alt}.$$

where:

1. TId is the thread identifier;
2. L is the thread location;
3. Act is the read-write set;
4. $\mathsf{IPred} \triangleq \mathsf{Set(EvID)}$ is the set of immediate predecessors;
5. $\mathsf{ISucc} \triangleq \mathsf{Set(EvID)}$ is the set of immediate successors;

6. $\mathsf{ICfl} \triangleq \mathsf{Set}(\mathsf{EvID})$ is the set of immediate conflicts;

7. $\mathsf{Disa} \triangleq \mathsf{Set}(\mathsf{EvID})$ is the set of disabled events and

8. $\mathsf{Alt} \triangleq \mathsf{Set}(\mathsf{List}(\mathsf{EvID}))$ is the set of alternatives.

A configuration Conf is represented as:

$$\mathsf{Conf} \triangleq \mathit{State} \times \mathsf{MaxEv} \times \mathsf{EnEv} \times \mathsf{CflEv}$$

where:

1. State is the state;

2. $\mathsf{MaxEv} \triangleq \mathsf{Set}(\mathsf{EvID})$ is the set of maximal events;

3. $\mathsf{EnEv} \triangleq \mathsf{Set}(\mathsf{EvID})$ is the set of enabled events and

4. $\mathsf{CflEv} \triangleq \mathsf{Set}(\mathsf{EvID})$ is the set of events in the configuration with immediate conflicts.

We now describe the three main components of the implementation of Algorithm 3: extensions, alternatives and cutoffs. The procedure to add events to G is exactly the one described in Section 4.2. We immediately remove the events in G.

Computing Extensions. In Line 3, the algorithm computes the extensions of a configuration C. In our implementation, we have a procedure that extends a configuration C based on a maximal event e. In particular, if $C \triangleq \varnothing$, we simply compute the set of enabled events given by the enabled statements at the initial state. Otherwise, $C \triangleq C' \cup \{e\}$ where e was the last event added. Assume that the conflicting extensions of C' have been computed in the previous step. Now, we need to compute the enabled and the conflicting extensions for C. We know that the enabled events of C' that are concurrent with e are still enabled events of C. Similarly, the conflicting extensions of C' are still conflicting extensions of C. Therefore, we only need to compute the new extensions of C where e is an immediate predecessor. These are the events labelled with statements that are enabled at $st(C)$ and are dependent with $h(e)$. Consider such statement f. Computing the enabled event at C associated with f amounts start traversing the configuration C backwards from the maximal events excluding events whose action is independent to one produced by λ_f. This procedure will terminate with a set $C'' \subseteq C$ such that all maximal events of C'' will

be dependent on λ_f and furthermore e is guaranteed to be one of them. This computes the history of the enabled event labelled with λ_f.

The final step is to compute the conflicting extensions labelled with λ_f. This process amounts to repeating the previous process except that we do not remove e as doing so would result in a configuration that would not enable λ_f. Each time we obtain a set of concurrent events that are dependent with the action of λ_f, we have found a conflicting extension to C with an event labelled with λ_f. Except for lock statements that can have two events in their minimal history, the minimal history for λ_f will be the set $\{e\}$.

Consider the problem of performing a stateful exploration with a DPOR described in Example 24. In our case, once the algorithm reaches the configuration whose state is t, we immediately compute events for both enabled transitions. This guarantees that we do not miss the deadlock state d_2.

Note that we fundamentally rely on a partial order representation as the following example shows.

Example 35. *Consider a concurrency diamond that starts at state r with transitions labelled with a and b. At the state r (assume that it is the empty configuration) we introduce two new events for a and b which are concurrent. Once we are in the configuration $\{a\}$, we immediately know that the event b is enabled at this configuration because of the independence relation.* ◁

Computing Alternatives. As specified in Algorithm 3, alternatives are pre-computed once the algorithm reaches a maximal configuration C. Instead of traversing all events in C, we restrict the search of alternatives to the pre-computed set of events in C that have immediate conflicts.

Computing Cutoffs. We have implemented the `iscutoff`$\cdot$ predicate using McMillan's size order relying on a hash table to record states. We define `iscutoff`(e, U, G) to hold iff there exists some event $e' \in U \cup G$ such that:

$$st([e]) = st([e']) \quad \text{and} \quad |[e']| < |[e]|. \tag{4.18}$$

This order is adequate [ERV02].

Table 4.1: Programs with acyclic state space. Columns are: $|P|$: nr. of threads; $|I|$: nr. of explored traces; $|B|$: nr. of sleep-set blocked executions; $t(s)$: running time; $|E|$: nr. of events in $\mathcal{U}$; $|E_{\text{cut}}|$: nr. of cutoff events; $|\Omega|$: nr. of maximal configurations; $\langle|U_\Omega|\rangle$: avg. nr. of events in U when exploring a maximal configuration. A $*$ marks programs containing bugs. `<7K` reads as "*fewer than 7000*".

Benchmark		Nidhugg			Poet(without cutoffs)				Poet(with cutoffs)				
Name	$\|P\|$	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
Stf	3	6	0	0.06	121	6	79	0.04	121	0	6	79	0.06
Stf*	3	–	–	0.05	–	–	–	0.02	–	–	–	–	0.03
Spin08	3	84	0	0.08	2974	84	1506	2.04	2974	0	84	1506	2.93
Fib	3	8953	0	3.36	<185K	8953	92878	305	<185K	0	8953	92878	704
Fib*	3	–	–	0.74	–	–	–	81.0	–	–	–	–	133
Ccnf(9)	9	16	0	0.05	49	16	46	0.07	49	0	16	46	0.06
Ccnf(17)	17	256	0	0.15	97	256	94	5.76	97	0	256	94	6.09
Ccnf(19)	19	512	0	0.28	109	512	106	22.5	109	0	512	106	22.0
Ssb	5	4	2	0.05	48	4	38	0.03	46	1	4	37	0.03
Ssb(1)	5	22	14	0.06	245	23	143	0.11	237	4	23	140	0.11
Ssb(3)	5	169	67	0.12	2798	172	1410	3.51	1179	48	90	618	0.90
Ssb(4)	5	336	103	0.15	<7K	340	3333	20.3	2179	74	142	1139	2.07
Ssb(8)	5	2014	327	0.85	<67K	2022	32782	4118	<12K	240	470	6267	32.1

4.5 Experiments

We ran Poet on a number of multi-threaded C programs. Most of them are adapted from benchmarks of the Software Verification Competition SVCOMP'15 [SV-]; others are used in related works [GFYS07, YCGK08, AAJS14]. We investigate the characteristics of average program unfoldings (depth, width, etc.) as well as the frequency and impact of cutoffs on the exploration. We also compare Poet with Nidhugg [AAA+15a], a state-of-the-art stateless model checking for multi-threaded C programs that implements Source-DPOR [AAJS14], an efficient but non-optimal DPOR. All experiments were run on an Intel Xeon CPU with 2.4 GHz and 4 GB memory. Tables 4.1 and 4.2 give our experimental data for programs with acyclic and non-acyclic state spaces, respectively.

For programs with acyclic state spaces (Table 4.1), Poet with and without cutoffs performs the same exploration when the unfolding has no cutoffs, as expected. Furthermore, the number of explored executions also coincides with Nidhugg when the latter reports zero sleep-set blocked executions. This provides experimental evidence that Poet is optimal.

The unfoldings of most programs in Table 4.1 do not contain cutoffs. All these programs are deterministic, and many of them highly sequential (such as Stf, Spin08, Fib), The cutoff criteria implemented is less effective in this kind of programs.

Table 4.2: Programs with non-terminating executions. Column b is the loop bound. The value is chosen based on experiments described in [AAA+15a].

Benchmark		Nidhugg				Poet(with cutoffs)				
Name	$\|P\|$	b	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
Szymanski	3	–	103	0	0.07	1121	313	159	591	0.36
Dekker	3	10	199	0	0.11	217	14	21	116	0.07
Lamport	3	10	32	0	0.06	375	28	30	208	0.12
Peterson	3	10	266	0	0.11	175	15	20	100	0.05
Pgsql	3	10	20	0	0.06	51	8	4	40	0.03
Rwlock	5	10	2174	14	0.83	<7317	531	770	3727	12.29
Rwlock(2)*	5	2	–	–	7.88	–	–	–	–	0.40
Prodcons	4	5	756756	0	332.62	3111	568	386	1622	5.00
Prodcons(2)	4	5	63504	0	38.49	640	25	15	374	1.61

Ccnf(n) are concurrent programs composed of $n-1$ threads where thread i and $i+1$ race on writing one variable, and are independent of all remaining threads. Their unfoldings represent $2^{(n-1)/2}$ traces with only $\mathcal{O}(n)$ events. Classical saturation-based unfolding methods are likely to out-perform both Nidhugg and Poet in these benchmarks even though adding one event is a NP-complete problem.

In the ssb benchmarks, Nidhugg encounters sleep-set blocked executions, thus performing sub-optimal exploration. We note that there is a slight difference between the number of traces in Nidhugg and the number of maximal configurations in Poet without cutoffs. We conjecture that it is because of the use of a different independence relation. By contrast, Poet finds many cutoff events and achieves a *super-optimal* exploration, exploring fewer traces than both Poet without cutoffs and Nidhugg. This data provides evidence that the use of cutoffs in a stateful exploration results in substantial savings in runtime.

For non-acyclic state spaces (Table 4.2), unfoldings are infinite. Thus, we compare Poet with cutoffs and Nidhugg with a loop bound. This means that while Nidhugg performs bounded verification, Poet does complete verification. The benchmarks include classical mutual exclusion protocols (Szymanski, Sekker, Lamport and Peterson), where Nidhugg is able to leverage an important static optimization that replaces each spin loop by a load and assume statement [AAA+15a]. Hence, the number of traces and maximal configurations is not comparable. Yet Poet, which could also profit from this static optimization to reduce the size of the unfolding, achieves a significantly better reduction because of cutoffs. Cutoffs dynamically prune redundant unfolding branches and arguably constitute a more robust approach than the load and assume syntactic substitution. The substantial reduction in number of explored traces, several orders of magnitude in some cases, translates in clear

runtime improvements. We note that cutoffs incur in some overhead because of the stateful nature of the exploration which is negligible compared to the performance gains.

Finally, in these experiments, both tools were able to successfully discover assertion violations in STF*, FIB* and RWLOCK(2)*.

In our experiments, POET's average maximal memory consumption (measured in events) is roughly half of the size of the unfolding. We also notice that most of these unfoldings are quite narrow and deep ($|E_{\text{cut}}| \div |E|$ is low) when compared with standard benchmarks for Petri nets. This suggests that they could be amenable for saturation-based unfolding verification, possibly pointing the opportunity of applying these methods in software verification.

4.6 Discussion

In this section, we discuss in more detail algorithmic aspects associated Algorithm 3, our implementation and experiments.

The Cost of Optimal Explorations. In [HRS+18], we show how to turn Algorithm 3 into a non-optimal DPOR by relaxing the constraints of the `Alt` function. We define a superset of the alternatives called *clues*.

Definition 25 (Clue)**.** *Let U be a set of events, $C \subseteq U$ a configuration, and $D \subseteq U$ a set of events. A* clue *to D after C in U is a configuration $J \subseteq U$ satisfying:*

- *$C \cup J$ is a configuration, and*
- *$D \cap J = \varnothing$.*

■

The only difference between Definition 25 and Definition 21 is that Equation (4.9) is relaxed to $D \cap J = \varnothing$. If `Alt` returns clues instead of alternatives, then Algorithm 3 can backtrack at Line 6 for a non-maximal configuration. Deciding if a set of events is an alternative in an unfolding of a multi-threaded program (such as the ones considered in this dissertation) is an NP-complete problem [HRS+18].

Example 36. *Consider the program in Figure 4.9.*

Assume that the variable k and the array V are global and the variables a, b, c are local. The constant N denotes the number of W threads and the constant M is a

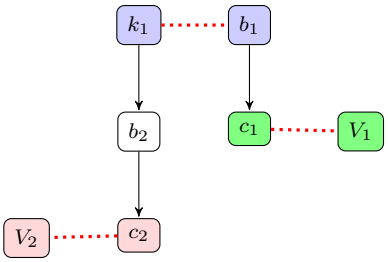
```
Thread P:                  | Thread Q:          | Thread Wi:
  for(a:=1;a<N;a++):       |   b:=k;            |   V[i]:=M;
    k++;                   |   c:=V[b];         |
```

Figure 4.9: Concurrent Program.

special value that marks which index of the array V has been written. Furthermore, consider the initial state $\tilde{s} \hat{=} k \mapsto 1, V[i] \mapsto 0$ for all $i \in \{1 \dots N\}$. For simplicity, we do not represent the program counters. In essence, `Thread Q` *picks a* non-deterministic *value for the variable b given by the position k in array V. Therefore, in a particular execution the event of* `Thread Q` *that writes to the array is only in conflict with one thread* `Thread Wi`*. All events from* `Threads Wi` *are concurrent as they write to distinct positions in the array.*

The relevant part of unfolding of this program for $N = 5$ is shown in Figure 4.10. The gray events are the minimal events of the unfolding. Note that all events V_i that write to the array are concurrent. The general structure of this unfolding is shaped by a N-sided polygon where the vertices correspond to first events of `Thread Q`*. This polygon is generated by the conflicts relative to the variable k between* `Thread Q` *and* `Thread P`*. Each V event is in conflict with one and only one extension of the polygon. Thus, it is clear that, in general, we have $2N$ M-traces.*

We now illustrate the difference between alternatives and clues, and show how relationships between races can potentially affect the performance of an implementation of source-DPOR. Consider the conflict structure of the unfolding for $N = 2$:



There are three pairs of immediate conflicting events, denoted by the three different colours. In a DPOR exploration, it is possible to reach the scenario where $D = \{k_1, V_2\}$. For example, we first explore the configuration $\{b_2, k_1, V_1, V_2\}$, discover the blue and red conflicts, and reverse them in order. Then, during the exploration, the algorithm issues a call to `Alt` $(\emptyset, \{k_1, V_2\})$*. Since V_2 is only in immediate conflict with a causal sucessor of k_1, no alternative per Definition 21 exist. However, the*

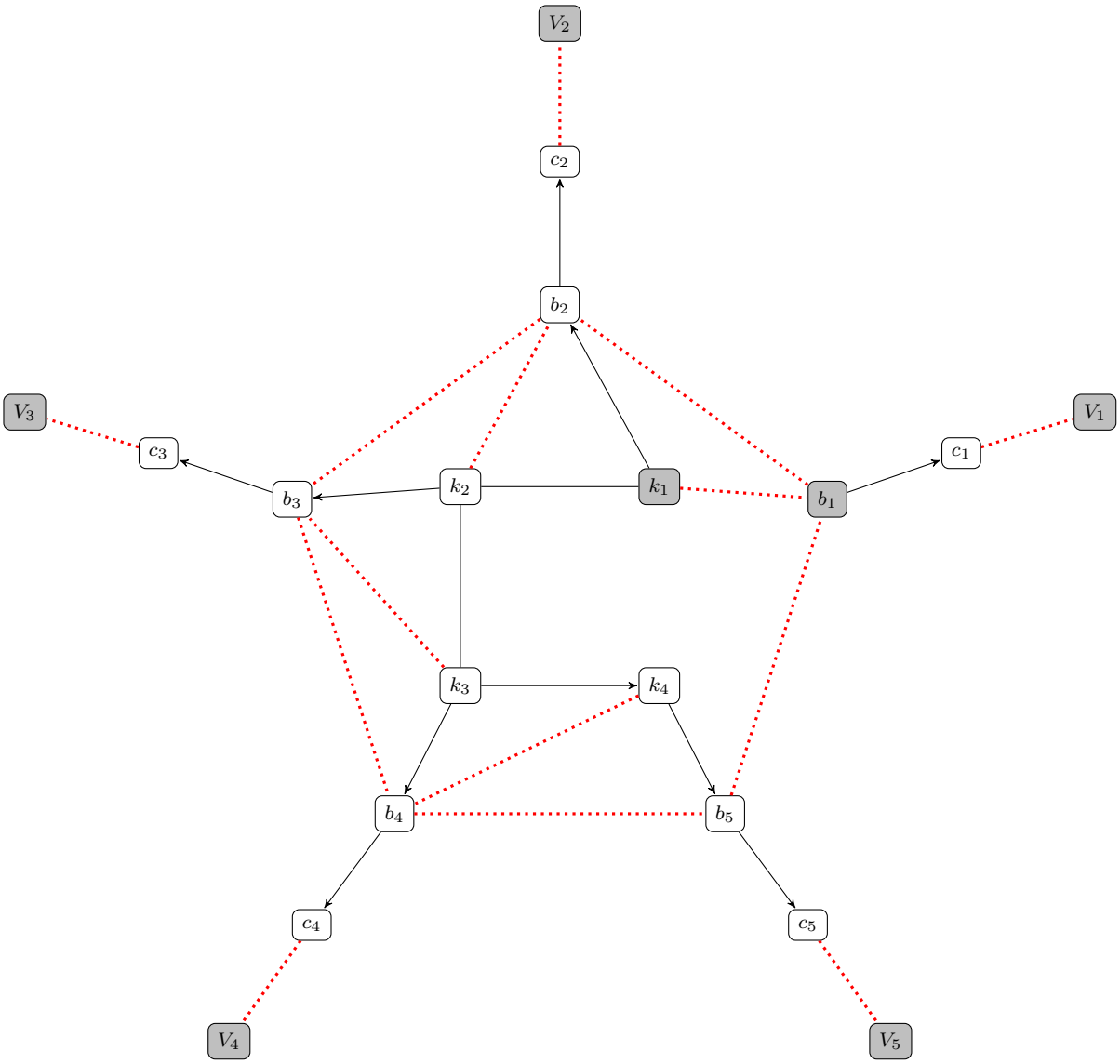


Figure 4.10: Template of the unfolding structure of program in Figure 4.9 for $N = 5$.

set $\{b_1\}$ is a clue as b_1 is an enabled extension of $\varnothing$ not in D. Continuing the exploration with this clue will necessarily lead to a SSB since the event V_2 will never be justified by another other event.

This example reveals a new relationship between conflicts which we believe has not been clarified in the literature. The red and the green conflicts are mutually exclusive owing to the order in which the blue conflict is decided. The source-DPOR algorithm [AAA+15b], does not keep track of such relationships which can lead to the exploration visiting an exponential number of SSBs (note that the number of M-traces is polynomial) we will briefly explain now. Assume that the first maximal configuration the algorithm visits is $\{V_2, V_1, k_1, b_2\}$. After it performes race detection, it discovers

the blue and red races. Since it does unaware of the entanglement between the races, it will attempt to reverse each race individually. However, once it reverses the blue race, it becomes impossible to reverse the red race. Depending on the contents of the stack, it is possible that this strategy leads to an exponential (2^N) number of SSB executions. ◁

Comparison with Algorithm in [RSSK15a]. In [RSSK15a, Algorithm 1], an optimal unfolding-based DPOR exploration is also presented. The algorithm presented in this chapter is a modification of this algorithm whose proofs are presented in the report [RSSK15b]. Our proofs are inspired by those in [RSSK15b] where the call graph explored by algorithm is also shown to be a finite binary tree.

The main difference between this algorithm and Algorithm 3 is that Algorithm 3 eagerly computes alternatives once a maximal configuration is reached. In [RSSK15a], it is unclear how the `Alt` procedure can be efficiently implemented. As a consequence, the `Remove` procedure that adds events in U to G was refined to keep fewer events in the prefix than previously. In Algorithm 3, we also explicitly formalize the backtracking condition via the `ena` function.

State Caching. Stateless explorations explore a single configuration of $\mathcal{U}$ at a time which does not leverage additional available memory. A feature of Algorithm 3 is that it exploits all available memory without imposing the liability of actually requiring it. The set G which stores events discarded from U, can be cleaned at discretion, e.g., when the memory is approaching full utilisation. Events cached in G are exploited in two different ways.

First, whenever an event in G shall be included again in U, we do not need to reconstruct it in memory (causality, conflicts, etc.). In extreme cases, this situation happen frequently. Second, cached events are necessary for cutoffs to prune the number of maximal configurations to visit. This means that this algorithm potentially visits *fewer* final states than the number of configurations of $\mathcal{U}$, thus conforming to the requirements of a *super-optimal DPOR*. The larger G is, the fewer configurations will be explored.

Limitations of the Implementation. A limitation of our implementation regarding computing extensions is that we need to execute redundant transitions. To see this, lets revisit Example 35. Here we have a concurrency diamond where immediately we explore both transitions of the beginning of the diamond. Necessarily once

we add the event a to the configuration we still need to compute the state of the configuration at the end of the diamond when we add the event b. Currently, our implementation simply re-executes the statement associated with b from the state of configuration $\{a\}$. Ideally, we would like to obtain a compositional method to compute the state of the configuration $\{a, b\}$ given the local state of the configurations $\{a\}$ and $\{b\}$ where a is independent with b.

A second limitation is related to the computation of immediate conflicts for the last event added to the prefix. Since the model of computation in our implementation is quite general, we have to resort to performing a topological sorting of the entire prefix to search which events might be in immediate conflict with this event. We have observed from preliminary experiments that this particular procedure can result in significant performance slowdown if we do not remove events of the prefix. A more recent implementation of Algorithm 3 in [HRS+18] presents efficient algorithms to compute causality and conflict by assuming a simpler model of computation which still has practical use.

Super-optimal Explorations. The notion of optimality presented in [AAJS14] is very natural considering that DPORs are backtracking explorations of representatives of equivalence classes. In this setting, optimality means that the exploration never backtracks until it finds a maximal execution and explores the same equivalence class exactly once. In the same setting, a sound super-optimal exploration would be one that is able to backtrack while exploring a representative of an equivalence class because it will observe previously seen states. In the case of finite non-acyclic state spaces, it is not surprising that our algorithm achieves super-optimal explorations as it is able to identify cycles in the state graph. However, super-optimal explorations can also be achieved in the case of acyclic state spaces. It is in this context that a comparison between our approach and optimal DPORs is more relevant. It is also important to note that when comparing different exploration strategies, super-optimal explorations can be considered more efficient with respect to an optimal exploration only if both explorations consider the same set of equivalence classes and the overhead required to achieve super-optimality is not significant. We believe this is the case in the comparison presented in Section 4.5 between Poet and Nidhugg as the underlying exploration strategy is very similar.

Related Work. This contributions of this chapter focus on explicit-state DPOR, as opposed to symbolic POR techniques exploited with SAT solvers, e.g., [KWG09,

GFYS07]. Arguably, these techniques have not been successfully been applied to programs as it is typically hard to predict the performance of the underlying solver.

Early POR statically compute the necessary transitions to fire at every state [Val91, God96]. Flanagan and Godefroid [FG05] first proposed a dynamic procedure to compute persistent set (DPOR). However, even when combined with sleep sets [God96], DPOR was still unable to explore exactly one interleaving per Mazurkiewicz trace. The major recent advance in the theory of DPOR has been the discovery of source sets in [AAJS14]. Source sets have been more recently studied in [AAJS17c, AAJS17b]. We view an optimal DPOR as a particular method to compute source sets that never lead to SSBs. In [AAJS14], this method uses wakeup trees, while in this chapter we have presented a new algorithm that uses *alternatives*. The exact relationship between alternatives and wakeup trees has not been formally studied. According to [AAJS14, AAJS17c], the construction and maintenance of wakeup trees is expensive and it is possible for the size of wakeup trees to be exponential. The only similar result is our recent investigation on the complexity of computing alternatives. In [AAJS17c], it is also presented an example alike to Example 36 where SDPOR can explore an exponential number of redundant executions for a space state with polynomial M-traces.

The use of cutoffs allows Algorithm 3 to potentially prune an exponentially number of M-traces. Not only this combination addresses non-acyclic state spaces but also turns combining a DPOR with stateful search. The latter combination has been considered to be challenging [YWY06] and not achieved for an optimal DPOR before.

Classic, saturation-based unfolding algorithms [McM93a, ERV02, BHK+14, KH14] are also related to the work in this chapter. They have been implemented for constructing unfoldings of Petri nets using a BFS-style exploration that cannot discard events from memory. However, a fundamental difference between these methods and our algorithm is that they explore events instead of configurations. Although adding one event to the prefix is a NP-complete problem their structure can be exponentially more compact than the tree explored by a DPOR. In principle, the computational cost per execution of our method is comparable to ODPOR [AAJS14] (contrary to what is stated in [AAJS17c]). In [KH14], Kähkönen and Heljanko use unfoldings for concolic testing of concurrent programs in a more restrictive setting that ours. There are recent methods [CCP+17, Hua15] that use coarser relations than the standard independence relations. They rely on SAT solvers to compute the witnesses of remaining equivalence classes which are not always representable as partial orders.

These methods still fall in the context of stateless model checking whose main application is data race detection or increasing the coverage of testing in multithreaded programs.

The theory of cutoffs has been extensively investigated for Petri net unfoldings [EH08, Chapter 5.4]. The application of these results is straightforward in our context modulo the complexity introduced by the fact that we remove events from the structure during exploration. However, a complete study of the impact of using different adequate orders in the context of stateless model checking of programs is currently missing. This would clarify the power of a stateful DPOR and provide quantitative data on the potential of super-optimal DPORs in industrial programs.

We believe that a deeper study in the structure of conflicts could improve current methods. So far, we are not aware of any such study that clarifies the possible relationships between races/conflicts how that can impact current explorations (Example 24 is a motivation example for such study).

Conclusions. In the context of independence-based DPOR, we introduced an optimal DPOR based on prime event structures. The new exploration can re-use the developed theory of Petri net unfoldings to mitigate limitations of state-of-the-art DPORs. In particular, the unfolding-based ODPOR leverages on cutoff events to prune the number of explored Mazurkiewicz traces achieving a *super-optimal* exploration. As a consequence, the technique copes with non-terminating executions, and uses state caching to speed up revisiting events. The exploration and underlying theory of prime event structures clarifies concepts from POR theory and provides a new foundation for more aggressive reduction algorithms.

Chapter 5

Causality-based Abstract Interpretation

In this chapter, we present and evaluate a technique, baptized *causality-based abstract interpretation*, for computing path-sensitive interference conditions during abstract interpretation of concurrent programs. Our main motivation is to extend the technique of Chapter 4 to non-deterministic programs by mitigating the remaining sources of explosion using abstract interpretation. Technically, we tackle the problem of extending an abstract interpreter for sequential programs to concurrent programs.

A naïve solution to this problem is a global fixpoint analysis involving all threads in the program based on the product control flow graph. A distinct solution is to analyze threads in isolation and exchange invariants on global variables between threads [Min14, Min12, CH09]. One of main goals in existing work (including ours) is to design analyses that preserve the scalability of the local fixpoints without losing the precision typically available in a global fixpoints. Our approach combines the unfolding data-structure of Chapter 3 and the exploration method of Chapter 4 with data domains to design and implement an algorithm that explores the *abstract unfolding* data-structure.

A challenge in combining unfoldings with data abstractions is that the domains typically provide approximations of states and transitions, not traces, and are not equipped with interference information. Another challenge is that unfolding algorithms are typically applied to explicit-state analysis of deterministic systems such as in Chapter 4 which restricts the applicability of the technique to general non-deterministic programs.

Our main insight is to construct an unfolding *of an analyzer*, rather than a program. In this setting, an event is the application of a transformer in an analysis context, and concurrent executions are replaced by a partial order on transformer

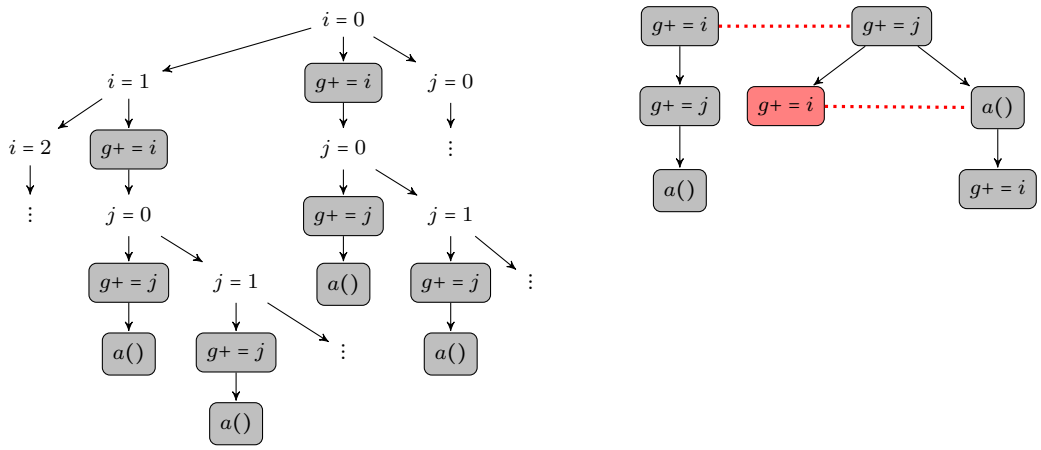


Figure 5.1: (a) DPOR exploration tree. (b) Abstract unfolding.

applications. We introduce independence for transformers and use this notion to construct the unfolding of the abstract program. Since, this unfolding is typically large, we use thread-local fixpoints to reduce its size without losing interference information. The end result is an unfolding where events represent invariants obtained with a standard abstract interpreter for sequential programs.

From a static analysis perspective, our analyzer is a path-sensitive abstract interpreter that uses the independence relation to compute a history abstraction (or trace partition) and organizes exploration information in an unfolding. Similarly to other approaches we also use a nested fixpoint. However, our key insight is that we can obtain a more precise method by using the unfolding construction as the outer fixpoint and use independence to guide the inner local fixpoints.

From a dynamic analysis perspective, our approach is an advanced DPOR (as in Chapter 4) that uses an abstract domain to collapse branches of the unfolding related to data and control non-determinism. Thus, we view our approach as a multifaceted method to overcome state explosion in analysis of concurrent programs.

Example 37. *We now illustrate the limitations of current approaches and the novelty of our method. Consider the following program which we wish to prove safe using an interval analysis.*

```
Thread 1:
  while (i++ < 99):
    if (nondet):
       break;
  g += i;
```

```
Thread 2:
  while (j++ < 149):
    if (nondet):
       break;
  g += j;
  assert (g <= 250);
```

Thread 1 (resp. 2) increments `i` *(resp.* `j`*) in a loop that can non-deterministically stop at any iteration. All variables are initialized to 0 and the program is safe as the* `assert` *in thread 2 cannot be violated.*

*When we use a DPOR approach, such as the one in Chapter 4, to prove safety of this program, the exploration algorithm exploits the fact that only the interference (*dependence*) between statements that modify the variable* `g` *can lead to distinct final states. Thus, as in Chapter 4, we use independence to define a safe fragment, shown in Figure 5.1 (a), of the computation tree of the program which can be efficiently explored. At every iteration of each loop, the conditionals open one more branch in the tree. Thus, each branch leads to a distinct write to the global variable, which is dependent with the writes of the other thread as the order of their application reaches different states. As a result, the exploration tree and the unfolding structure quickly becomes intractable. Note that the problem of exploring the DPOR tree is further exacerbated if the initial value of a variable is non-deterministic. In that case, we observe the three sources of explosion: concurrency, data and control non-determinism.*

Thread-modular static analysis implemented in ASTREEA *[Min14] or* FRAMA-C *[YB12] incorrectly triggers an alarm for this program. These tools statically analyze each thread in isolation assuming that* `g` *equals 0. Both discover that thread 1 (resp. 2) can assign the interval* $[0, 100]$ *(resp.* $[0, 150]$*) to* `g` *when it reads 0 from it. Since each thread can modify the variable read by the other, they repeat the analysis starting from the join of the new interval with the initial interval. In this iteration, they discover that thread 2 can write* $[0, 250]$ *to* `g` *when it reads* $[0, 150]$ *from it. The analysis now incorrectly determines that it needs to re-analyze thread 2, because thread 1 also wrote* $[0, 250]$ *in the previous iteration and that is a larger interval than that read by thread 2. This is the reasoning behind the false alarm. The core problem here is that these methods are path-insensitive across thread context switches and that is insufficient to prove this assertion. The analysis considers a thread context switch that can never happen (namely the one that flows* $[0, 250]$ *to thread 2 before thread 2 increments* `g`*). More recent approaches [KW16, MM17] achieve a higher degree of flow-sensitivity but they either require manual annotations to guide the trace partitioning or are restricted to program locations outside of a loop body.*

Our key contribution is an unfolding that is flow- and path-sensitive across interfering statements of the threads and path-insensitive inside the non-interfering blocks of statements. Figure 5.1 (b) shows the unfolding structure that our method would explore for this program. The events *represent firing of a transformer after a history.*

This unfolding contains three maximal configurations (executions), which correspond to the three ways in which the statements reading or writing to variable `g` *can interleave.*

Conceptually, we can construct this unfolding using the following idea: start by picking an arbitrary interleaving. Initially we pick the empty one which reaches the initial state of the program. Now we run a sequential abstract interpreter on one thread, say thread 1, from that state and stop on every location that reads or writes a global variable. In this case, the analyzer would stop at the statement `g += i` *with the invariant that* $\langle g \mapsto [0,0], i \mapsto [0,100] \rangle$. *This invariant corresponds to the first event of the unfolding (top-left corner). The unfolding contains now a new execution, so we iterate again the same procedure by picking the execution consisting of the event we just discovered. We run the analyzer on thread 2 from the invariant reached by that execution and stop on any global action. That gives rise to the event* `g+=j`*, and in the next step using the execution composed of the two events we have seen, we discover its causal successor* `a()` *(representing the* `assert`*). Note however that before visiting that event, we could have added event* `g+=j` *corresponding to the invariant of running an analyzer starting from the initial state on thread 2. Furthermore we know that because both invariants are related to the same shared variable, these two events must be ordered. We enforce that order with the conflict relation.*

Our method mitigates the aforementioned branching explosion of the DPOR tree because it never unfolds the conflicting branches of a naive exploration. In comparison to thread-modular analysis, it remains precise about the context switches because it uses an history-preserving data-structure.

Another novelty of our approach is the observation that certain events are equivalent *in the sense that the state associated with one is* subsumed *by the second. In our example, one of these events, known as a* cutoff event*, is labelled by* `g+=i` *and denoted with the light red box. Specifically, the configuration* $\{g\texttt{+=}i, g\texttt{+=}j\}$ *reaches the same state as* $\{g\texttt{+=}j, g\texttt{+=}i\}$. *Thus, no causal successor of a cutoff event needs to be explored as any action that we can discover from the cutoff event can be found somewhere else in the structure.* ◁

To summarize, the main contributions of this chapter are:

1. A new notion of transformer independence for unfolding domains (Section 5.1).

2. The unfolding of an analysis instance, a sound method to combine transformer application and independence-based partial-order reduction (Section 5.2.1).

3. A method to construct the unfolding using thread-local analysis and a new cutoff criterion (Section 5.3, Section 5.4).

4. An implementation and empirical evaluation demonstrating the trade-offs compared to an abstract interpreter and solver-based tools (Section 5.5, Section 5.6).

Analysis Instance

To simplify presentation, we equip domains with sufficient structure to lift notions from transition systems to domains.

Let $M \triangleq (\mathit{State}, \mathit{Trans} \subseteq \mathit{State} \times \mathit{Act} \times \mathit{State}, \tilde{s} \in \mathit{State})$ be a LTS. Recall from Section 2.2 that a domain $\mathcal{A} = (A, F_A)$ is a complete lattice A and a collection of transformers $f^{\mathcal{A}} : A^{ar(f)} \to A$, for each symbol in some set of symbols Sig. For the remainder of this chapter, we consider unary transformers and $\mathit{Sig} \triangleq \mathit{Act}$. We also associate a transformer to a sequence of transformers $\sigma \triangleq (f_1, \ldots, f_m)$, defined by function composition $f_\sigma \triangleq (f_m \circ \ldots \circ f_1)$.

Definition 26 (Analysis Instance)**.** *An analysis instance* $\mathcal{C} \triangleq (C, \sqsubseteq, F_C, c_0)$*, consists of a domain* $(C, \sqsubseteq, F_C)$ *and an* initial element $c_0 \in C$*.* ■

We denote by $f_a^{\mathcal{C}}$ the transformer associated to action a in domain $\mathcal{C}$ and by $\sigma^{\mathcal{C}}$ the sequence of transformers in $\mathcal{C}$. We define the map $m_0 : \mathit{Act} \to F_C$ from actions/statements to transformers: $m_0(a) \triangleq f_a^{\mathcal{C}}$. Also, we denote by $R_{\mathcal{C}}$ the *pointwise-lifting* of a binary relation R on actions or transformers of some domain to the transformers in the domain $\mathcal{C}$. When it is clear from the context, we drop superscripts and subscripts.

A transformer f is *enabled* at an element $c \in C$ when $f(c) \neq \bot$, and the result of *firing* f at c is $\mathit{st}(f, c) \triangleq f(c)$. We overload this operator when $c = c_0$ and denote $\mathit{st}(f, c_0)$ the element *generated by* or *reached by* f (which can be a sequence) by $\mathit{st}(f)$. The sequence σ is a *run* if $\mathit{st}(\sigma) \neq \bot$. As in Section 2.3.1, $\mathit{Reach}(\mathcal{C})$ denotes the set of reachable elements of $\mathcal{C}$ and $\mathit{Runs}(\mathcal{C})$ is the set of all runs of $\mathcal{C}$.

Definition 27 (Collecting Semantics)**.** *The* collecting semantics *of a transition system* $M \triangleq (\mathit{State}, \mathit{Trans}, \tilde{s})$ *is the analysis instance* $\mathcal{C}_M \triangleq (\wp(\mathit{State}), \subseteq, F, \{\tilde{s}\})$*, where* F *contains a transformer* $f_a(S) \triangleq \left\{s' \in \mathit{State} : s \in S \wedge s \xrightarrow{a} s'\right\}$ *for every action* a *of* M*.* ■

We now formalize the definition of *abstraction* used in this chapter.

Definition 28 (Abstraction)**.** *An analysis instance* $(D, \sqsubseteq, F_D, d_0)$ *is an* abstraction *of* $(C, \leqslant, F_C, c_0)$ *if there exists a* concretization function $\gamma : D \to C$*, which is monotone and satisfies:*

1. $c_0 \leqslant \gamma(d_0)$ *and*

2. $f^{\mathcal{C}} \circ \gamma \leqslant \gamma \circ f^{\mathcal{D}}$, *where the order between functions is pointwise.*

■

In the case of an abstraction, we consider the map $m{:}\, F_C \to F_D$ from concrete transformers to abstract transformers: $m(f_a^{\mathcal{C}}) \mathrel{\hat{=}} f_a^{\mathcal{D}}$.

5.1 Independence for Transformers

Similarly to independence of concurrent actions used by partial order reductions, we will use independence between transformers to compactly represent transformer applications that lead to the same result. The contribution of this section is a notion of independence for transformers and a demonstration that abstraction may both create and violate independence relationships.

Recall from Section 3.1 the definition of independence of concurrent actions. A relation $\diamond \subseteq \mathit{Act} \times \mathit{Act}$ is an *independence* for M if it is symmetric, irreflexive, and satisfies that every $(a, b) \in \diamond$ commute at every reachable state of M.

Suppose independence for transformers is defined by replacing actions and transitions with transformers and transformer application, respectively. Example 38 shows that the lifting of an independence relation on concurrent actions to a relation on transformers might not be an independence.

Example 38. *Consider the collecting semantics $\mathcal{C}_M$ of a system M with two variables,* `x` *and* `y`*, two concurrent actions $a \mathrel{\hat{=}}$* `assume(x==0)` *and $b \mathrel{\hat{=}}$* `assume(y==0)`*, and an initial element $d_0 \mathrel{\hat{=}} \{(x \mapsto 0, y \mapsto 1), (x \mapsto 1, y \mapsto 0)\}$. Since a and b read different variables, $R \mathrel{\hat{=}} \{(a, b), (b, a)\}$ is an independence relation on M. Now, observe that $\{(f_a, f_b), (f_b, f_a)\}$ is not an independence relation on $\mathcal{C}_M$, as f_a and f_b disable each other. Note, however, that the result of $(f_a \circ f_b)(d_0)$ and $(f_b \circ f_a)(d_0)$ is $\bot$ in both cases.* ◁

Weak independence, defined below, allows transformers to be considered independent even if they disable each other.

Definition 29 (Weak Independence)**.** *Let $\mathcal{D} \mathrel{\hat{=}} (D, \sqsubseteq, F_D, d_0)$ be an analysis instance. A binary relation $\diamond \subseteq F_D \times F_D$ is a weak independence on the transformers of $\mathcal{D}$ if it is irreflexive, symmetric, and satisfies that:*

$$\textit{For all } d \in \mathit{Reach}(\mathcal{D}), \textit{ if } f_1 \diamond f_2, \textit{ then } (f_1 \circ f_2)(d) = (f_2 \circ f_1)(d).$$

Moreover, $\diamond$ *is an* independence *if it is a weak independence and satisfies that if* $f_1(d) \neq \bot$ *then* $(f_1 \circ f_2)(d) \neq \bot$ *iff* $f_2(d) \neq \bot$*, for all* $d \in \mathit{Reach}(\mathcal{D})$. ■

Observe that the lifting of the relation R in Example 38 is a weak independence on $\mathcal{C}_M$. The proposition below shows the lifting of an independence relation on actions is weak independence on transformers over $\mathcal{C}_M$.

Proposition 1. *If* $\diamond$ *is an independence relation on* M *(Definition 13), then the lifted relation* $\diamond_{\mathcal{C}_M}$ *is a weak independence on the collecting semantics* $\mathcal{C}_M$.

Proof. Let f_1, f_2 be transformers of $\mathcal{C}_M$ such that $f_1 \diamond_{\mathcal{C}_M} f_2$. Let $a_1 \hat{=} m_0^{-1}(f_1)$ and $a_2 \hat{=} m_0^{-1}(f_2)$ be the corresponding program statements. We know that $a_1 \diamond a_2$. Let $d \hat{=} \{s_1, \ldots, s_n\} \in \mathit{Reach}(\mathcal{C}_M)$ be an element of $\mathcal{C}_M$. By definition of $\mathcal{C}_M$ we know that d contains only reachable states of M. Furthermore, we know that a_1 and a_2 commute on all of them. Let $d_1 \hat{=} f_1 \circ f_2(d)$ and $d_2 \hat{=} f_2 \circ f_1(d)$ be the abstract elements obtained after executing the abstract transformers in both orders. We need to show that $d_1 = d_2$. W.l.o.g., we show that $d_1 \subseteq d_2$ (the opposite direction holds by symmetry). Let $s' \in d_1$ be an state in d_1. Then there is some $s \in d$ such that $s \xrightarrow{a_2,a_1} s'$. Since a_1 and a_2 are independent under $\diamond$, then also $s \xrightarrow{a_1,a_2} s'$ (by commutativity a_1 is enabled at s because it was at the state reached after executing a_2 and both orderings reach s'). By definition of f_1, f_2 we get that $s' \in d_2$. □

We now show that independence and abstraction interact in non-trivial ways. In particular, independence of concrete transformers does not imply that their associated abstract transformers are independent. Also, those that are not independent in the concrete semantics may become independent in the abstract semantics.

Let $\mathcal{D} \hat{=} (D, \sqsubseteq, F_D, d_0)$ be an abstraction of $\mathcal{C} \hat{=} (C, \leqslant, F_C, c_0)$ and a weak independence $\diamond \subseteq F_C \times F_C$. We call the lifted relation $\diamond_{\mathcal{D}} \subseteq F_D \times F_D$ from $\mathcal{C}$ to $\mathcal{D}$, the *inherited relation* of $\mathcal{D}$ from $\mathcal{C}$.

Example 39 (Abstraction breaks independence)**.** *Consider a system* M *with the initial state* $(x \mapsto 0, y \mapsto 0)$*, and two concurrent actions* $a \hat{=}$ `x:=2`, $b \hat{=}$ `y:=7`. *Let* |v| *be the domain for interval analysis composed of pairs of intervals* (i_x, i_y) *for the values of variables* x *and* y*. The initial interval state is* $d_0 \hat{=} (x \mapsto [0,0], y \mapsto [0,0])$. *Abstract transformers for* a *and* b *are shown below.*

$$f_a^{|v|}((i_x, i_y)) := ([2,4], i_y) \qquad f_b^{|v|}(i_x, i_y) := (i_x,\ (\mathit{if}\ 3 \in i_x\ \mathit{then}\ [7,9]\ \mathit{else}\ [6,8]))$$

These transformers are deliberately imprecise to highlight that sound transformers are not necessarily the most precise. The relation $\diamond \hat{=} \{(a,b),(b,a)\}$ *is an independence*

on M, and $\diamond_{\mathcal{C}_M}$ is a weak independence on $\mathcal{C}_M$. In this example, $\diamond_{\mathcal{C}_M}$ is also an independence. However, the relation $\diamond_{\mathsf{lvl}}$ is not a weak independence because f_a^{lvl} and f_b^{lvl} do not commute at d_0, due to the imprecisions introduced by abstraction. ◁

Example 39 seems contrived as we are deliberately not using the best abstract transformers. We now present an example where the inherited independence of the best transformers is not an independence.

Example 40 (Abstraction breaks independence for best transformers)**.** *Consider the concurrent actions $a \mathrel{\hat{=}}$ `assume(x != 9)` and $b \mathrel{\hat{=}}$ `assume(x < 10)` with $c_0 \in \mathcal{C}_M \mathrel{\hat{=}} \{0, \ldots, 10\}$. The transformers associated with the concrete semantics commute at c_0, but the best transformers associated with the interval semantics do not commute at the interval $(x \mapsto [0, 10])$.* ◁

On the other hand, two concrete transformers or actions that are not independent can become independent in the abstract semantics.

Example 41 (Abstraction creates independence)**.** *Consider two actions $a \mathrel{\hat{=}}$ `x:=2` and $b \mathrel{\hat{=}}$ `x:=3`, with abstract transformers $f_a^{\mathsf{lvl}}(i_x) = [2, 3]$ and $f_b^{\mathsf{lvl}}(i_x) = [2, 3]$. Clearly the actions a and b do not commute at any state or set of states, but due to imprecision, we have that f_a^{lvl} and f_b^{lvl} commute at every interval.* ◁

5.2 Unfolding of an Abstract Domain with Independence

This section shows how unfoldings, analysis instances and abstractions can be combined. An abstract unfolding is an event structure in which an event is recursively defined as the application of a transformer after a minimal set of dependent or interfering events; and a configuration represent equivalent sequences of transformer applications (events). Analogous to an invariant map in abstract interpreters and an abstract reachability tree in software model checkers, our abstract unfolding allows for constructing an over-approximation of the set of firable transitions in a program.

5.2.1 The Unfolding of a Domain

We apply the construction in Section 3.3.2 to generate a PES $\mathcal{E} \mathrel{\hat{=}} (\mathit{Event}, <, \#, h)$ for an analysis instance $\mathcal{D} \mathrel{\hat{=}} (D, \sqsubseteq, F_D, d_0)$ with respect to a relation $\bowtie \subseteq F_D \times F_D$. We denote the unfolding by $\mathcal{U}_{\mathcal{D},\bowtie}$.

Recall that a configuration is a set of events that is causally-closed and conflict-free. Events in $\mathcal{E}$ have the form $e = (f^D, C)$, representing that the transformer f^D is applied after the transformers in configuration C are applied. The order in which transformers must be applied is represented by $<$, while $\#$ encodes transformer applications that cannot belong to the same configuration.

Recall that a configuration C generates a set of interleavings $\mathit{inter}(C)$, which define the *state* of the configuration.

$$st(C) \mathrel{\hat{=}} \bigsqcap_{\sigma \in \mathit{inter}(C)} st(\sigma)$$

A difference w.r.t. Section 3.3.2 is that we consider the state of a configuration as the meet of the states of all interleavings while in Section 3.3.2 we use the join. However, the results in Section 3.3.2 also hold with the definition of the configuration state above, since in that context, all interleavings of the configuration reach the same final state. Also, if $\bowtie$ is a weak independence relation, all interleavings lead to the same state.

The definition of the unfolding of an analysis instance is the same as Definition 17.

Definition 30 (Unfolding of an Analysis Instance)**.** *The* unfolding *of an analysis instance* $\mathcal{D} \mathrel{\hat{=}} (D, \sqsubseteq, F_D, d_0)$ *with respect to a relation* $\bowtie \subseteq F_D \times F_D$ *is the deterministic* $\bowtie$*-branching named* PES $\mathcal{U}_{\mathcal{D},\bowtie} \mathrel{\hat{=}} \mathsf{lfp}\ \mathcal{E}.\ \mathcal{E}^{\perp} \sqcup \mathit{ext}_{\rightarrow}(\mathcal{E})$ *defined as the least fixpoint of the extension transformer.* ■

Proposition 2. *The structure* $\mathcal{U}_{\mathcal{D},\bowtie}$ *generated by Definition 30 is unique.*

Proof. By Theorem 3. □

If $\bowtie$ is a weak independence, every configuration of $\mathcal{U}_{\mathcal{D},\bowtie}$ represents sequences of transformer applications *producing the same element*. If C is a local configuration, i.e. it has a unique maximal event, or if C is generated by an independence (as in Section 3.3.2), $st(C)$ will not be $\perp$. Since the set of weak independence relations is a superset of the independence relations, treating transformers as independent if they generate $\perp$ enables greater reduction in the analysis.

Theorem 13. *Let* $\diamond$ *be a* weak *independence on* $\mathcal{D} \mathrel{\hat{=}} (D, \sqsubseteq, F_D, d_0)$*,* C *be a configuration of* $\mathcal{U}_{\mathcal{D},\diamond}$ *and* σ, σ' *be interleavings of* C*. Then:*

1. $st(\sigma) = st(\sigma')$;

2. *$st(\sigma) \neq \bot$ when $\diamond$ is an independence relation;*

3. *If C is a local configuration, then $st(\sigma) \neq \bot$.*

Proof. Item (2) has already been proved in Lemma 10.

To prove item (3) we assume that (1) holds. Item (3) then holds as a consequence of the way in which the set $\mathcal{H}_{\mathcal{E},\bowtie,f}$ of histories for an event is defined. Assume that $C \hat{=} [e]$ is the local configuration of event $e \hat{=} (f, H)$. Since there is only one maximal event in C (event e), then necessarily σ has the form $\sigma \hat{=} \tilde{\sigma}.h(e)$, for $\tilde{\sigma} \in inter(H)$. From item (1) we know that all interleavings of H reach the same dataflow fact, and since $st(H)$ is defined as the meet of all of them, then necessarily $st(H) = st(\tilde{\sigma})$. We also know, from Definition 30 that transformer f is enabled at $st(H)$. This means that $st(\sigma) \neq \bot$.

Finally we prove (1). The proof is by induction on the size $|C|$ of the configuration.

Base case. $|C| = 0$ and so, $C = \varnothing$. The set of interleavings of C contains zero linearizations and the result trivially holds.

Inductive Step. Assume that the result holds for configuration of size $k-1$ and assume that $|C| = k$. Let $e \in C$ be any $<$-maximal event in C, and assume that σ and σ' have the form

$$\sigma \hat{=} \tilde{\sigma}, h(e), f_1, \ldots, f_l$$
$$\sigma' \hat{=} \tilde{\sigma}', h(e), g_1, \ldots, g_m.$$

Recall that the interleavings of a configuration are the topological orderings of events w.r.t. causality. As a result transformer $h(e)$ is independent in $\diamond$ to the transformers that label all events $f_1, \ldots, f_l$ and $g_1, \ldots, g_m$.

Now consider the dataflow fact $d \hat{=} st(\tilde{\sigma})$. If $d = \bot$, then clearly $\tilde{\sigma}, h(e), f_1$ reaches $\bot$ as well. If $d \neq \bot$, then $\tilde{\sigma}$ is a run of $\mathcal{D}$ and $d \in Reach(\mathcal{D})$. Now, by construction of $\diamond$, we have that $\tilde{\sigma}, f_1, h(e)$ is also a run of $\mathcal{D}$ and

$$st(\tilde{\sigma}, h(e), f_1) = st(\tilde{\sigma}, f_1, h(e)).$$

Applying the same argument $l-1$ times more we prove that $\tilde{\sigma}, f_1, \ldots, f_l, h(e)$ is a run of $\mathcal{D}$ and that

$$st(\sigma) = st(\tilde{\sigma}, f_1, \ldots, f_l, h(e)).$$

That is, we have "*pushed back*" the occurrence of transition $h(e)$ in the interleaving without changing the state ($\bot$ or not) reached by the interleaving. Using the same argument, this time applied to σ' instad of σ, we can also show that

$$st(\sigma') = st(\tilde{\sigma}', g_1, \ldots, g_m, h(e)).$$

Now, we remark that both $\tilde{\sigma} f_1, \ldots, f_l$ and $\tilde{\sigma}' g_1, \ldots, g_m$ are interleavings of $C \setminus \{e\}$, a configuration of size $k-1$. By induction hypothesis both interleavings thus satisfy that

$$st(\tilde{\sigma}, f_1, \ldots, f_l) = st(\tilde{\sigma}', g_1, \ldots, g_m)$$

It then follows that

$$\begin{aligned} st(\sigma) &= st(\tilde{\sigma}, f_1, \ldots, f_l, h(e)) \\ &= st(\tilde{\sigma}', g_1, \ldots, g_m, h(e)) \\ &= st(\sigma') \end{aligned}$$

□

Theorem 13 states that configurations of the unfolding represent classes of interleavings that reach the same state, when $\diamond$ is a weak independence relation. As a corollary, if one interleaving σ is an execution of $\mathcal{D}$ (i.e. $st(\sigma) \neq \bot$), then the remaining interleavings of the configuration are also executions, as all of them reach the same state. Items 2 and 3 show that if $\bowtie$ is (additionally) an independence relation or the configuration is a local configuration, then all the interleavings of C are then necessary executions of $\mathcal{D}$. Item (2) of Theorem 13 was proved in Lemma 10; items (1) and (3) are new.

Theorem 14 is the analog of Theorem 4 and shows that the unfolding is adequate for analysis in the sense that every sequence of transformer applications leading to non-$\bot$ elements that could be generated during standard analysis with a domain will be contained in the unfolding.

Theorem 14. *For every weak independence relation $\diamond$ on $\mathcal{D} \hat{=} (D, \sqsubseteq, F_D, d_0)$, and sequence of transformers $\sigma \in Runs(\mathcal{D})$, there is a unique configuration C of $\mathcal{U}_{\mathcal{D},\diamond}$ such that $\sigma \in inter(C)$.*

Proof. 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 one transformer f, then f is enabled at d_0, the initial dataflow fact of $\mathcal{D}$. Then $\varnothing$ is a history for f, as necessarily $st(\varnothing)$ enables f. This means that $e \hat{=} (f, \varnothing)$ is an event of $\mathcal{U}_{\mathcal{D},\diamond}$, and clearly $\sigma \in inter(e)$. It is easy to see that no other event e' different than e but such that $h(e) = h(e')$ can exist in $\mathcal{U}_{M,\diamond}$ and satisfy that the history $\lceil e' \rceil$ of e' equals the empty set. The representative configuration for σ therefore exists and is unique.

Inductive Step. Assume that $\sigma \mathrel{\hat{=}} \sigma' f$. By the induction hypothesis, we assume that there exist a unique configuration C' such that $\sigma' \in \mathit{inter}(C')$. By Lemma 10, all sequences in $\mathit{inter}(C')$ reach the same dataflow fact d. Furthermore, σ' is one of them, and it is also a run in $\mathit{Runs}(\mathcal{D})$. This implies that $d \neq \bot$. It also implies that f is enabled at d.

If all $<$-maximal events $e' \in C'$ satisfy that $h(e') \diamond\!\!\!\!\diamond f$, then C' is a history for transformer f, and $\mathcal{U}_{\mathcal{D},\diamond}$ contains an event $e \mathrel{\hat{=}} (f, C')$. Let $C \mathrel{\hat{=}} C' \cup \{e\} = [e]$ be the configuration that contains C' and e. Clearly $\sigma \in \mathit{inter}(C)$. Below we show that such C is unique.

Alternatively, C' could have one or more maximal events e' such that $h(e') \diamond f$. We now find a history for f inside of C', as follows. Let $C'' \mathrel{\hat{=}} C' \setminus \{e'\}$, for any such e', and let $d'' \mathrel{\hat{=}} \mathit{st}(C'')$. Since f is enabled at d and $f \diamond h(e')$, then necessary f is also enabled at d'', as otherwise $f \circ h(e)(d'') = h(e) \circ f(d'')$ would be $\bot$, and f would not be enabled at d. If all $<$-maximal events of C'' are dependent with f, then C'' is a history for f, and we set $e \mathrel{\hat{=}} (f, C'')$. If not, we can apply again the argument a finite number of times (as C' is finite) until we find a history H for f inside of C' ($\varnothing$ is always a valid history). We set $e \mathrel{\hat{=}} (f, H)$ and $C \mathrel{\hat{=}} C' \cup \{e\}$. As before, clearly $\sigma \in \mathit{inter}(C)$.

In both cases we found a configuration $C \supseteq C'$ such that $\sigma \in \mathit{inter}(C)$. We now argue that such C is unique. By induction hypothesis we know that C' is the only configuration that represents σ'. If there was another C'' that represents σ and such that $C' \nsubseteq C''$, then removing the maximal event that represents f in σ would yield a second representative for σ'. This implies that any such C'' must include C'. Showing uniqueness now reduces to showing that e is the only event in $\mathcal{U}_{\mathcal{D},\diamond}$ such that $C' \cup \{e\}$ represents σ.

By contradiction, assume that $\mathcal{U}_{\mathcal{D},\diamond}$ contains another event $e' \mathrel{\hat{=}} (f, H')$ such that $C' \cup \{e'\}$ represents σ. Assume that e has the form (f, H). Since $e \neq e'$ we know that $H \neq H'$. By construction we know that $H \subseteq C'$. If $H' \nsubseteq C'$, then $C' \cup \{e'\}$ would not be a configuration (not causally closed). So also $H' \subseteq C'$. Now, since $H \neq H'$, w.l.o.g. at least one of the maximal events in H is not in H'. Furthermore, that event is in C'. By Definition 30 this means that e' is in conflict with that event, and so $C' \cup \{e'\}$ is not a configuration. This is a contradiction. □

5.2.2 Abstract Unfoldings

The fundamental soundness theorem of abstract interpretation [CC77] state when a fixed point computed in an abstract domain soundly approximates fixed points in

a concrete domain. Our analysis constructs unfoldings instead of fixed points. The soundness of our analysis *does not* follows from fixed point soundness because the abstract unfolding we construct depends on the independence relation used. In this section, we show that even though independence may not be preserved under lifting, as shown in Example 39, inherited relations can still be used to obtain sound results.

Example 42. *In Example 39, the transformer composition* $(f_a^{|\cdot|} \circ f_a^{|\cdot|})$ *produces* $(x \mapsto [2,4], y \mapsto [6,8])$*, while* $(f_b^{|\cdot|} \circ f_a^{|\cdot|})$ *produces* $(x \mapsto [2,4], y \mapsto [7,9])$*. Consider that the events* e_a *and* e_b *are associated with the first occurrence of* $f_a^{|\cdot|}$ *and* $f_b^{|\cdot|}$*, respectively. If* $f_a^{|\cdot|}$ *and* $f_b^{|\cdot|}$ *are considered independent, the state of the configuration* $\{e_a, e_b\}$ *is* $st(f_a^{|\cdot|} \circ f_b^{|\cdot|}) \sqcap st(f_b^{|\cdot|} \circ f_a^{|\cdot|})$*, which is the abstract element* $(x \mapsto [2,4], y \mapsto [7,7])$*. Note that this abstract element is a sound abstraction of the final state* $(x \mapsto 2, y \mapsto 7)$ *reached in the concrete semantics.* ◁

We will show that sound abstractions of weakly independent transformers can be treated as independent without compromising soundness of the analysis. The soundness theorem below shows a correspondence between sequences of concrete transformer applications and the abstract unfolding. The concrete and abstract objects in Theorem 15 have different types: we are not relating a concrete unfolding with an abstract unfolding, but concrete transformer sequences with abstract configurations. Since $st(C)$ is defined as a meet of transformer sequences, the proof of Theorem 15 relies on the independence relation and has a different structure from standard proofs of fixed point soundness from transformer soundness.

Theorem 15. *Let* $\mathcal{D}$ *be a sound abstraction of an analysis instance* $\mathcal{C}$*,* $\diamond$ *be a weak independence on* $\mathcal{C}$*, and* $\diamond_{\mathcal{D}}$ *be the inherited relation on* $\mathcal{D}$*.*

For every sequence $\sigma^{\mathcal{C}} \in Runs(\mathcal{C})$ *satisfying* $st(\sigma^{\mathcal{C}}) \neq \bot$*, there is a unique configuration* C *of* $\mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$ *such that* $\sigma^{\mathcal{D}} \in inter(C)$*.*

Proof. For the same reasons as in Theorem 14, the statement of the theorem restricts σ to have at least one transformer. The proof is by induction on the length $|\sigma|$ of the run.

Base Case. Run σ fires one transformer f which is enabled at d_0. Let $\bar{f} \hat{=} m(f)$ be the associated abstract transformer. Then $\bar{f}$ is enabled at $\bar{d}_0$, and $\varnothing$ is a history for $\bar{f}$. As a result $\bar{e} \hat{=} (\bar{f}, \varnothing)$ is an event of $\mathcal{U}_{\bar{\mathcal{D}},\bar{\diamond}}$, and clearly $m(\sigma) = \bar{f} \in inter(\{\bar{e}\})$. It is immediate to show that $\{\bar{e}\}$ is the only configuration that represents $m(\sigma)$.

Inductive Step. Assume that $\sigma \hat{=} \sigma' f$. By the induction hypothesis, we assume that there exist a unique configuration $\bar{C}'$ in $\mathcal{U}_{\bar{\mathcal{D}},\bar{\diamond}}$ such that $m(\sigma') \in inter(\bar{C}')$.

We fix some notation. Let $\bar{d}' \hat{=} st(\bar{C}')$ be the abstract state reached by $\bar{C}'$. Let $d' \hat{=} st(\sigma')$ be the concrete state reached by σ'. Let $\bar{f} \hat{=} m(f)$ be the abstract counterpart of f.

Now we show that $d' \sqsubseteq \gamma(\bar{d}')$. Recall that $\bar{d}'$ is defined as the meet of the state reached by all interleavings of $\bar{C}'$. Therefore, $\bar{d}'$ does not satisfy that $st(m(\sigma')) \bar{\sqsubseteq} \bar{d}'$, which would probably be the easiest strategy to prove our goal. We follow a different reasoning. Since $m(\sigma') \in inter(\bar{C}')$, by we get that $m^{-1}(inter(\bar{C}'))$ is a set of runs of the concrete domain $\mathcal{D}$. Furthermore, all those runs reach the same concrete dataflow fact d' as σ'. Then all runs in $inter(\bar{C}')$ reach abstract dataflow facts that soundly approximate d'. What is more, in a Galois connection, the concretization map γ preserves abstract meets. Formally, for any two abstract facts $\bar{d}_1, \bar{d}_2$, we have

$$\gamma(\bar{d}_1) \sqcap \gamma(\bar{d}_2) \;\sqsubseteq\; \gamma(\bar{d}_1 \bar{\sqcap} \bar{d}_2).$$

We thus can make the following development:

$$\begin{aligned}
\gamma(\bar{d}') &= \gamma(\bar{\bigsqcap}_{\bar{\sigma} \in inter(\bar{C}')} st(\bar{\sigma})) \\
&\sqsupseteq \bigsqcap_{\bar{\sigma} \in inter(\bar{C}')} \gamma(st(\bar{\sigma})) \\
&\sqsupseteq \bigsqcap_{\bar{\sigma} \in inter(\bar{C}')} d' \\
&= d'
\end{aligned}$$

This shows that $d' \sqsubseteq \gamma(\bar{d}')$. It also shows that $\bar{f}$ is enabled at $\bar{d}' = st(\bar{C}')$, since $\bar{f}$ is a sound approximation of f.

If all maximal events e of $\bar{C}'$ are such that $h(e) \bar{\diamond} \bar{f}$, then $\bar{C}'$ is a history for $\bar{f}$, and $\bar{e} \hat{=} (\bar{f}, \bar{C}')$ is an event of $\mathcal{U}_{\bar{\mathcal{D}},\bar{\diamond}}$. Let $\bar{C} \hat{=} \bar{C}' \cup \{\bar{e}\} = [\bar{e}]$ be the configuration that contains $\bar{C}'$ and $\bar{e}$. Clearly $m(\sigma) \in inter(\bar{C})$. Below we show that such $\bar{C}$ is unique.

Alternatively, $\bar{C}'$ could have one or more maximal events independent with $\bar{f}$. Let $\bar{e}'$ be any $<$-maximal event in $\bar{C}'$ such that $h(e') \bar{\diamond} f$. In the sequel we find a history for $\bar{f}$ inside of C'. Let $\bar{C}'' \hat{=} \bar{C}' \setminus \{\bar{e}'\}$, and let $\bar{d}'' \hat{=} st(\bar{C}'')$.

We show that $\bar{f}$ is enabled at $\bar{d}''$. Since all interleavings of $\bar{C}'$ correspond to runs of $\mathcal{D}$, necessarily all interleavings of $\bar{C}''$ are also executions of $\mathcal{D}$; and all of them reach the same dataflow fact, say d''. Using the same reasoning as above, we can show that $d'' \sqsubseteq \gamma(\bar{d}'')$. Now, showing that $\bar{f}$ is enabled at $\bar{d}''$ reduces to showing that f is enabled at d''. This, in turn, is a consequence of the fact that $m^{-1}(h(\bar{e}'))(d'') = d'$ and $f(d') \neq \bot$ and the fact that $m^{-1}(h(\bar{e}'))$ and f are independent (we skip details).

This shows that $\bar{f}$ is enabled at $st(\bar{C}'')$. If all maximal events of $\bar{C}''$ are dependent with $\bar{f}$, then $\bar{C}''$ is a history for $\bar{f}$. If not, we can apply again the argument a finite

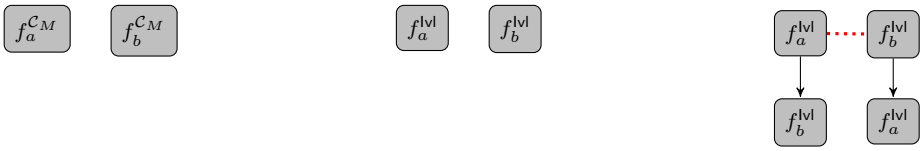


Figure 5.2: Three unfoldings: (a) $\mathcal{U}_{\mathcal{C}_M,\diamond}$, (b) $\mathcal{U}_{\mathsf{lvl},\diamond_{\mathsf{lvl}}}$ and (c) $\mathcal{U}_{\mathsf{lvl},\varnothing}$.

number of times (as $\bar{C}''$ is finite) until we find a history H for $\bar{f}$ inside of $\bar{C}''$ ($\varnothing$ is always a valid history). We set $\bar{e} \mathrel{\hat{=}} (\bar{f}, H)$ and $\bar{C} \mathrel{\hat{=}} \bar{C}' \cup \{\bar{e}\}$.

In both cases we found a configuration $\bar{C} \supseteq \bar{C}'$ such that $m(\sigma) \in \mathit{inter}(\bar{C})$. Showing that $\bar{C}$ is unique follows the same reasoning than in Theorem 14. □

Theorem 15 and Theorem 14 are fundamentally different. Theorem 14 shows that an unfolding parameterized by a weak independence relation is a data-structure for representing all sequences of transformer applications that may be generated during analysis within a domain. Theorem 15 shows that every concrete sequence of transformers has a corresponding sequence of abstract transformers. However, the abstract unfolding in Theorem 15 may not represent all transformer applications of the abstract domain in isolation. Formally, considering $\mathcal{U}^{\mathcal{D}}_{\mathcal{C},\diamond}$ the lifting of an unfolding over a concrete domain to an abstract domain, by replacing every concrete transformer with the corresponding abstract transformer, we have that $\mathcal{U}^{\mathcal{D}}_{\mathcal{C},\diamond} \trianglelefteq \mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$. In fact, every configuration of $\mathcal{U}^{\mathcal{D}}_{\mathcal{C},\diamond}$ will be isomorphic to a configuration in $\mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$.

Example 43. *Consider the concurrent actions* $a \mathrel{\hat{=}}$ `assume(x != 9)` *and* $b \mathrel{\hat{=}}$ `assume(x < 10)` *with* $c_0 \in \mathcal{C}_M \mathrel{\hat{=}} \{0, \ldots, 10\}$. *As we shown in Example 40, the relation* $\diamond \hat{=} \{(f_a, f_b), (f_b, f_a)\}$ *is a weak independence relation in* $\mathcal{C}_M$*, but inherited relation* $\diamond_{\mathsf{lvl}}$ *is not a weak independence in* lvl. *In Figure 5.2 we show three unfoldings. In (a), we show the unfolding of the concrete semantics domain using the weak independence relation. In (b), we show the unfolding of the interval domain using the inherited independence relation. Note that the* PES*s of (a) and (b) are isomorphic. Furthermore, because the transformers associated with the events should be* dependent*, the linearizations of the configuration* $\{f_a^{\mathsf{lvl}}, f_b^{\mathsf{lvl}}\}$ *reach different intervals.*

Note that we can still use a standard weak independence relation to construct the unfolding of the abstract domain $\mathcal{D}$*. In Figure 5.2 (c), we present the unfolding of the interval semantics with empty independence.* ◁

Using similar approach, we can prove a result analog to that of Theorem 14:

Theorem 16. *Let $\mathcal{D}$ be a sound abstraction of an analysis instance $\mathcal{C}$, $\diamond$ be a weak independence on $\mathcal{C}$, and $\diamond_{\mathcal{D}}$ be the inherited relation on $\mathcal{D}$.*

If C is a configuration of $\mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$ and $inter(C)^{\mathcal{C}}$ contains at least one execution in $Runs(\mathcal{C})$, then:

1. $inter(C)^{\mathcal{C}} \subseteq Runs(\mathcal{C})$ *and* (1)

2. $inter(C) \subseteq Runs(\mathcal{D})$ (2)

Proof. Clearly Equation (5.2) is a consequence of Equation (5.1) and the fact that every concrete execution $\sigma \in Runs(\mathcal{C})$ has a representative counterpart $m(\sigma) \in Runs(\bar{\mathcal{D}})$ in the abstract domain.

Assume that $m^{-1}(inter(C))$ contains one execution σ from the set $Runs(\mathcal{C})$. Recall that $\diamond_{\mathcal{D}}$ is the pointwise lifting of a weak relation $\diamond$ on $\mathcal{D}$. As a result, there exists a configuration C' in $\mathcal{U}_{\mathcal{C},\diamond}$ the unfolding of the concrete domain such that $\sigma \in inter(C')$. In fact, C and C' are isomorphic labelled partial orders, so the set of interleavings of C is in bijective correspondence with the interleavings of C'. Since σ is a run of the concrete domain $\mathcal{C}$, by Theorem 13 all interleavings of C' are also runs of $\mathcal{C}$. This proves Equation (5.1). □

Observe that the inherited relation $\diamond_{\mathcal{D}}$ might not be a weak independence. A consequence of using weak independence relations is that the state of some configurations will be $\bot$. For instance, in Example 38 the resulting unfolding would have a configuration with two concurrent events, each labelled with a different transformer, and the state of this configuration is $\bot$. Configurations reaching $\bot$ represent infeasible executions.

An implication of practical relevance is that, in general, the runs in $inter(C)$ of a configuration C yield different elements. Thus computing the state of a configuration requires enumerating its set of interleavings, which is what one ultimately expects to avoid by using partial-order methods. However, in the definition of $st(C)$, observe that the fact reached by an arbitrary interleaving overapproximates the state of the configuration. An algorithm for building $\mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$ can thus replace $st(C)$ by the overapproximated state in Definition 30 using just one of the interleavings. This results in a practical algorithm for constructing the abstract unfolding. The minor consequence of this result is that we lose the uniqueness guarantee of $\mathcal{U}_{\mathcal{D},\diamond_{\mathcal{D}}}$.

As shown in Section 3.4, larger independence relations produce smaller unfoldings. Example 41 shows that abstraction can create independence between transformers. The results of this section can be extended to exploit an arbitrary weak independence relation that might be available in $\mathcal{D}$.

5.3 Plugging Thread-Local Analysis

Unfoldings compactly represent concurrent executions using partial orders. However, they are a branching structure and one extension of the unfolding can multiply the number of branches, leading to a blow-up in the number of branches. Abstract unfoldings are able to mitigate part of the branching structure caused by data non-determinism as configurations represent sets of transformer applications. However, they are unable to address a third source of explosion caused by control non-determinism (e.g., the branches of the tree related to loop iterations in Figure 5.1 (a)).

Static analyses of sequential programs often avoid this explosion (at the expense of precision) by over-approximating (with joins or widenings) the abstract state at the control flow graph (CFG) locations where two or more program paths converge. Adequately lifting this simple idea of merging at CFG locations from sequential to concurrent programs is a highly non-trivial problem [FM07].

In this section, we present a method that addresses this challenge and can mitigate the blow-up in the size of the unfolding caused by conflicts between events of the same thread representing control non-determinism. The key idea of our method is to merge abstract states generated by statements that operate over local data of a single thread, i.e., those whose impact over the memory/environment is invisible to other threads. Intuitively, the key insight is that we can merge configurations of the unfolding and still preserve its structural properties with respect to interference. In this context, interference refers to dependence of concurrent actions, thus excluding dependence of actions in the same thread. The state of the resulting configuration will be a sound over-approximation of the states of the merged configurations at no loss of precision with respect to conflicts between events of multiple threads.

Consider an abstraction $\mathcal{D} \mathrel{\hat{=}} (D, \sqsubseteq, F_D, d_0)$ of the concrete semantics $\mathcal{C}_M \mathrel{\hat{=}} (\wp(State), \subseteq, F, \{\tilde{s}\})$ (of a program P) and a weak independence relation $\diamond \subseteq F \times F$. Our approach is to analyze P by constructing the unfolding $\mathcal{Q}_{\mathcal{D},\diamond}$ using a thread-local procedure that over-approximates the effect associated with the transformers of one thread as long as they respect the weak independence relation. Conceptually, events in this PES represent invariants obtained with the standard fixpoint of an abstract interpreter for sequential programs.

Assume that P has n threads. Let $F_1, \ldots, F_n$ be the partitioning of the set of transformers F by the thread to which they belong. Thus, we have $F = \biguplus_{1 \leq i \leq n} F_i$. For $f \in F_i$, we let $p(f) \mathrel{\hat{=}} i$ denote the thread to which f belongs. We define, per thread, the (local) transformers which can be used to run the merging analysis and

Algorithm 4: Causality-based Abstract Interpretation.

```
1  Procedure unfold(𝒟, ◇, n)
2      ℰ ≔̂ ℰ^⊥
3      forall i, C in [1...n] × conf(ℰ) do
4          for f enabled on tla(i, st(C))
5              e ≔̂ mkevent(f, C, ◇)
6              if iscutoff(e, ℰ) continue
7              ℰ ≔̂ ℰ ∪ {e}.
8  Procedure mkevent(f, C, ◇)
9      do
10         Remove from C any <-maximal event
             e such that f ◇ h(e)
11     while C changed
12     return ⟨f, C⟩
```

the (global) ones upon which we need to stop it. A transformer $f \in F_i$ is *local* when, for all other threads $j \neq i$ and all transformers $f' \in F_j$ we have $f \diamond f'$. Local transformers can only be dependent on transformers of their own thread. A transformer is *global* if it is not local. We denote by F_i^{loc} and F_i^{glo}, respectively, the set of local and global transformers in F_i. In the program of Example 37, the global transformers are the ones associated with actions to the variable `g`. The remaining statements correspond to local transformers.

We formalize the thread-local analysis using the function $\texttt{tla}\colon \mathbb{N} \times D \to D$ which plays the role of an off-the-shelf static analyzer for sequential thread code. A call to $\texttt{tla}(i, d)$ will run a static analyzer on thread i, restricted to F_i^{loc}, starting from d, and return its result which we assume is a sound fixed point. Formally, we assume that $\texttt{tla}(i, d)$ returns $d' \in D$, such that for every sequence $f_1 \cdot \ldots \cdot f_n \in (F_i^{\text{loc}})^*$ we have $(f_n \circ \ldots \circ f_1)(d) \sqsubseteq d'$. This condition requires any implementation of $\texttt{tla}(i, d)$ to return a sound approximation of the state that thread i could possibly reach after running only local transformers starting from d.

Algorithm 4 presents the pseudo-code for *causality-based abstract interpretation*. Procedure `unfold` builds an *abstract unfolding* for $\mathcal{D}$ under independence relation $\diamond$ for a program with n threads. This procedure follows exactly the same high-level strategy than the algorithms presented in Section 4.2. In particular, it extends the current configuration C with a single event from a thread i. The main difference with respect to the algorithms in Section 4.2 is that instead of executing a single transition to extend the current configuration, it executes a static analyzer on thread i starting on the state reached by C. If during this execution, a global transformer $f \in F_i^{\text{glo}}$ becomes enabled, the algorithm will try to insert a corresponding event to the unfolding. The function `mkevent` that extends the event structure with events according to Definition 17 is exactly the one described in Section 4.4. If the new event e is

a *cutoff*, i.e. an *equivalent* event is already in the unfolding prefix, then it will be ignored. Otherwise, we add it to current prefix.

We denote by $\mathcal{Q}_{\mathcal{D},\diamond}$ the PES constructed by a call to `unfold`$(\mathcal{D}, \diamond, n)$. Events of $\mathcal{Q}_{\mathcal{D},\diamond}$ are labelled by global transformers of $\mathcal{D}$. As a result, we adapt the definition of $st(C)$ to account for the effects of `tla` on a thread. We now introduce some new notions necessary to formalize the operation performed by the thread-local analysis.

Given $\mathcal{D}$ and an implementation of `tla`, we will define a new analysis instance $\mathcal{Q}$, called the *collapsing domain*. For any global transformer $f \in F_i^{\text{glo}}$, we define the *collapsing transformer* $f^{\mathcal{Q}} \colon D \to D$ of f as $f^{\mathcal{Q}} \mathrel{\hat{=}} f \circ \texttt{tla}(i)$.

The set of collapsing transformers induce a new analysis instance

$$\mathcal{Q} \mathrel{\hat{=}} (D, \sqsubseteq, F^{\mathcal{Q}}, d_0)$$

that soundly approximates $\mathcal{D}$, where the set of transformers is $F^{\mathcal{Q}} \mathrel{\hat{=}} \{f^{\mathcal{Q}} \mid f \text{ is global in } \mathcal{D}\}$.

Our notion of approximation here is different than the one given in Definition 28. There, we assume a bijection between abstract and concrete transformers (there is one transformer per action), while $\mathcal{Q}$ has only abstract transformers for those actions that can be considered global. The notion of approximation we are interested here is a suttering simulation of runs. For any run $\sigma \in \mathit{Runs}(\mathcal{D})$, recall that $\sigma^{\mathcal{Q}}$ is the subsequence of σ obtained by removing the local transformers. Clearly, $\sigma^{\mathcal{Q}} \in \mathit{Runs}(\mathcal{Q})$ when $\sigma \in \mathit{Runs}(\mathcal{D})$.

Let $\diamond$ be a weak independence on $\mathcal{D}$. In the sequel, we use a variation of the inherited relation of $\diamond$ to $\mathcal{Q}$, defined as $\diamond_{\mathcal{Q}} \mathrel{\hat{=}} \{(f^{\mathcal{Q}}, g^{\mathcal{Q}}) \mid f, g \text{ are global and } f \diamond g\}$. As in Section 5.2.2, in general, the inherited relation $\diamond_{\mathcal{Q}}$ is not a weak independence relation in $\mathcal{Q}$. For example, the function $\texttt{tla}(\cdot,\cdot)$ can apply widening on local loops or join results from different control paths. However, using a similar argument to Section 5.2.2 we can proof the following:

Lemma 23. *For any execution $\sigma \in \mathit{Runs}(\mathcal{D})$ there is a unique configuration C of $\mathcal{U}_{\mathcal{Q},\diamond_{\mathcal{Q}}}$ such that $\sigma^{\mathcal{Q}} \in \mathit{inter}(C)$.*

Proof. The proof of this result is very similar to that of Theorem 15.

Base Case. Run σ fires only local transformers and the length of $\sigma^{\mathcal{Q}}$ is zero. As in Theorem 15 there is a unique representative configuration in $\mathcal{U}_{\mathcal{Q},\diamond_{\mathcal{Q}}}$.

Inductive Step. Assume that $\sigma \mathrel{\hat{=}} \sigma' f$. By the induction hypothesis, we assume that there exist a unique configuration C' in $\mathcal{U}_{\mathcal{Q},\diamond_{\mathcal{Q}}}$ such that $\sigma' \in \mathit{inter}(C')$.

We distinguish two cases:

- Transformer f is local. Then $\hat{\sigma} = \hat{\sigma}'$ and C' is a representative configuration for σ.

- Transformer f is global. Then assume that σ is of the form $\sigma = \sigma_g \sigma_l f$, where σ_g ends in a global transformer and σ_l contains only local transformers. Observe that C' is also a representative configuration of σ_g.

  Clearly, $st(C') \sqsupseteq st(\sigma_g)$. Since $\texttt{tla}(\cdot,\cdot)$ always over-approximates the execution of any arbitrary sequence of local transformers, it must also over-approximate the execution of σ_l from $st(\sigma_g)$. This proves that f is enabled at $st(C')$.

  If all maximal events in C' are dependent with f in $\diamond_{\mathcal{Q}}$, then C' is a history for f and $e \hat{=} \langle f, C' \rangle$ an event of $\mathcal{U}_{\mathcal{Q},\diamond_{\mathcal{Q}}}$. If not, using the same reasoning as in Theorem 15 we can find a history $H \subseteq C'$ for f, and define event $e \hat{=} \langle f, H \rangle$.

  In both cases, by construction $C \hat{=} C' \cup \{e\}$ is a configuration of $\mathcal{U}_{\mathcal{Q},\diamond_{\mathcal{Q}}}$. Configuration C is a representative of $\sigma^{\mathcal{Q}}$.

□

When the `tla` performs a path-insensitive analysis, the structure $\mathcal{U}_{\mathcal{D},\diamond}$ is:

1. path-insensitive for runs that execute only local code,

2. partially path-sensitive for runs that execute one or more global transformer, and

3. flow-sensitive with respect to interference between threads.

We refer to this analysis as a *causally-sensitive* analysis as it is precise with respect to the *dynamic* interference between threads. Hence, the name *causality-based abstract interpretation*.

Algorithm 4 embeds multiple constructions explained in this chapter. For instance, when `tla` is implemented by the function $g(d,i) \hat{=} d$ and the check of cutoffs is disabled (`iscutoff` systematically returns *false*), the algorithm is equivalent to Definition 30. This is equivalent to executing algorithms of Chapter 4 without cutoffs.

We now show that $\mathcal{Q}_{\mathcal{D},\diamond}$ is a safe abstraction of $\mathcal{D}$ when `tla` applies multiple local transformers.

Theorem 17. *Let $\diamond$ be a weak independence on $\mathcal{D}$ and $\mathcal{U}_{\mathcal{D},\diamond}$ be the* PES *computed by a call to* `unfold`$(\mathcal{D},\diamond,n)$ *with* `iscutoff`$(e,\mathcal{E}) \hat{=} \mathsf{false}$. *Then, for any execution $\sigma \in Runs(\mathcal{D})$ there is a unique configuration C in $\mathcal{U}_{\mathcal{D},\diamond}$ such that $\sigma \in inter(C)$.*

Proof. A call to `unfold`$(\mathcal{D}, \diamond, n)$ with cutoff checking disabled computes the unfolding of $\mathcal{Q}$, so we have that

$$\mathcal{U}_{\mathcal{D},\diamond} = \mathcal{Q}_{\mathcal{D},\diamond}.$$

The theorem holds as a consequence of Lemma 23. □

Example 44. *When we apply Algorithm 4 to Example 37, all transformers associated to the code of the loops would be local. As a result, the control branches created by each loop iteration are folded into one single event. In Figure 5.1 (b) they correspond to events labelled with statements that write to variable* `g`. ◁

5.4 Abstract Cutoff Theory

Let $\mathcal{D}$ be an analysis instance and $\diamond$ a weak independence. If we remove the conditional statement in Line 6 of Algorithm 4, the algorithm would only terminate if every run of $\mathcal{D}$ contains finitely many global transformers sequences. This conditional check has two purposes:

1. Prevent infinite executions from inserting infinitely many events into the prefix and

2. Prune branches of the unfolding that starting with *equivalent* events.

The procedure `iscutoff` decides when an event is marked as a *cutoff*. In such cases, no causal successor of the event will be explored. Similarly to Chapter 4, the implementation of `iscutoff` cannot prune *too often*, as we want the computed PES to be a *sound* representation of the executions of $\mathcal{D}$. In particular, if a transformer is firable, then some event in the PES will be labelled by it.

In this section, we define a new cutoff criterion to be used in `iscutoff`. The main novelty of this criterion is that it exploits the lattice order $\sqsubseteq$ for more aggressive pruning than the *standard* cutoffs in the literature and in Section 4.3.

In order to prune the unfolding, we need to refer to the order in which it is constructed. A *strategy* is any strict (partial) order $\prec$ on the finite configurations of $\mathcal{U}_{\mathcal{D},\diamond}$ satisfying that when $C \subseteq C'$, then $C \prec C'$. In other words, strategies refine the natural order in which the domain is unfolded.

Each strategy identifies a set of feasible and cutoff events. Intuitively, feasible events will be those which have no cutoff among the set of causal predecessor. We now extend the cutoff definition in Section 4.3 to elements of a lattice.

Definition 31 (Cutoffs)**.** *An event e of $\mathcal{U}_{\mathcal{D},\diamond}$ is* $\prec$-feasible *if all causal predecessors $e' \in \lceil e \rceil$ are not $\prec$-cutoff. A $\prec$-feasible event is a $\prec$-cutoff if there exists some $\prec$-feasible event $e' \in \mathcal{U}_{\mathcal{D},\diamond}$, called the* corresponding event*, such that $[e'] \prec [e]$ and:*

$$st([e]) \sqsubseteq st([e']). \tag{5.3}$$

■

In other words, e will be a cutoff iff the fact reached by the branch it represents (local configuration) has already been *seen* when $\mathcal{D}$ is unfolded in the order stated by $\prec$. More formally, *seen* means that another element that is equally or less precise has been unfolded before.

While the notion of cutoffs has been around for a while in the literature of unfoldings [McM93a, ERV02, BHK+14], to the best of our knowledge, Definition 31 is the first to use a subsumption relation to match the corresponding event. The most general previous definition [BHK+14] only allowed states to be compared using equivalence relations in Equation (5.3), while we used the partial order $\sqsubseteq$.

Using the adequate orders from Definition 24, we know generalize the notion of feasible prefixes. The set of $\prec$-feasible events defines an unfolding prefix of $\mathcal{U}$:

Definition 32 (Feasible prefix)**.** *The $\prec$-prefix of $\mathcal{D}$ is the unique unfolding prefix $\mathcal{P}^{\prec}_{\mathcal{D},\diamond}$ of $\mathcal{U}_{\mathcal{D},\diamond}$ that contains exactly all $\prec$-feasible events which are not $\prec$-cutoffs.* ■

The shape and properties of the $\prec$-prefix strongly depend on the underlying strategy $\prec$. One is interested in strategies that identify complete prefixes.

A well-known unfolding strategy is the *size order* $\prec_s \subseteq E \times E$ [McM93a] as $C \prec_s C'$ iff $|C| < |C'|$. As we describe in Section 4.3, strategies using adequate orders [ERV02, BHK+14] can yield up to exponentially smaller prefixes.

We now define the notion of semantically complete for a PES and show that the size order can be used to obtain a complete prefix of the unfolding.

Theorem 18. *The unfolding prefix $\mathcal{P}^{\prec_s}_{\mathcal{D},\diamond}$ is $\mathcal{D}$-complete.*

Proof. Let $d \in Reach(\mathcal{D})$ be a reachable state. Then there is some configuration C in $\mathcal{U}_{\mathcal{D},\diamond}$ such that $d = st(C)$. If C is free of $\prec$-cutoff events, then C is in $\mathcal{P}^{\prec_s}_{\mathcal{D},\diamond}$ and we found the configuration that we searched.

If not, let $e \in C$ be a $\prec$-cutoff event and e' the corresponding event in $\mathcal{U}_{\mathcal{D},\diamond}$. Since $d \neq \bot$, any interleaving of C is a run of $\mathcal{D}$. Since $st([e']) \sqsupseteq st([e])$, any interleaving of $[e]$ can be extended with the transformers that label in any topological sorting of the events in $C \setminus [e]$, and the resulting sequence is a run $\sigma' \in Runs(\mathcal{D})$ that satisfies

$d \sqsubseteq st(\sigma')$. Furthermore, since all runs of $\mathcal{D}$ are represented as (unique) configurations of $\mathcal{U}_{\mathcal{D},\diamond}$, it is possible to extend configuration $[e]$ into a unique configuration that represents σ'. Let it be C'. We have that $d = st(C) \sqsubseteq st(C')$, and C' is at least one event smaller than C.

If C' has no cutoff, then we found the configuration that we were searching. If not, we only need to repeat this argument a finite number of times (since every time we remove at least one event from the configuration) until we find a configuration that reaches a state that covers d. □

Formally, given $\mathcal{D}$, a PES $\mathcal{E}$ is *$\mathcal{D}$-complete* iff for every reachable element $d \in Reach(\mathcal{D})$ there is a configuration C of $\mathcal{E}$ such that $st(C) \sqsupseteq d$. The key idea behind cutoff events is that, if event e is marked as a cutoff, then for any configuration C that includes e it must be possible to find a configuration C' without cutoff events such that $st(C) \sqsubseteq st(C')$. Regarding Algorithm 4, we define `iscutoff`$(e, \mathcal{E})$ to be the predicate that checks whether there is an event $e' \in \mathcal{E}$ that satisfies Definition 31. When such e' exists, including the event e in $\mathcal{E}$ is unnecessary because any configuration C such that $e \in C$ can be replayed in $\mathcal{E}$ by first executing $[e']$ and then (copies of) the events in $C \setminus [e]$.

We now prove that the PES computed with Algorithm 4 is $\mathcal{D}$-complete when using the predicate above restricted to the case where `tla` *respects independence*. More formally, we require that `tla` satisfies the following property: for any $d \in Reach(\mathcal{D})$ and any two global transformers $f \in F_i^{\text{glo}}$ and $f' \in F_j^{\text{glo}}$, if $f \diamond f'$ then

$$(f' \circ \texttt{tla}(j) \circ f \circ \texttt{tla}(i))(d) = (f \circ \texttt{tla}(i) \circ f' \circ \texttt{tla}(j))(d)$$

The reason for this restriction is because in general `tla` may over-approximate the global state (e.g. via joins and widening) in a way that it can affect the commutativity of the global transformers. If we allowed that situation to occur, cutoffs could erroneously prune the unfolding.

Theorem 19. *Let $\diamond$ be a weak independence in $\mathcal{D}$. Assume that* `tla` *respects independence and that* `iscutoff` *uses the procedure defined above. Then the* PES $\mathcal{Q}_{\mathcal{D},\diamond}$ *computed by Algorithm 4 is $\mathcal{D}$-complete.*

Proof. If $\diamond$ respects independence, then it is straightforward to show that $\diamond_{\mathcal{Q}}$ is a weak independence in $\mathcal{Q}$. That means that Theorem 18 is applicable and implies that $\mathcal{Q}_{\mathcal{D},\diamond}$ is $\mathcal{D}$-complete, as Algorithm 4 computes exactly that prefix. □

Note that Algorithm 4 terminates if the lattice order $\sqsubseteq$ is a well partial order (every infinite sequence contains an increasing pair) which includes, for instance, all finite

domains. Furthermore, it is also possible to accelerate the termination of Algorithm 4 using widenings in `tla` to *force cutoffs*.

5.5 Implementation

In this section, we present the implementation of a new program analyzer based on abstract unfoldings baptized APOET which implements an efficient variant of the exploration algorithm described in Algorithm 4. The exploration strategy is based on Algorithm 3, our stateless optimal partial order reduction method using unfoldings.

As described in Algorithm 4, APOET is an analyzer parameterized by a domain and a set of procedures: `tla`, `iscutoff` and `mkevent`. As a proof of concept, we have implemented an interval analysis and a basic parametric segmentation functor for arrays [CCL11] which we instantiate with intervals and concrete integers values (to represent offsets). In this way, we are able to precisely handle arrays of threads and mutexes. APOET supports dynamic thread creation and uses CIL to inline functions calls. The analyzer receives as input a concurrent C program that uses the POSIX thread library and parameters to control the widening level and the use of cutoffs. We implemented cutoffs according to the definition in Section 5.4 using an hash table that maps control locations to abstract values and the size of the local configuration of events. This is similar to our description of POET in Section 4.4.

APOET is parameterized by a domain functor of actions that is used to define independence and control the `tla` procedure. We have implemented an instance of the domain of actions for memory accesses and thread synchronizations. Transformers *record* the segments of the memory, intervals of addresses or sets of addresses, that have been read or written and synchronization actions related to thread creation, join and mutex lock and unlock operations. This approach is used to compute a conditional independence relation as transformers can perform different actions depending on the state. The conditional independence relation is dynamically computed and is used in the procedure `mkevent`.

Finally, the `tla` procedure was implemented with a worklist fixpoint algorithm which uses the widening level received as input. We enforce the constraints in `tla` using a predicate over the actions that denotes which transformers are global. During the standard sequential analysis based on the CFG of a thread, the fixpoint does not add transformers considered global to the worklist. This modularity allow us to define two modes of analysis for APOET: one where we assume the program to be data race free and thus only consider global transformers those that yield actions

related to thread synchronization and a more expensive mode that can detect data races by considering an action global if it accesses the heap or is related to thread synchronization.

5.6 Experiments

In this section we evaluate our approach based on abstract unfoldings. The goal of our experimental evaluation is to explore the following questions:

- Are abstract unfoldings practical? (i.e., is our approach able to yield efficient algorithms that can be used to prove properties of concurrent programs that require precise interference reasoning?)
- How does abstract unfoldings compare with competing approaches such as thread-modular analysis and symbolic partial order reduction?

Benchmark Selection. We used six benchmarks adapted from the SV-COMP'17 (corresponding to nine rows in Table 5.1) and four parametric programs (the remaining fifteen rows in Table 5.1) written by the authors: map-reduce DNA sequence analysis, producer-consumer, parallel sorting, and a thread pool. The majority of the SV-COMP benchmarks are not applicable for this evaluation since they are data deterministic (whereas our approach is primarily for data non-deterministic programs) or create unboundedly many threads, or use non-integer data types (e.g., structs, which are not currently supported by our prototype). Thus, we devised parametric benchmarks that expose data non-determinism and complex synchronization patterns, where the correctness of assertions depend on the synchronization history. We believe that all new benchmarks are as complex as the most complex ones of the SV-COMP (excluding device drivers).

Each program was annotated with assertions enforcing, among others, properties related to thread synchronization (e.g., after spawning the worker threads, the master analyses results only after all workers finished), or invariants about data (e.g., each thread accesses a non-overlapping segment of the input array).

Tool Selection. We compare our approach against the two approaches most closely related to ours: abstract interpretation based on thread-modular methods (represented by the tool Astreea) and partial-order reductions (PORs) handling data-nondeterminism (represented by two tools, Impara and CBMC v5.6).

Table 5.1: Experimental results. All experiments with APOET, IMPARA and CBMC were performed on an Intel Xeon CPU with 2.4 GHz and 4 GB memory with a timeout of 30 minutes; ASTREEA was ran on HP ZBook with 2.7 GHz i7 processor and 32 GB memory. Columns are: P: nr. of threads; A: nr. of assertions; $t(s)$: running time (MO - memory out; TO - timeout); E: nr. of events in the unfolding; E_{cut}: nr. of cutoff events; W: nr. of warnings; V: verification result (S - safe; U - unsafe); N: nr. of node states; A $*$ marks programs containing bugs. <2 reads as "*less than 2*".

Benchmark			APOET				ASTREEA		IMPARA			CBMC	
Name	P	A	$t(s)$	E	E_{cut}	W	$t(s)$	W	V	$t(s)$	N	V	$t(s)$
ATGC	3	7	0.36	47	0	1	1.07	2	-	TO	-	S	20.28
ATGC(3)	4	7	5.89	432	0	1	1.69	2	-	TO	-	S	68.84
ATGC(4)	5	7	135.61	7195	0	1	2.68	2	-	TO	-	S	287.30
FMAX	2	8	0.66	100	15	0	0.31	0	-	TO	-	-	TO
FMAX(3,3)	2	8	0.56	85	11	0	<2	2	-	TO	-	-	TO
FMAX(4,3)	2	8	0.58	85	11	0	<2	2	-	TO	-	-	TO
FMAX(5,3)	2	8	0.60	85	11	0	1.50	2	-	TO	-	-	TO
FMAX(2,4)	2	8	3.38	277	43	0	<2	2	S	TO	-	-	TO
FMAX(2,6)	2	8	47.80	1663	321	0	<2	2	S	TO	-	-	TO
FMAX(3,6)	2	8	62.53	2224	276	0	1.50	2	S	TO	-	-	TO
FMAX(4,6)	2	8	63.56	2230	207	0	<2	2	S	TO	-	-	TO
FMAX(2,7)	2	8	149.37	3709	769	0	1.87	2	S	TO	-	-	TO
FMAX(3,7)	2	8	256.85	6172	813	0	<2	2	S	TO	-	-	TO
FMAX(4,7)	2	8	295.79	6966	671	0	<2	2	S	TO	-	-	TO
LAZY	4	2	0.01	72	0	0	0.50	2	-	TO	-	S	4.43
LAZY*	4	2	0.01	72	0	1	0.49	2	U	1752.83	7957	U	1.09
SIGMA	5	5	2.57	7126	0	0	0.43	0	-	TO	-	S	136.68
SIGMA*	5	5	2.55	7126	0	1	0.43	1	-	TO	-	U	7.27
STF	3	2	0.01	69	0	0	0.66	2	S	5.81	250	S	1.96
TPOLL*	3	11	1.26	141	7	1	1.97	2	U	0.16	80	-	MO
TPOLL(3)*	4	11	108.16	1712	168	2	3.77	3	U	0.38	113	-	TO
TPOLL(4)*	5	11	1027.13	33018	1762	2	8.06	3	U	0.39	152	-	TO
THPOOL	2	24	34.71	353	103	0	1.44	5	S	TO	-	-	TO

ASTREEA implements thread-modular abstract interpretation for concurrent programs [Min14], IMPARA combines POR with interpolation-based reasoning to cope with data non-determinism [WKO13], and CBMC uses a symbolic encoding based on partial orders [AKT13]. We sought to compare against symbolic execution approaches for multithreaded programs, but we were either unable to obtain the tools from the authors or the tools were unable to parse the benchmarks.

Experimental Results. Table 5.1 presents the experimental results. When the program contained non-terminating executions (e.g., spinlocks), we used 5 loop unwindings for CBMC as well as cutoffs in APOET and a widening level of 15. For the family of FMAX benchmarks, we were not able to run ASTREEA on all instances, so we report approximated execution times and warnings based on the results provided by Antoine Miné on some of the instances. With respect to the size of the abstract unfolding, our experiments show that APOET is able to explore unfoldings up to 33K events and it was able to terminate on all benchmarks with an average execution time

of 81 seconds. In comparison with AstreeA, APoet is far more precise: we obtain only 12 warnings (of which 5 are false positives) with APoet compared to 43 (32 false positives) with AstreeA. We observe a similar trend when comparing APoet with the MThread plugin for Frama-C [YB12] and confirm that the main reason for the source of imprecision in AstreeA related to reasoning about thread interference. In the case of APoet, we obtain warnings in benchmarks that are buggy (lazy*, sigma* and tpoll* family), as expected. Furthermore, APoet reports warnings in the atgc benchmarks caused by imprecise reasoning of arrays combined with widening and also in the cond benchmark as it contains non-relational assertions.

APoet is able to outperform Impara and CBMC on all benchmarks. We believe that these experiments demonstrate that effective symbolic reasoning with partial orders is challenging as CBMC only terminates on 46% of the benchmarks and Impara only on 17%.

5.7 Discussion

Pruning without Weak Independence. In Section 5.2.2 we unfolded an abstract domain $\mathcal{D}$ into the event structure $\mathcal{U}_{\mathcal{D},\diamond_\mathcal{D}}$ using the inherited independence relation $\diamond_\mathcal{D}$ that was weak in the concrete domain $\mathcal{C}$ but not necessarily weak in the abstract one $\mathcal{D}$.

It would be natural to extend the cutoff criterion introduced above, which requires a weak independence, to employ the non-weak relation $\diamond_\mathcal{D}$. Unfortunately, in this case the feasible $\lessdot$-prefix is not necessarily complete. The proof of Theorem 18 relies on the fact that all runs of $\mathcal{D}$ will appear under the form of one configuration in $\mathcal{U}_{\mathcal{D},\diamond_\mathcal{D}}$. However, the non-weak relation $\diamond_\mathcal{D}$ fails to guarantee that.

Alternatively, one may try to change the completeness criterion, asking that all facts reachable in the concrete domain $\mathcal{C}$ are present in the unfolding of the abstract domain. Unfortunately, proving Theorem 18 with this notion of completeness fails again, for the same reason. The reasoning behind the notion of cutoff events fundamentally relies on the fact that arbitrary executions of $\mathcal{D}$ must be present in $\mathcal{U}_{\mathcal{D},\diamond_\mathcal{D}}$.

Therefore, using cutoff criteria for the abstract unfolding is possible only together with weak independence relations. Fortunately, at least for simple domains such as intervals, computing a weak independence seems to be inexpensive in practice.

Causality-based Symbolic Model Checking. In this chapter, we presented a static analysis approach that combines unfolding explorations with abstract interpretation. In principle, we could adapt our technique to the context of symbolic model checking [McM93b] where the most closely related work would be the technique in [KF13]. The major obstacle in the combination of symbolic model checking with unfoldings is that it is not clear how to compute independence relations between symbolic transformers because of the interaction between independence and abstraction. Recent work [AGIR18] in using SMT solvers to refine independence relations is a potential direction towards such combination.

Related work. The thread-modular approach in the style of rely-guarantee reasoning has been extensively studied in the past [Min14, Min12, CH09, MPR07, FQ03, KW16, MM17]. In [Min14], Miné proposes a flow-insensitive thread-modular analysis based on the interleaving semantics which forces the abstraction to cope with interleaving explosion. We address the interleaving explosion using the unfolding as a data-structure to compute a flow and path-sensitive thread interference analysis. A recent approach [MM17] uses relational domains and trace partitioning to recover precision in thread modular analysis but requires manual annotations to guide the partitioning and does not scale with the number of global variables. The analysis in [FK12] is not as precise as our approach (confirmed by experiments with Duet on a simpler version of our benchmarks) as it employs an abstraction for unbounded parallelism. The work in [KW16] presents a thread modular analysis that uses a lightweight interference analysis to achieve an higher level of flow sensitivity similar to [FK12]. The interference analysis of [KW16] uses a constraint system to discard unfeasible pairs of read-write actions which is static and less precise than our approach based on independence. The approach is also flow-insensitive in the presence of loops with global read operations.

The interprocedural analysis for recursive concurrent programs of [Jea12] does not address the interleaving explosion. A related approach that uses unfoldings is the causality-based bitvector dataflow analysis proposed in [FM07]. There, unfoldings are used as a method to obtain dataflow information while in our approach they are the fundamental data-structure to drive the analysis. Thus we can apply thread-local fixpoint analysis while their unfolding suffers from path explosion due to local branching. Furthermore, we can build unfoldings for general domains even with invalid independence relations while their approach is restricted to the independence encoded in the syntax of a Petri net and bitvector domains.

Compared to dynamic analysis of concurrent programs [AAJS14, FHRV13, KSH14, GLSW17], our approach builds directly on top of Poet (presented in Chapter 4) and is able to overcome a high degree of path explosion unrelated to thread interference.

Conclusions. We introduced a new algorithm for static analysis of concurrent programs based on the combination of abstract interpretation and unfoldings. We show in Examples 39 and 41 that there is a non-trivial interaction between independence and data abstractions which poses a fundamental problem in the direct application of independence-based explorations and data abstractions. Our algorithm explores an abstract unfolding using a new notion of independence to avoid redundant transformer application in an optimal POR strategy, thread-local fixed points to reduce the size of the unfolding, and a novel cutoff criterion based on subsumption to guarantee termination of the analysis.

Our experiments show that APoet generates about 10x fewer false positives than a mature thread modular abstract interpreter and is able to terminate on a large set of benchmarks as opposed to solver-based tools that have the same precision. We observed that the major reasons for the success of APoet are: (1) use of cutoffs to cope and prune cyclic explorations caused by spinlocks and (2) `tla` mitigates path explosion in the threads. Our analyzer is able to scale with the number of threads as long as the interference between threads does not increase.

Chapter 6

Event Abstractions and Event Structures

In the previous chapters, we tackled multiple sources of state explosion in multi-threaded programs using independence of concurrent actions and data abstractions. A key observation is that the representations of the state space rely on the fundamental notion atomic unit of behavior called *event*. However, in most algorithmic approaches this notion is usually unclear. For example, the notion of event in [AAJS14] is not consistent across executions of the same M-trace. As a consequence, it is often quite hard to design and implement algorithms that exploit more advanced reduction strategies as they are not typically connected to a mature model of concurrency such as Mazurkiewicz trace theory or event structures.

So far, we have considered events to be elements of the M-traces generated by independence of actions. In this chapter, we define a general notion of events as Boolean abstractions that satisfy a history condition. Furthermore, we introduce a domain functor for generating a domain of event sequences from a domain of events. Starting with labelled transition systems as the concrete semantics for concurrent programs, we characterize the event abstractions whose event sequences can be represented with prime event structures.

The Abstract Interpretation Perspective Abstract interpretation is a theory for approximation of semantics. A common, practical application of abstract interpretation is static analysis of programs, but the framework has been applied to compare and contrast semantic models [Cou02]. The abstract interpretation approach to comparative semantics is to start with an expressive semantics (the so called concrete semantics) that describes all the properties of interest in a system and then derive

other semantic representations as abstractions. Certain relationships between semantic models then manifest as relationships between abstractions. In particular, the notion of completeness in abstract interpretation denotes when the domains of two semantics are equivalent [GRS00].

In the rest of this chapter, we show how to derive event models from the labelled transition systems associated to concurrent programs. This is of practical relevance since we are interested in state space exploration algorithms based on event models.

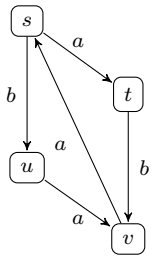
For the rest of this chapter, consider a LTS $M \mathrel{\hat{=}} (State, Trans, \tilde{s})$. Recall that $Trans \subseteq Rel \mathrel{\hat{=}} State \times Act \times State$. The most detailed behavior we consider for a system is the domain of transition sequences (Definition 2). Note that this differs from the concrete domain based on histories [Cou02].

6.1 The Event Domain

The contribution of this section is to show that various notions of an event arise as abstract interpretations of transition sequences. Intuitively, an event is an occurrence of an action [1]. This simple intuition for the notion of event allows multiple interpretations for an indivisible unit of behavior in a model of concurrency [Win88]. For example, in interleaving models, an event can be an occurrence of an action a at a particular state s. This could be any (sub)set of transitions enabled at s labelled by a, where in the extremes, each transition is an event (a suitable interpretation of events in LTS) or the maximal set (as in deterministic transition systems).

We now illustrate different notions of an event that may arise from a single system.

Example 45. *The transition system below represents a possible* [2] LTS *semantics of the process algebra [BK84] term* $((\mathrm{a} \parallel \mathrm{b}); \mathrm{a})^*$. *The initial state of this system is* s.



[1] We assume that the set of actions (or labels) is appropriated.

[2] It is not clear that the term $a \parallel b$ should have a single final state. Our use of the term is to denote that a and b are from different concurrent processes.

We consider five different notions of an event that arise in models of concurrency and say that events in behavioral models are history-dependent.

An event as a ***fireable history*** *corresponds to treating states in the synchronization tree of the system as events [SNW96]. This notion is history- and interleaving-dependent, so* $(s,a,t)(t,b,v)$ *and* (s,b,u) *represent different events. A* ***prime event****, inspired by prime event structures is history-dependent and* interleaving-independent*. A prime event respects history but disregards concurrent scheduling differences. For example, we can consider that the sequences* $(s,a,t)(t,b,v)$ *and* (s,b,u) *represent the same event* b *when the sequences* (s,a,t) *and* $(s,b,u)(u,a,v)$ *represent the same event* a *with* a *and* b *being independent. Note that a prime event is not a simple representation of a set of transition sequences* $\{\sigma_1,\sigma_2,\ldots\sigma_n\}$ *where each* σ_i *ends with a transition with the same label. The sequences* (s,b,u) *and* $(s,a,t)(t,b,v)(v,a,s)(s,b,u)$ *represent different events because the longer sequence is composed of multiple transitions of the same process.*

Viewing ***transitions*** *as events leads to a history-independent and interleaving-dependent notion. An* ***independent transition*** *is a history- and interleaving-independent notion that does not distinguish between transitions that arise due to scheduling differences. For example* (s,b,u) *and* (t,b,v) *are equivalent, and* (s,a,t) *and* (u,a,v) *are equivalent, leading to three events in this example. An extreme case of independent transitions which we call* ***action events*** *is to view events as actions. This system only has two action events.*

The dichotomy branching vs linear time also leads to different notions of event. For example, we can view an event as a subset of the set of transitions with the same source state and action. When we consider singleton sets, we obtain transitions as events (as explained above). ◁

We formalize events as Boolean abstractions of transition sequences. Let *Event* be a set of events and $ev\colon \mathit{Rel}^* \rightharpoonup \mathit{Event}$ be a partial function. We define the image and the preimage of ev as the standard functions obtained by lifting ev to the powerset domain:

$$ev_{\rightarrow}\colon \wp(\mathit{Rel}^*) \to \wp(\mathit{Event}) \qquad ev_{\leftarrow}\colon \wp(\mathit{Event}) \to \wp(\mathit{Rel}^*)$$
$$ev_{\rightarrow}(X) \mathrel{\hat{=}} \{ev(x) \mid x \in X, ev \downarrow x\} \qquad ev_{\leftarrow}(E) \mathrel{\hat{=}} \{x \mid ev \downarrow x, ev(x) \in E\}$$

Note that both $\wp(\mathit{Rel}^*)$ and $\wp(\mathit{Event})$ are Boolean algebras. If ev is a total function, then $ev_{\rightarrow}$ is a semilattice homomorphism as it does not preserves meets (e.g. $ev_{\rightarrow}(\{\sigma\} \cap \{\tau\}) = ev_{\rightarrow}(\varnothing) = \varnothing \neq ev_{\rightarrow}(\{\sigma\}) \cap ev_{\rightarrow}(\{\tau\})$) and $ev_{\leftarrow}$ is a boolean

algebra homomorphism. Since ev is a partial function, it is possible that for $X \neq \varnothing$, we have $ev_{\rightarrow}(X) = \varnothing$, i.e. the set of transition sequences X does not correspond to an event. Further, $ev_{\leftarrow}$ is no longer a Boolean algebra homomorphism as it does not preserve complements, e.g. $\neg(ev_{\leftarrow}(\varnothing)) = \neg(\varnothing) = Rel^* \neq ev_{\leftarrow}(Event) = ev_{\leftarrow}(\neg\varnothing)$.

We now prove some basic properties of $ev_{\rightarrow}$ and $ev_{\leftarrow}$.

Lemma 24. *Let S be a set and $ev: Rel^* \rightharpoonup Event$ be a partial function. $e \in ev_{\rightarrow}(S)$ iff exists a non-empty $S' \subseteq S$ such that $\{e\} = ev_{\rightarrow}(S')$.*

Proof.

($\Rightarrow$) Assume $e \in ev_{\rightarrow}(S)$. By definition of $ev_{\rightarrow}$, $e = ev(s)$ for $s \in S$. Let $S' \mathrel{\hat{=}} \{s\}$. $S' \subseteq S$ and by definition of $ev_{\rightarrow}$, $\{e\} = ev_{\rightarrow}(S')$.

($\Leftarrow$) Assume $\{e\} = ev_{\rightarrow}(S')$ for some $S' \subseteq S$. By definition of $ev_{\rightarrow}$, $e = ev(s)$ for $s \in S'$. Since $\{s\} \subseteq S' \subseteq S$, by definition of $ev_{\rightarrow}$ we have $e \in ev_{\rightarrow}(S)$. □

Lemma 25. *Let E be a set and $ev: Rel^* \rightharpoonup Event$ be a partial function. $s \in ev_{\leftarrow}(E)$ iff exists a non-empty $E' \subseteq E$ such that $\{s\} = ev_{\leftarrow}(E')$.*

Proof.

($\Rightarrow$) Assume $s \in ev_{\leftarrow}(E)$. By definition of $ev_{\leftarrow}$, $e = ev(s)$ for $e \in E$. Let $E' \mathrel{\hat{=}} \{e\}$. $E' \subseteq E$ and by definition of $ev_{\leftarrow}$, $\{s\} = ev_{\leftarrow}(E')$.

($\Leftarrow$) Assume $\{s\} = ev_{\leftarrow}(E')$ for $E' \subseteq E$. By definition of $ev_{\leftarrow}$, $e = ev(s)$ for $e \in E'$. Since $\{e\} \subseteq E' \subseteq E$, by definition of $ev_{\leftarrow}$ we have $s \in ev_{\rightarrow}(E)$. □

Lemma 26. *Let $ev: Rel^* \rightharpoonup Event$ be a partial function. The functions $ev_{\rightarrow}, ev_{\leftarrow}: \wp(Rel^*) \to \wp(Event)$ are monotone.*

Proof.

($ev_{\rightarrow}$ monotone). For all X and Y, if $X \subseteq Y$ then $ev_{\rightarrow}(X) \subseteq ev_{\rightarrow}(Y)$. Assume $e \in ev_{\rightarrow}(X)$. By Lemma 24, if $e \in ev_{\rightarrow}(X)$, then $\{e\} = ev_{\rightarrow}(X')$ for $X' \subseteq X$. Since $X \subseteq Y$, it follows that $X' \subseteq X \subseteq Y$. By Lemma 24, if $\{e\} = ev_{\rightarrow}(X')$, then $e \in ev_{\rightarrow}(Y)$.

($ev_{\leftarrow}$ monotone). For all E and F, if $E \subseteq F$ then $ev_{\leftarrow}(E) \subseteq ev_{\leftarrow}(F)$. Assume $x \in ev_{\leftarrow}(E)$. By Lemma 25, if $x \in ev_{\rightarrow}(E)$, then $\{x\} = ev_{\leftarrow}(E')$ for $E' \subseteq E$. Since $E \subseteq F$, it follows that $E' \subseteq E \subseteq F$. By Lemma 25, if $\{x\} = ev_{\rightarrow}(E')$, then $x \in ev_{\leftarrow}(F)$. □

Lemma 27. *Let $ev: Rel^* \rightharpoonup Event$ be a partial function.*

$(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\wp(Event), \subseteq)$ *where:*

$$\alpha_{\mathsf{Ev}}\colon \wp(Rel^*) \to \wp(Event) \qquad \gamma_{\mathsf{Ev}}\colon \wp(Event) \to \wp(Rel^*)$$
$$\alpha_{\mathsf{Ev}}(X) \mathrel{\hat{=}} ev_{\rightarrow}(X) \qquad \gamma_{\mathsf{Ev}}(E) \mathrel{\hat{=}} ev_{\leftarrow}(E) \cup \{x \mid ev \uparrow x\}$$

is a Galois connection.

Proof.

(α_{Ev} monotone). By Lemma 26.

(γ_{Ev} monotone). By Lemma 26 we know that $ev_{\leftarrow}$ is monotone. The function that unions a constant set to the input set is clearly monotone. The composition of two monotone functions is monotone.

($\gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}}$ extensive). We show that for all $X \in \wp(Rel^*)$, $X \subseteq (\gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}})(X)$. By definition $(\gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}})(X) = ev_{\leftarrow}(ev_{\rightarrow}(X)) \cup \{x \mid ev \uparrow x\}$. Let $x \in X$. If $ev \uparrow x$, then $x \in (\gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}})(X)$. Otherwise, $ev \uparrow x$. By definitions of $ev_{\rightarrow}$ and $ev_{\leftarrow}$ it follows that $x \in ev_{\leftarrow}(ev_{\rightarrow}(X))$ and consequently $x \in (\gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}})(X)$.

($\alpha_{\mathsf{Ev}} \circ \gamma_{\mathsf{Ev}}$ reductive). We show that for all $E \in \wp(Event)$, $(\alpha_{\mathsf{Ev}} \circ \gamma_{\mathsf{Ev}})(E) \subseteq E$. By definition, $(\alpha_{\mathsf{Ev}} \circ \gamma_{\mathsf{Ev}})(E) = ev_{\rightarrow}(\{x \in \wp(Rel^*) \mid ev \uparrow x \text{ or } (ev \downarrow x \text{ and } ev(x) \in E)\})$. By definition of $ev_{\rightarrow}$, it follows that $(\alpha_{\mathsf{Ev}} \circ \gamma_{\mathsf{Ev}})(E) = \{e \in E \mid x \in \wp(Rel^*), ev \downarrow x, e = ev(x)\}$. Thus, $(\alpha_{\mathsf{Ev}} \circ \gamma_{\mathsf{Ev}})(E) \subseteq E$.

□

The Galois connection abstracts transition sequences with events that are defined in the function ev. Because ev is a partial function, the bottom element ($\varnothing$) in $\wp(Event)$ can be obtained from any subset of the transition sequences that are undefined in ev. Since we cannot retrieve the particular set of transition sequences that was abstracted, the concretization function maps the events to the preimage of ev and the transition sequences that are undefined in ev (the maximal element). Note that it is possible for an event to correspond to a set of transition sequences that are not defined in ev, i.e. $\gamma_{\mathsf{Ev}}(\{e\}) = \gamma_{\mathsf{Ev}}(\varnothing)$ for some event e. In an event abstraction in which $\alpha_{\mathsf{Ev}}(\{\tau\}) \neq \varnothing$ exactly if τ is a fireable history, $\gamma_{\mathsf{Ev}}(\varnothing)$ will contain exactly the infeasible sequences.

Definition 33. *An event abstraction* Ev *is parameterized by* $ev\colon Rel^* \rightharpoonup Event$ *satisfying:*

1. $(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\wp(Event), \subseteq)$ *is a Galois connection.*
2. $ev \uparrow \epsilon$.
3. *If* $ev \downarrow \sigma$ *and* $\tau \preccurlyeq \sigma$, *for non-empty* τ, *then* $ev \downarrow \tau$.

4. *If* $\sigma = \tau(r, a, s)$, $\upsilon = \phi(t, b, \upsilon)$, $a \neq b$, $ev \downarrow \sigma$ *and* $ev \downarrow \upsilon$, *then* $ev(\sigma) \neq ev(\upsilon)$.

The event abstraction is total *if* $\gamma_{\mathsf{Ev}}(\{e\}) \neq \gamma_{\mathsf{Ev}}(\varnothing)$ *for every event* e. ■

The first condition above asserts that events are abstractions of transition sequences generated by the function ev (Lemma 27). The second and third conditions asserts that ev is undefined for the empty sequence and is prefix-closed. Finally, the fourth condition asserts that events will be occurrences of one action.

We now investigate some properties of the event abstraction.

Lemma 28. *For every transition sequence* σ, $\alpha_{\mathsf{Ev}}(\{\sigma\}) \subseteq \{e\}$ *for some event* e.

Proof. By definition, $\alpha_{\mathsf{Ev}}(\{\sigma\}) = ev_{\rightarrow}(\{\sigma\})$. If $ev \downarrow \sigma$, then $ev_{\rightarrow}(\{\sigma\}) = \{e\} \subseteq \{e\}$. If $ev \uparrow \sigma$, then $ev_{\rightarrow}(\{\sigma\}) = \varnothing \subseteq \{e\}$. □

As a consequence of Lemma 28, a particular transition sequence corresponds to at most one event.

Lemma 29. *An event abstraction is total iff* ev *is surjective.*

Proof. We will show that for all $e \in Event, \gamma_{\mathsf{Ev}}(\{e\}) \neq \gamma_{\mathsf{Ev}}(\varnothing)$ if and only if exists $x \in Rel^*, ev(x) = e$. Let $e \in Event$.

($\Rightarrow$) Assume $\gamma_{\mathsf{Ev}}(\{e\}) \neq \gamma_{\mathsf{Ev}}(\varnothing)$. By definition of γ_{Ev}, it follows that $ev_{\leftarrow}(\{e\}) \cup \{x \in Rel^* \mid ev \uparrow x\} \neq \{x \in Rel^* \mid ev \uparrow x\}$. It follows that $ev_{\leftarrow}(\{e\}) \neq \varnothing$. By definition of $ev_{\leftarrow}$, it follows that $\{x \in Rel^* \mid ev \downarrow x, ev(x) \in \{e\}\} \neq \varnothing$. It follows that there exists a $x \in Rel^*$ such that $ev(x) = e$. Thus, ev is surjective.

($\Leftarrow$) Assume that exists a $x \in Rel^*$ such that $ev(x) = e$. By definition of γ_{Ev}, $\gamma_{\mathsf{Ev}}(\{e\}) = ev_{\leftarrow}(\{e\}) \cup \{x \in Rel^* \mid ev \uparrow x\}$. By definition of $ev_{\leftarrow}$, we have that $ev_{\leftarrow}(\{e\}) \supseteq \{x\}$. Thus, $\gamma_{\mathsf{Ev}}(\{e\}) \supseteq \gamma_{\mathsf{Ev}}(\varnothing)$ and the event abstraction is total. □

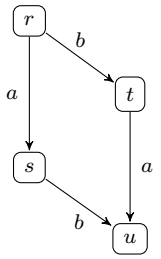
Since ev is surjective iff $ev_{\rightarrow}$ is surjective we have the following corollary.

Corollary 2. *An event abstraction is total iff* $(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\wp(Event), \subseteq)$ *is a Galois insertion.*

We give an alternative characterization of events by equivalence relations, as this formulation is sometimes convenient to use. Recall that a *partial equivalence relation* (PER) on a set S is a symmetric, transitive, binary relation on S. Unlike an equivalence relation, a PER is not reflexive and $[s \in S]_{\equiv}$ may not be defined. Given a binary relation $R \subseteq Rel^* \times Rel^*$ on transition sequences, we denote by $\equiv_R$ the PER obtained by the symmetric and transitive closure of R. Unless explicitly stated, we consider $\epsilon \equiv \epsilon$.

A PER on transition sequences is *prefix-closed* if whenever $\tau \equiv \sigma$ and there exists a non-empty τ' such that $\tau' \preccurlyeq \tau$, there exists σ' such that $\tau' \equiv \sigma'$. Note that there is no requirement that σ' be a prefix of σ. Furthermore, a PER on transition sequences is *labelled* if the last transition of an equivalence class is labelled with the same action, i.e. for all non-empty $\sigma, \tau \in Rel^*$, if $\sigma \equiv \tau$, then $\sigma = \sigma'(s, a, r)$ and $\tau = \tau'(s', a, r')$.

Example 46. *Consider the following* LTS *and four* PER*s on* Rel^**:*



$(r,a,s)(s,b,u)R_1(t,a,u)$ $\qquad$ $(r,a,s)(s,b,u)R_3(r,b,t)$

$(r,a,s)(s,b,u)R_2(r,a,s)$ $\qquad$ $(r,a,s)(s,b,u)R_4(r,b,t)$
$(r,a,s)R_4(r,b,t)(t,a,u)$

The PER $\equiv_{R_1}$ *is not* prefix-closed *(since* $[(r,a,s)]$ *is not defined) and not* labelled. *The* PER $\equiv_{R_2}$ *is* prefix-closed *and not* labelled. *The* PER $\equiv_{R_3}$ *is not* prefix-closed *(similarly to* $\equiv_{R_1}$*) and* labelled. *The* PER $\equiv_{R_4}$ *is* prefix-closed *and* labelled. *Note that* $(r,a,s)(s,b,u) \equiv_{R_4} (r,b,t)$ *and* $(r,a,s) \equiv_{R_4} (r,b,t)(t,a,u)$ *where* $(r,b,t)(t,a,u) \not\equiv (r,b,t)$. ◁

A PER $\equiv$ generates a lattice $(\wp(Rel^*/\equiv), \subseteq)$ of equivalence classes of sequences. This lattice is an abstraction of transition sequences as given by the following Galois insertion.

Lemma 30. *Let* $\equiv$ *be a* PER *on transition sequences* Rel^*.
$(\wp(Rel^*), \subseteq) \underset{\alpha_\equiv}{\overset{\gamma_\equiv}{\leftrightarrows}} (\wp(Rel^*/\equiv), \subseteq)$ *where:*

$$\alpha_\equiv : \wp(Rel^*) \to \wp(Rel^*/\equiv) \qquad \gamma_\equiv : \wp(Rel^*/\equiv) \to \wp(Rel^*)$$
$$\alpha_\equiv(X) \mathrel{\hat{=}} \{[s] \mid s \in X, s \equiv s\} \qquad \gamma_\equiv(X) \mathrel{\hat{=}} (\bigcup X) \cup \{s \in Rel^* \mid s \not\equiv s\}$$

is a Galois insertion.

Proof.

($\alpha_\equiv$ monotone). We will show that for all $X, Y \in \wp(\{Rel\}).X \subseteq Y \implies \alpha_\equiv(X) \subseteq \alpha_\equiv(Y)$. Assume that $X \subseteq Y$ and that $[s \in X] \in \alpha_\equiv(X)$. By $X \subseteq Y$, it follows that $s \in Y$. By definition of $\alpha_\equiv$, it follows that $[s \in X] \in \alpha_\equiv(Y)$.

($\gamma_\equiv$ monotone). The union of sets is monotone.

($\gamma_\equiv \circ \alpha_\equiv$ extensive). We show that for all $X \in \wp(Rel^*)$, $X \subseteq (\gamma_\equiv \circ \alpha_\equiv)(X)$. By definition $(\gamma_\equiv \circ \alpha_\equiv)(X) = (\bigcup [x \in X \mid x \equiv x]) \cup \{x \in Rel^* \mid x \not\equiv x\}$. Let $x \in X$. If $x \not\equiv x$, then $x \in (\gamma_\equiv \circ \alpha_\equiv)(X)$. Otherwise, $x \equiv x$ and $x \in (\gamma_\equiv \circ \alpha_\equiv)(X)$.

(γ_{Ev} injective). We show that for all $X \in \wp(Rel^*/\equiv)$, $(\alpha_\equiv \circ \gamma_\equiv)(X) = X$. By definition, $(\alpha_\equiv \circ \gamma_\equiv)(X) = \{[x] \mid x \in (\bigcup X), x \equiv x\}$. Thus, $(\alpha_\equiv \circ \gamma_\equiv)(X) = X$.

□

The abstraction replaces sequences by equivalence classes while the concretization is the union the contents of equivalence classes and those sequences on which equivalence is not defined.

We now relate both abstractions and show that they are equivalent.

Lemma 31. *Let $ev: Rel^* \to Event$ be a partial function satisfying the conditions in Definition 33. The kernel of ev, $\approx_{ev} \hat{=} \{(\sigma, \tau) \in Rel^* \times Rel^* \mid ev \downarrow \sigma, ev \downarrow \tau, ev(\sigma) = ev(\tau)\}$ is a prefix-closed labelled* PER *on Rel^*.*

Proof.

(Labelled). By contradiction. Let $\sigma, \tau \in Rel^*$. Assume $\sigma \approx_{ev} \tau$ with $\sigma = \sigma'(s, a, t)$ and $\tau = \tau'(u, b, v)$ such that $a \neq b$. By definition of $\approx_{ev}$ and $\sigma \approx_{ev} \tau$ it follows that $ev \downarrow \sigma, ev \downarrow \tau$ and $ev(\sigma) = ev(\tau)$. By condition 4 in Definition 33, we have that $ev(\sigma) \neq ev(\tau)$ thus reaching a contradiction.

(Prefix-closed). Let $\sigma, \tau \in Rel^*$ and assume $\sigma \approx_{ev} \tau$. If there is no non-empty $\sigma' \preccurlyeq \sigma$, then $\approx_{ev}$ is trivially prefix-closed. Otherwise, assume that $\sigma' \preccurlyeq \sigma$. By the definition of $\approx_{ev}$ it follows that $ev \downarrow \sigma$. By condition 3 in Definition 33, we have that $ev \downarrow \sigma'$. Let $\tau' \hat{=} \sigma'$. It follows from the definition of $\approx_{ev}$ that $\sigma' \approx_{ev} \tau'$. Hence, $\approx_{ev}$ is prefix-closed.

□

Lemma 32. *Let $\equiv$ be a prefix-closed labelled* PER *on transition sequences Rel^* such that $\varepsilon \not\equiv \varepsilon$. A function $ev_\equiv: Rel^* \rightharpoonup Event$ satisfying:*

1. *$ev_\equiv \uparrow \sigma$ if $\sigma \not\equiv \sigma$ and*
2. *$ev_\equiv(\sigma) = ev_\equiv(\tau)$ if $\sigma \equiv \tau$*

also satisfies the conditions in Definition 33. Furthermore, $\equiv$ is the kernel of $ev_\equiv$.

Proof. Assume that $ev_\equiv$ satisfies the conditions in the statement. We will first show that it also satisfies the conditions in Definition 33:

1. This condition is about the abstraction not the function $ev_\equiv$.
2. By $\varepsilon \not\equiv \varepsilon$ it follows that $ev_\equiv \uparrow \varepsilon$.

3. Assume $\sigma \equiv \tau$. If there is no non-empty $\sigma' \preccurlyeq \sigma$, then $ev_{\equiv}$ is prefix-closed. Otherwise, $\sigma' \preccurlyeq \sigma$ and because $\equiv$ is prefix-closed it follows that $\sigma' \preccurlyeq \tau'$. Since $\sigma' \preccurlyeq \tau'$, it follows that $ev_{\equiv} \uparrow \sigma'$.

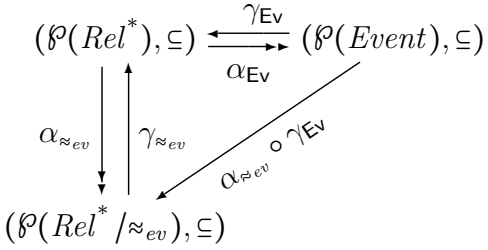
4. It follows from the definition of labelled PER that $\sigma \equiv \tau$ if their final transition is labelled with the same action. Since $ev_{\equiv}(\sigma) = ev_{\equiv}(\tau)$ if $\sigma \equiv \tau$, it follows by definition of $ev_{\equiv}$ that both transition sequences will correspond to the same event. Therefore, if two transition sequences terminate with a transition labelled with different actions they will not be equivalent and by the restriction (2) on $ev_{\equiv}$, they will correspond to different events.

It follows from the definition of $ev_{\equiv}$ and Lemma 31 that $\equiv = \approx_{ev_{\equiv}}$. □

Theorem 20 shows that prefix-closed PERs (satisfying $\varepsilon \not\equiv \varepsilon$) and total event abstractions are equivalent ways of defining the same concept.

Theorem 20. *Every total event abstraction* $(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\wp(Event), \subseteq)$ *generated by* $ev : Rel^* \rightarrowtail\!\!\!\!\circ\; Event$ *is equivalent to the abstraction* $(\wp(Rel^*), \subseteq) \underset{\alpha_{\equiv}}{\overset{\gamma_{\equiv}}{\leftrightarrows}} (\wp(Rel^*/\equiv), \subseteq)$ *generated by a prefix-closed labelled per* $\equiv$.

Proof. We assume without loss of generality from Lemmas 31 and 32 that the PER is the kernel of ev. To show that both abstractions are equivalent we will show that $\alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}}$ is a complete lattice isomorphism (Definition 1) illustrated by the following diagram.



($\alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}}$ complete lattice homomorphism). Follows directly from the definition that arbitrary unions and intersections are preserved.

($\alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}}$ bijective). By definition, the inverse of $\alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}}$ is $\alpha_{\mathsf{Ev}} \circ \gamma_{\approx_{ev}}$. It also follows directly from the definition that for all $E \in \wp(Event), E = (\alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}} \circ \alpha_{\mathsf{Ev}} \circ \gamma_{\approx_{ev}})(E)$ and that for all $X \in \wp(Rel^*/\approx_{ev}), X = (\alpha_{\mathsf{Ev}} \circ \gamma_{\approx_{ev}} \circ \alpha_{\approx_{ev}} \circ \gamma_{\mathsf{Ev}})(X)$. □

Example 47. *A* PER *for the prime event abstraction in Example 45 is the least* PER *over fireable histories satisfying these constraints:* $\tau \equiv \tau$ *for all non-empty fireable histories* τ *and for all* $\tau \equiv \sigma$, $\tau(s,a,t) \equiv \sigma(s,b,u)(u,a,v)$ *and* $\tau(s,b,u) \equiv \sigma(s,a,t)(t,b,v)$. *Note that the concatenations above must produce fireable histories.* ◁

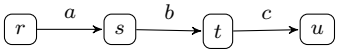
An event abstraction is *history independent* when $ev \downarrow \sigma$, $ev \downarrow \tau$ and $ev(\sigma) = ev(\tau)$ with $\sigma \preccurlyeq \tau$ for some $\sigma, \tau \in Rel^*$.

Since an event abstracts a set of transition sequences, we lift the transition sequence transformers to events, $en^{\mathsf{Ev}}_{\rightarrow}, en^{\mathsf{Ev}}_{\leftarrow}, ext^{\mathsf{Ev}}_{\rightarrow}, ext^{\mathsf{Ev}}_{\leftarrow} : \wp(Event) \rightarrow \wp(Event)$:

$$en^{\mathsf{Ev}}_{\rightarrow}(E) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ en_{\rightarrow} \circ \gamma_{\mathsf{Ev}})(E) \qquad ext^{\mathsf{Ev}}_{\rightarrow}(E) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ ext_{\rightarrow} \circ \gamma_{\mathsf{Ev}})(E)$$
$$en^{\mathsf{Ev}}_{\leftarrow}(E) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ en_{\leftarrow} \circ \gamma_{\mathsf{Ev}})(E) \qquad ext^{\mathsf{Ev}}_{\leftarrow}(E) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ ext_{\leftarrow} \circ \gamma_{\mathsf{Ev}})(E)$$

The enabledness transformers construct the enabled events by abstracting the transitions while the extension transformers construct the extension events by concatenating it with its history. Note that the extension transformers does not use the enabledness transformers; in particular since an event is a set of transition sequences, the extension transformer constructs in a *more natural sense* the set of enabled events.

Example 48. *Consider the following* LTS *with initial state r and the event abstraction generated by ev:*



$$ev((r,a,s)) \mapsto e_a \quad ev((s,b,t)) \mapsto e_b \quad ev((t,c,u)) \mapsto e_c$$
$$ev((r,a,s)(s,b,t)) \mapsto e_{ab} \quad ev((s,b,t)(t,c,u)) \mapsto e_{bc}$$
$$ev((r,a,s)(s,b,t)(t,c,u)) \mapsto e_{abc}$$

The event e_b is enabled enabled after e_a and e_{ab} is a forward extension of e_a. The event e_a is enabled before e_{bc} and the event e_{abc} is a backwards extension of e_{bc}. Clearly, the enabledness and the extension transformers are incomparable. ◁

An event abstraction is an upper closure ρ on the domain of transition sequences where the set of fixpoints of ρ (which corresponds to the image of ρ) is the set of events. Recall that an abstract interpretation ρ is *forward-complete* for a transformer f when $\rho \circ f = \rho \circ f \circ \rho$. Since the kernel of an injective function is the identity relation, an event abstraction parameterized by an injective function $ev\colon Rel^* \rightarrow Event$ is forward complete.

Event abstractions can be ordered using standard subset ordering over the set of equivalence classes induced by the PER. In Figure 6.1, we present the relationship between the several event abstractions described in Example 45. The *Empty* abstraction corresponds to the event abstraction where ev is always undefined and *Transition Sequences* corresponds to the event abstraction where ev is a bijection.

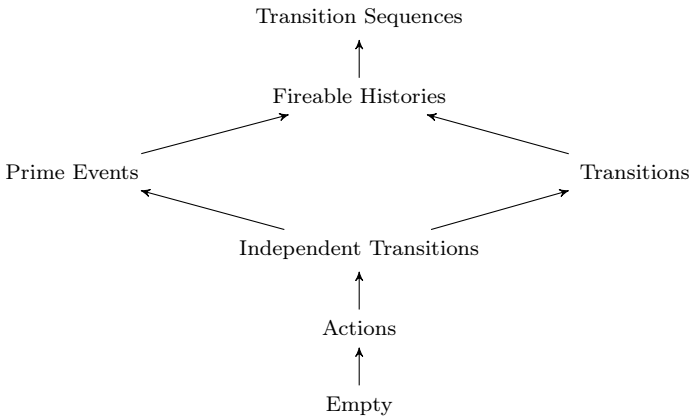


Figure 6.1: Hierarchy of event abstractions.

6.2 The Event Sequence Domain

In this section we define a domain functor for constructing event sequences from an event abstraction. The main contribution is a method that checks whether the event sequences generated by an event abstraction can be represented with prime event structures.

Events represent an abstract unit of behavior. The evolution of a system over time has different representations, including as partial orders on events [Win88] or relations between sets of events [vGP09]. We model behavior as sequences of events. We introduce a domain functor for generating a domain of event sequences from a domain of events. This domain exploits the prefix-closure condition of Definition 33 to derive event sequences from transition sequences.

Example 49. *Consider the sequence* $\sigma \mathrel{\hat{=}} (s,a,t)(t,b,v)(v,a,s)(s,b,u)$ *from the system in Example 45. When fireable histories define events, each prefix of* σ *of length* n *represents a unique event* e_n*. In particular* $\alpha_{\mathsf{Ev}}(\{\sigma\}) = \{e_4\}$*. The sequence* σ *can be viewed in terms of the events fired along it:* $e_1e_2e_3e_4$*.* ◁

Consider a total event abstraction $\wp(Rel^*) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} \wp(Event)$ generated by $ev\colon Rel^* \rightharpoonup Event$. We define the lifting of ev to $Event^*$ as:

$$ev^*\colon Rel^* \rightharpoonup Event^*$$
$$ev^*(\epsilon) \mathrel{\hat{=}} \epsilon$$
$$ev^*(\sigma = \sigma'(r,a,s)) \mathrel{\hat{=}} ev^*(\sigma') \cdot ev(\sigma) \text{ if } ev \downarrow \sigma$$

Lemma 33. *For all non-empty $\sigma \in Rel^*$, $ev \downarrow \sigma$ iff $ev^* \downarrow \sigma$.*

Proof. Both directions follow immediately from the definition of ev^* and the fact ev is prefix closed. □

Note that ev^* is not a monoid homomorphism since it does not respect concatenation, i.e. for all $\sigma, \tau \in Rel^*$, $ev^*(\sigma \cdot \tau) \neq ev^*(\sigma) \cdot ev^*(\tau)$. However, because $ev^*(\epsilon) = \epsilon$, it follows that $ev^*(\epsilon \cdot \sigma) = ev^*(\epsilon) \cdot ev^*(\sigma) = ev^*(\sigma)$.

Lemma 34 shows that ev^* is a monotone function between the subset of $(Rel^*, \preccurlyeq)$ where ev^* is defined and the free monoid $(Event^*, \preccurlyeq)$.

Lemma 34. *For all $\sigma, \tau \in Rel^*$, if $ev^* \downarrow \sigma$, $ev^* \downarrow \tau$ and $\sigma \preccurlyeq \tau$, then $ev^*(\sigma) \preccurlyeq ev^*(\tau)$.*

Proof. Assume $ev^* \downarrow \sigma$, $ev^* \downarrow \tau$ and $\sigma \preccurlyeq \tau$. By $\sigma \preccurlyeq \tau$, let $\tau \mathrel{\hat{=}} \sigma \cdot \tau'$. If $\tau' = \varepsilon$, then $\tau = \sigma$ and $ev^*(\sigma) \preccurlyeq ev^*(\tau)$. Otherwise, by definition of ev^* and since $ev^* \downarrow \tau$, $ev^*(\sigma \cdot \tau') = ev^*(\sigma) \cdot ev^*(\sigma \cdot \tau'[0]) \cdot \ldots \cdot ev^*(\tau)$. Hence, $ev^*(\sigma) \preccurlyeq ev^*(\tau)$. □

We denote by $ev^*_{\rightarrow}, ev^*_{\leftarrow} : \wp(Rel^*) \to \wp(Event^*)$ the image and preimage of ev^*. It is clear by its definition that $ev^*_{\rightarrow}$ maps a transition sequence to at most one event sequence with the same length.

The functor $\mathsf{ESeq}(\cdot)$ maps an event abstraction $\wp(Rel^*) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} \wp(Event)$ to the event sequence abstraction $\mathsf{ESeq}(\mathsf{Ev})$ defined in Lemma 35 [3].

Lemma 35. *Let $(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{Ev}}}{\overset{\gamma_{\mathsf{Ev}}}{\leftrightarrows}} (\wp(Event), \subseteq)$ be a total event abstraction generated by $ev : Rel^* \multimap Event$.*

$(\wp(Rel^*), \subseteq) \underset{\alpha_{\mathsf{ESeq}}}{\overset{\gamma_{\mathsf{ESeq}}}{\leftrightarrows}} (\wp(Event^*), \subseteq)$ where:

$$\alpha_{\mathsf{ESeq}} : \wp(Rel^*) \to \wp(Event^*) \qquad \gamma_{\mathsf{ESeq}} : \wp(Event^*) \to \wp(Rel^*)$$
$$\alpha_{\mathsf{ESeq}}(X) \mathrel{\hat{=}} ev^*_{\rightarrow}(X) \qquad \gamma_{\mathsf{ESeq}}(E) \mathrel{\hat{=}} ev^*_{\leftarrow}(E) \cup \{x \mid ev^* \uparrow x\}$$

is a Galois connection.

Proof. By Lemma 27. □

When clear from context, we write ESeq for $\mathsf{ESeq}(\mathsf{Ev})$. The concretization map goes from an event sequence to the transition sequences undefined in ev^* together with the transition sequence (if there exists one) whose prefixes generate exactly the same event sequence.

[3] In [DKS17], we provide a more cumbersome Galois connection that uses the adjoints in the event abstraction.

If we remove the second condition from Definition 33 it would have been possible to consider an event abstraction where ϵ and a transition sequence (r, a, s) is abstracted to some event e. In that case, if $ev^*(\epsilon) = \epsilon$, then the event abstraction for ϵ would not be lifted to event sequences, i.e. $\alpha_{\mathsf{Ev}}(\{\epsilon\}) \neq \alpha_{\mathsf{ESeq}}(\{\epsilon\})$. Alternatively, if $ev^*(\epsilon) = ev(\epsilon)$ if $ev \downarrow \epsilon$,it follows that $ev^*(\epsilon{\cdot}\sigma) \neq ev^*(\sigma)$ for some σ. Furthermore, the condition $ev \uparrow \epsilon$ follows our intuition that an event is an occurrence of action.

An event sequence σ is *feasible* if $\gamma_{\mathsf{ESeq}}(\{\sigma\}) \neq \gamma_{\mathsf{ESeq}}(\varnothing)$.

Example 50. *Consider the function ev where:*

$$ev(t_1) \mapsto e_1 \qquad ev(t_2) \mapsto e_2 \qquad ev(t_1t_2) \mapsto e_3$$

The event sequence e_3 is not feasible *because it is missing the prefix for the transition sequence t_1. The sequence e_1e_2 is* not feasible *because none of transition sequences corresponding to e_1 is a prefix of the transition sequences corresponding to e_2.* ◁

The enabled event transformers $en_{\rightarrow}^{\mathsf{ESeq}}, en_{\leftarrow}^{\mathsf{ESeq}} : \wp(Event^*) \to \wp(Event)$ map from a transition sequence to events enabled at the beginning or end.

$$en_{\rightarrow}^{\mathsf{ESeq}}(X) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ ext_{\rightarrow} \circ \gamma_{\mathsf{ESeq}})(X)$$
$$en_{\leftarrow}^{\mathsf{ESeq}}(X) \mathrel{\hat{=}} (\alpha_{\mathsf{Ev}} \circ en_{\leftarrow} \circ \gamma_{\mathsf{ESeq}})(X)$$

The forward enabledness constructs the enabled event via the extension on transition sequences, and backwards enabledness is simply the event abstraction of a transition. Note that $en_{\leftarrow}^{\mathsf{ESeq}} = en_{\leftarrow}^{\mathsf{Ev}}$ for event sequences with one event. We now define the extension transformers.

$$ext_{\rightarrow}^{\mathsf{ESeq}}(X) \mathrel{\hat{=}} \left\{\sigma e \mid \sigma \in X, e \in en_{\rightarrow}^{\mathsf{ESeq}}(\{\sigma\})\right\}$$
$$ext_{\leftarrow}^{\mathsf{ESeq}}(X) \mathrel{\hat{=}} (\alpha_{\mathsf{ESeq}} \circ ext_{\leftarrow} \circ \gamma_{\mathsf{ESeq}})(X)$$

The forward extension transformer concatenates a sequence with an enabled event while the backward extension transformer is a simple lifting.

Example 51. *Consider the event abstraction of Example 48. The forward extension of the event sequence e_ae_{ab} is $e_ae_{ab}e_{abc}$. Note that $en_{\rightarrow}^{\mathsf{ESeq}}(\{e_ae_{ab}\}) = \{e_{abc}\}$. Let $\sigma \mathrel{\hat{=}} (s, b, t)(t, c, u)$. It follows by definition that $\alpha_{\mathsf{ESeq}}(\{\sigma\}) = \{e_be_{bc}\}$ and $ext_{\leftarrow}(\{\sigma\}) = (r, a, s)(s, b, t)(t, c, u)$. The backward enabled event at the event sequence e_be_{bc} is $en_{\leftarrow}^{\mathsf{ESeq}}(\{e_be_{bc}\}) = \{e_a\}$. The backward extension $ext_{\leftarrow}^{\mathsf{ESeq}}(X) \mathrel{\hat{=}} \left\{e\sigma \mid \sigma \in X, e \in en_{\leftarrow}^{\mathsf{ESeq}}(\{\sigma\})\right\}$ of [DKS17] is unsound. Using this transformer, $\alpha_{\mathsf{ESeq}}(\{(r, a, s)(s, b, t)(t, c, u)\}) = \{e_ae_{ab}e_{abc}\} \notin ext_{\leftarrow}^{\mathsf{ESeq}}(\{e_be_{bc}\}) = \{e_ae_be_{bc}\}$.* ◁

We now present some examples of event sequence abstractions.

Histories The set of histories *Hist* consists of feasible, consistent transition sequences. The event abstraction Hist maps every history to itself and ignores other sequences. The least fixed point providing fireable histories when evaluated on event sequences will concretize to the set of fireable histories, representing the *unfolding* of a transition system (a synchronisation tree) [SNW96].

Transitions The domain Tr uses transitions in *Rel* as events where $ev_{\mathsf{Tr}}(\sigma(r,a,s)) \mathrel{\hat{=}} (r,a,s)$. The domain $\mathsf{ESeq}(\mathsf{Tr})$ of event sequences is equivalent to the domain of transition sequences. The use of prefixes in α_{ESeq} and γ_{ESeq} is necessary for this equivalence to arise. A simplistic lifting of events to event sequences that ignored prefix information would not lead to this equivalence.

Actions Consider *Act* to be the set of events and Act as the domain generated by $ev_{\mathsf{Act}}(\sigma(r,a,s)) \mathrel{\hat{=}} a$. The event sequences generated by the domain functor correspond to the language abstraction of the system, where only the sequence of labels is retained and not the underlying states. The least fixed point of fireable sequences generates the *Hoare language* for a system [SNW96].

6.2.1 Event Structure Representation

Consider a total event abstraction Ev on a set of events *Event* and the lifting to event sequences $\mathsf{ESeq}(\mathsf{Ev})$. Consider the event sequences $\mathcal{L}_{Event}$ obtained by abstracting the set of runs of a LTS M, i.e. $\mathcal{L}_{Event} \mathrel{\hat{=}} \alpha_{\mathsf{ESeq}}(Runs(M))$.

We now present a method to check whether $\mathcal{L}_{Event}$ can be represented as a prime event structure $\mathcal{E} \mathrel{\hat{=}} (Event, <, \#, \lambda{:}\, Event \to Act)$. Since we have an event abstraction, by Definition 33 the labels of the events are clearly defined. Thus, in the sequel we do not consider the labelling of events.

The decision procedure is as follows:

1. Check that there is no event sequence where an event e occurs more than once – the event abstraction is history dependent.

2. Partition $\mathcal{L}_{Event}$ by permutations, i.e. $\sigma \equiv \tau$ iff σ is a permutation of τ. This produces a set of equivalence classes $\mathcal{X} \mathrel{\hat{=}} \{X_1, X_2, \ldots, X_n\}$. Generate a set of partial orders by taking the intersection of each $X_i \in \mathcal{X}$. More formally, $\mathcal{P} \mathrel{\hat{=}} \{(E_1, <_1), \ldots, (E_n, <_n) \mid (E_i, <_i) \mathrel{\hat{=}} \bigcap X_i \in \mathcal{X}\}$.

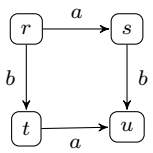
3. Check that for each $P_i \mathrel{\hat{=}} (E_i, <_i) \in \mathcal{P}$ it holds that: 1) $E_i \subseteq Event$ and 2) X_i is a realizer of P_i, i.e. $lin(P_i) = X_i$.

4. Finally, check that for each event $e \in \mathit{Event}$ their causes across different partial orders are the same. More formally, for all $e \in \mathit{Event}$, the set $\left\{\lceil e \rceil_{<_i} \in \wp(E_i) \mid e \in E_i\right\}$ is a singleton or the empty set.

We define the prime event structure $\mathcal{E} \hat{=} (\mathit{Event}, <, \#)$ where: 1) $< \hat{=} \bigcup <_i$ for $<_i \in \mathcal{P}$ and 2) $e \# e'$ iff $e \cup e' \notin E_i$.

We now informally show that if $\mathcal{L}_{\mathit{Event}}$ satisfies the conditions above, then the event structure is a representation of the language (closed under prefixes). That is, a sequence $\sigma \in \mathcal{L}_{\mathit{Event}}$ iff $\sigma \in \mathit{lin}(C_i)$ for some maximal configuration $C_i \in \mathit{conf}(\mathcal{E})$. Assume that $\mathcal{L}_{\mathit{Event}}$ satisfies conditions (1) through (4) above. If $\sigma \in \mathcal{L}_{\mathit{Event}}$, then we know that σ is a linearization of some P_i as $\mathcal{X}$ is a partition of $\mathcal{L}_{\mathit{Event}}$. By the construction of $\mathcal{E}$, it follows that P_i are maximal configurations. Thus, σ will be a linearization of one such maximal configuration. In the other direction, if σ is a linearization of a maximal configuration C_i, then since C_i is a partial order generated by the partition of $\mathcal{L}_{\mathit{Event}}$, it follows that $\sigma \in \mathcal{L}_{\mathit{Event}}$. We know that C_i is a partial order because any two events belong to C_i iff they belong to some partial order in $\mathcal{X}$.

Example 52. *We now provide an example of an event abstraction where the language of event sequences cannot be represented by a prime event structure and another abstraction where the event sequences can be represented by a prime event structure. Consider the following* LTS *with r as the initial state:*



Consider the event abstraction given by the following ev function:

$$ev((r,a,s)) \mapsto e_{a_1} \quad ev((r,a,s)(s,b,u)) \mapsto e_b$$
$$ev((r,b,t)) \mapsto e_b \quad ev((r,b,t)(t,a,u)) \mapsto e_{a_2}$$

The language of event sequences over the events Event $\hat{=} \{e_{a_1}, e_{a_2}, e_b\}$ is the set $\{e_{a_1}e_b, e_b e_{a_2}\}$. This language is not representable as an event structure since the event e_b as different causes failing check (4).

Now consider the event abstraction given by the following ev function:

$$ev((r,a,s)) \mapsto e_a \quad ev((r,a,s)(s,b,u)) \mapsto e_b$$
$$ev((r,b,t)) \mapsto e_b \quad ev((r,b,t)(t,a,u)) \mapsto e_a$$

The language of event sequences over the events Event $\hat{=} \{e_a, , e_b\}$ *is the set* $\{e_a e_b, e_b e_a\}$. *The partition of this set by permutations is the set* $\{\{e_a e_b, e_b e_a\}\}$. *Since this equivalence class is a realizer of the partial order* $(\{e_a, e_b\}, \varnothing)$, *this language satisfies all conditions above. As expected, the prime event structure is* $(\{e_a, e_b\}, \varnothing, \varnothing)$. ◁

6.3 Discussion

In this section we discuss related work and discuss the connection between events and independence.

Related Work. The idea of event as a derived notion arises in multiple models of concurrency. In particular, the notion of event as an element of a M-trace [Maz87] is one used up to this chapter. Another interesting notion of event arises in the model of transition systems with independence [SNW96]. These are standard labelled transition systems with an additional independence relation over the transitions that respect certain axioms. In this model, events correspond to sets of transitions which are part of local concurrency diamonds with other transitions (which will correspond to other events). However, to our knowledge, we are the first to use abstraction interpretation to formalize the idea of *events as abstraction*. In [CC94], Section 5, Cousot and Cousot define a Galois insertion for an equivalence relation in a general setting.

Independence Abstractions. From the definition of event in trace theory, we are able to derive the independence (or concurrency) relation between events. This amounts to checking whether two events are causally unordered and in the same M-trace. However, the notions of event and independence between events are definitely distinct.

Example 53. *Consider a system composed of two events a and b. Furthermore, consider that there are two possible sequences of events: ab and ba. Independence can be used to distinguish between the processes* $a \parallel b$ *and* $ab + ba$. *In the former, a*

is independent with b. This means that if we consider equivalence classes of event sequences, we would have $\{\{ab, ba\}\}$ *for* $a \parallel b$ *and* $\{\{ab\}, \{ba\}\}$ *for* $ab + ba$. ◁

In [DKS17], we separate these two concepts by treating events and independence as distinct abstractions. Independence generates an abstraction over the domain of event sequences which can be intuitively understood as a closure of behavior under concurrent interleavings.

Chapter 7

Conclusions and Open Problems

In this chapter we present our main conclusions and current open research problems.

Conclusions. In this dissertation, we have presented a new approach for practical verification of multithreaded programs based on event structures and data abstractions. One of the main goals was to re-use the rich theory of prime event structures as a semantics for concurrency and study its applicability as a data-structure for algorithmic state space exploration of **programs**.

We have found multiple advantages in using event structures:

- As we showed in Chapter 3, Mazurkiewicz trace theory, another mature semantics used in state space exploration, corresponds to a particular domain of prime event structures. This means that we can potentially obtain more reductions than optimal DPORs as we are using a more compact semantic object.
- Having a global view on the conflict relation provided a simple language of concepts to explain the requirements for optimal explorations. For example, the set D in our algorithms of Chapter 4 concisely captures parts of the state space that have been explored. Also, we believe the characterization of source sets presented in Section 4.1.1 using event structures is very intuitive and clear.
- We could build on previous work in Petri net unfoldings that use the theory of cutoffs to combine our explorations with early pruning based on states. This was to our knowledge a fundamental limitation of current DPOR methods as explained in Example 24. To our knowledge, this resulted in the first *super-optimal* DPOR method as shown in the experiments of Section 4.5.

As conveyed throughout this dissertation, the definition of *the prime event structure* of a state space fundamentally depends on the adopted notion of *event*. We have

showed two methods for defining a suitable notion of event which can be captured in prime event structures: 1) using the notion of independence of concurrent actions (as used since the early 1990s by PORs [God96]) and 2) in Section 6.1 by considering events as particular abstractions of sets of transition sequences.

We have developed most of our work using *independence* for two main reasons:

- In the context of stateless model checking, we could unify algorithm advantages of both DPORs and net unfoldings;
- Once we consider data abstractions, the interplay between independence and other abstractions clarified our notion of event in Chapter 5 as non-interfering invariants obtained with sequential abstract interpreters.

In the application of prime event structures to programs, we have stumbled at two fundamental limitations that affect the compactness of the structure: binary independence and conjunctive (AND) causality. These limitations have a great impact on the overall efficiency of the verification method. Fortunately, we know that the prime event structures studied in the previous chapters are at the bottom of the hierarchy of event-based models w.r.t. achievable compaction/reduction in the representation. Unfortunately, we do not have efficient algorithms to explore these more compact models yet. We found that these limitations were connected with other sources of explosion that occur in the presence of data.

In Chapter 5, we presented *causality-based abstract interpretation*, a multifaceted approach to address three sources of explosion: concurrency, data and control nondeterminism. Our main algorithmic contribution was a new combination of DPOR explorations over event structures with data abstractions in a nested fixpoint. Our approach solves the strictness of binary independence by considering abstract notions of independence that arise in interaction with data abstractions and we mitigate (AND) causality with the use of thread-local fixpoints provided by sequential abstract interpreters. These two components together with the natural generalization of cutoffs to domains resulted (as shown in Section 5.6) in speedups of orders of magnitude compared to symbolic PORs and high levels of precision compared to thread-modular abstract interpreters. Therefore, we have shown the potential of the combination of independence-based reasoning with data abstractions in efficient state space exploration of multithreaded programs.

Open Problems. Our main conclusion is that the state explosion problem can be mitigated by a combination of abstractions. Nevertheless, this only shifts the problem to the questions of *what abstractions to use* and *how to automatically refine abstractions*. In this dissertation, we investigated the interplay between two abstractions, binary independence and non-relational data abstractions in the style of abstract interpretation. We believe that a general method to combine independence relations in DPORs explorations with other abstractions is an under-explored research direction with practical consequences.

Designing efficient algorithms that use more relaxed kinds of independence relations or operate at the general level of equivalence classes (not necessarily partial order representations) is an open problem in algorithmic verification of concurrent programs. Current advanced DPORs [CCP+17, Hua15] are already able to leverage coarser notions of equivalence, e.g. to understand that in programs with n-concurrent writes to the same variable with distinct values only n interleavings suffice to explore the set of final states, but these approaches are not applicable in general. We believe that defining these semantics using independence in the framework of Chapter 4 could reveal new algorithmic insights.

Furthermore, independence has been used in forwards explorations of the interleaving semantics. An interesting direction is to define the dual *backwards independence* that could be used in backwards explorations. In particular, the main use of prime event structures in this work is as a compact representation of a set of partial orders by sharing their common prefixes. The definition of event structures that share common suffixes of partial orders and exploration algorithms of this structure is an interesting follow-up direction from our work. Our work in Chapter 5 already demonstrates that the idea of subsumption can be used and combined with DPORs to speed-up the exploration of the state space. We believe that an interesting idea to obtain further speed-ups would be to consider irreflexive independence relations. In this setting, $a \diamond_{\rightarrow} b$ denotes that the state reached with $a \cdot b$ is subsumed by the state reached with $b \cdot a$. To the best of our knowledge, it is an open problem to design such *directional* DPOR exploration that use irreflexive independence relations.

A conceptually straightforward follow-up direction towards more aggressive reductions from our work is to study new cutoff criteria for programs (as opposed to Petri nets) and their impact in the explorations.

Follow-up directions and unexplored problems related to Chapter 5 include:

- How to combine state-of-the-art symbolic execution or symbolic model checking with explorations of event structures?

- In the context of abstract unfoldings, how to apply our approach for non-relational domains and how to compute independence relations in these domains?
- How to automatically refine the domain to reduce false positives?

Finally, we believe that developing new applications of this work to other domains: e.g. weak memory models, programs using complex synchronization mechanisms and other models such as message passing models, could lead to novel algorithms.

In our opinion, advanced in these directions could provide new insights in how to use event structures as representations of proofs for concurrent programs where these structures can be automatically generated and refined.

Bibliography

[AAA+15a] 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. (Cited on pages 105 and 106.)

[AAA+15b] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 353–367. Springer, 2015. (Cited on pages 2, 60, and 109.)

[AAJS14] 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. (Cited on pages 2, 60, 65, 68, 69, 73, 74, 75, 82, 83, 89, 105, 111, 112, 142, and 143.)

[AAJS17a] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. *Comparing Source Sets and Persistent Sets for Partial Order Reduction*, pages 516–536. Springer International Publishing, 2017. (Cited on page 68.)

[AAJS17b] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Comparing source sets and persistent sets for partial order reduction. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 516–536, 2017. (Cited on page 112.)

[AAJS17c] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017. (Cited on page 112.)

[Aal87] IJsbrand Aalbersberg. *Studies in Trace Theory*. PhD thesis, 1987. (Cited on page 28.)

[AGIR18] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained dynamic partial order reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, Lecture Notes in Computer Science, pages 392–410. Springer, 2018. (Cited on page 141.)

[AJLS18] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, Lecture Notes in Computer Science, pages 229–248. Springer, 2018. (Cited on page 64.)

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013. (Cited on page 139.)

[BHK+14] Blai Bonet, Patrik Haslum, Victor Khomenko, Sylvie Thiébaux, and Walter Vogler. Recent advances in unfolding technique. *Theoretical Comp. Science*, 551:84–101, September 2014. (Cited on pages 61, 101, 112, and 135.)

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984. (Cited on pages 29 and 144.)

[BP82] R. Bonnet and M. Pouzet. *Linear Extensions of Ordered Sets*, pages 125–170. 1982. (Cited on page 16.)

[BR84] Stephen D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer, 1984. (Cited on page 29.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press. (Cited on page 125.)

[CC94] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California. (Cited on page 158.)

[CC00] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proc. of Principles of Programming Languages*, pages 12–25, New York, NY, USA, 2000. ACM Press. (Cited on page 21.)

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Principles of Programming Languages (POPL)*, pages 105–118. ACM, 2011. (Cited on page 137.)

[CCP+17] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 2(POPL):31:1–31:30, December 2017. (Cited on pages 3, 64, 112, and 162.)

[CH09] Jean-Loup Carre and Charles Hymans. From Single-thread to Multithreaded: An Efficient Static Analysis Algorithm. *arXiv:0910.5833 [cs]*, October 2009. (Cited on pages 114 and 141.)

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002. (Cited on pages 143 and 144.)

[Die95] Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995. (Cited on page 28.)

[DKS17] Vijay D'Silva, Daniel Kroening, and Marcelo Sousa. Independence abstractions and models of concurrency. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pages 151–168, 2017. (Cited on pages 9, 10, 154, 155, and 159.)

[DP90] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990. (Cited on page 13.)

[Dro90] Manfred Droste. Concurrency, automata and domains. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 1990. (Cited on page 29.)

[DS17] Vijay D'Silva and Marcelo Sousa. Complete abstractions and subclassical modal logics. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pages 169–186. Springer, 2017. (Cited on page 10.)

[EH08] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008. (Cited on pages 2, 57, 60, 100, and 113.)

[Eng91] Joost Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28(6):575–591, June 1991. (Cited on page 38.)

[ERV02] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002. (Cited on pages 4, 57, 61, 99, 104, 112, and 135.)

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, pages 110–121. ACM, 2005. (Cited on pages 2, 60, 64, 66, 68, 73, and 112.)

[FHRV13] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 37–47, New York, NY, USA, 2013. ACM. (Cited on page 142.)

[FK12] Azadeh Farzan and Zachary Kincaid. Verification of Parameterized Concurrent Programs by Modular Reasoning About Data and Control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 297–308, New York, NY, USA, 2012. ACM. (Cited on page 141.)

[FM07] Azadeh Farzan and P. Madhusudan. Causal Dataflow Analysis for Concurrent Programs. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 4424 in Lecture Notes in Computer Science, pages 102–116. Springer Berlin Heidelberg, 2007. (Cited on pages 130 and 141.)

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-Modular Model Checking. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software*, number 2648 in Lecture Notes in Computer Science, pages 213–224. Springer Berlin Heidelberg, May 2003. (Cited on page 141.)

[GFYS07] 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. (Cited on pages 2, 60, 105, and 112.)

[GLSW17] Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *LNCS*, pages 246–265. Springer, 2017. (Cited on page 142.)

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996. (Cited on pages 35, 36, 64, 112, and 161.)

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Principles of Programming Languages (POPL)*, pages 174–186. ACM, 1997. (Cited on page 2.)

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. of the ACM*, 47(2):361–416, 2000. (Cited on page 144.)

[Gup94] Vineet Gupta. *CHU Spaces: A Model of Concurrency*. PhD thesis, Stanford, CA, USA, 1994. UMI Order No. GAX95-08371. (Cited on page 3.)

[GW92] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, CAV '91, pages 332–342, London, UK, UK, 1992. Springer-Verlag. (Cited on page 3.)

[HRS+18] Nguyen Huyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. 2018. (Cited on pages 9, 107, and 111.)

[Hua15] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 165–174, 2015. (Cited on pages 112 and 162.)

[Jea12] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, March 2012. (Cited on page 141.)

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. (Cited on page 5.)

[KF13] Andrey Kupriyanov and Bernd Finkbeiner. Causality-based verification of multi-threaded programs. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, Lecture Notes in Computer Science, pages 257–272. Springer, 2013. (Cited on page 141.)

[KH14] 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. (Cited on page 112.)

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992. (Cited on pages 35 and 58.)

[KP93] Marta Z. Kwiatkowska and Iain Phillips. Concurrency and conflict in CSP. In *Theory and Formal Methods 1993, Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29-31 March 1993*, Workshops in Computing, pages 209–225. Springer, 1993. (Cited on page 29.)

[KSH14] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Unfolding based automated testing of multithreaded programs. *Automated Software Engineering*, pages 1–41, May 2014. (Cited on pages 58 and 142.)

[KW16] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Foundations of Software Engineering (FSE)*, pages 799–809. ACM, 2016. (Cited on pages 116 and 141.)

[KWG09] 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. (Cited on page 112.)

[LB99] Rom Langerak and Ed Brinksma. A complete finite prefix for process algebra. In *CAV*, pages 184–195. 1999. (Cited on page 57.)

[Maz87] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 278–324. Springer, 1987. (Cited on pages 28, 36, and 158.)

[McM93a] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of async. circuits. In *Proc. CAV'92*, volume 663 of *LNCS*, pages 164–177. Springer, 1993. (Cited on pages 2, 3, 4, 60, 98, 112, and 135.)

[McM93b] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. (Cited on page 141.)

[Min12] Antoine Miné. Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs. *Logical Methods in Computer Science*, 8(1), March 2012. (Cited on pages 114 and 141.)

[Min14] Antoine Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, number 8318 in Lecture Notes in Computer Science, pages 39–58. Springer Berlin Heidelberg, 2014. (Cited on pages 5, 114, 116, 139, and 141.)

[MM17] Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *LNCS*, pages 386–404. Springer, 2017. (Cited on pages 116 and 141.)

[MPR07] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise Thread-Modular Verification. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, number 4634 in Lecture Notes in Computer Science, pages 218–232. Springer Berlin Heidelberg, August 2007. (Cited on page 141.)

[MR87] E. T. Morgan and R. R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE-13(10):1080–1091, Oct 1987. (Cited on page 2.)

[MR95] Ugo Montanari and Francesca Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995. (Cited on page 58.)

[MT92] Madhavan Mukund and P. S. Thiagarajan. A logical characterization of well branching event structures. *Theor. Comput. Sci.*, 96(1):35–72, 1992. (Cited on page 38.)

[NPW81] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981. (Cited on pages 3, 24, and 26.)

[PE01] Julia Padberg and Hartmut Ehrig. Parameterized net classes: A uniform approach to petri net classes. In *Unifying Petri Nets, Advances in Petri Nets*, pages 173–229, 2001. (Cited on page 21.)

[Pet66] Carl Adam Petri. *Communication with automata*. PhD thesis, Universität Hamburg, 1966. (Cited on page 21.)

[Pra91] Vaughn Pratt. Modeling concurrency with geometry. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 311–322, New York, NY, USA, 1991. ACM. (Cited on page 3.)

[Pra00] Vaughan R. Pratt. Higher dimensional automata revisited. *Mathematical Structures in Computer Science*, 10(4):525–548, 2000. (Cited on page 23.)

[RSSK15a] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *Concurrency Theory (CONCUR)*, volume 42 of *Leibniz International Proceedings in Informatics*, pages 456–469. Dagstuhl Publishing, 2015. (Cited on pages 8, 9, 10, 52, 59, and 110.)

[RSSK15b] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. *CoRR*, abs/1507.00980, 2015. (Cited on pages 101 and 110.)

[RT91] Brigitte Rozoy and P. S. Thiagarajan. Event structures and trace monoids. *Theor. Comput. Sci.*, 91(2):285–313, 1991. (Cited on pages 41, 58, and 59.)

[SD16] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, New York, NY, USA, 2016. ACM. (Cited on page 10.)

[SDL18] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *Proc. ACM Program. Lang.*, 2(OOPSLA):165:1–165:29, October 2018. (Cited on page 10.)

[SDV+14] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. Consolidation of queries with user-defined functions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 554–564, New York, NY, USA, 2014. ACM. (Cited on page 11.)

[Shi97] M. W. Shields. *Semantics of Parallelism: Non-Interleaving Representation of Behaviour*. Springer-Verlag, Berlin, Heidelberg, 1997. (Cited on page 29.)

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. (Cited on page 13.)

[SNW96] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996. (Cited on pages 27, 29, 41, 145, 156, and 158.)

[Sou] Marcelo Sousa. POET - Partial Order Exploration Tool. (Cited on page 102.)

[SRDK17] Marcelo Sousa, César Rodríguez, Vijay D'Silva, and Daniel Kroening. Abstract interpretation with unfoldings. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 197–216, 2017. (Cited on pages 9 and 10.)

[SS13] Marcelo Sousa and Alper Sen. LLVMVF: A generic approach for verification of multicore software. *Journal of Electronic Testing*, 29(5):635–646, Oct 2013. (Cited on page 11.)

[Sta89] Eugene W. Stark. Concurrent transition systems. *Theor. Comput. Sci.*, 64(3):221–269, 1989. (Cited on page 29.)

[SV-] http://sv-comp.sosy-lab.org/2015/. (Cited on page 105.)

[Szp30] E. Szpilrajn. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, 16:386–389, 1930. (Cited on page 16.)

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515. Springer, 1991. (Cited on pages 3 and 112.)

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag. (Cited on page 2.)

[vGP95] Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*, pages 199–209, 1995. (Cited on page 21.)

[vGP04] Rob J. van Glabbeek and Gordon D. Plotkin. Event structures for resolvable conflict. In *Mathematical Foundations of Computer Science 2004, 29th International Symposium, MFCS 2004, Prague, Czech Republic, August 22-27, 2004, Proceedings*, pages 550–561, 2004. (Cited on page 21.)

[vGP09] Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures, event structures and petri nets. *Theor. Comput. Sci.*, 410(41):4111–4159, 2009. (Cited on pages 3, 21, 22, 23, 24, and 153.)

[VH17] Antti Valmari and Henri Hansen. Stubborn set intuition explained. *T. Petri Nets and Other Models of Concurrency*, 12:140–165, 2017. (Cited on page 3.)

[Win80] Glynn Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980. (Cited on page 21.)

[Win88] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, pages 364–397, 1988. (Cited on pages 3, 23, 144, and 153.)

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 210–217, 2013. (Cited on page 139.)

[YB12] Boris Yakobowski and Richard Bonichon. Frama-C's Mthread plug-in. Report, Software Reliability Laboratory, 2012. (Cited on pages 5, 116, and 140.)

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Model Checking Software (SPIN)*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008. (Cited on pages 2, 60, and 105.)

[YWY06] Xiaodong Yi, Ji Wang, and Xuejun Yang. Stateful dynamic partial-order reduction. In *Formal Methods and Sw. Eng.*, number 4260 in LNCS, pages 149–167. Springer, 2006. (Cited on pages 2, 60, and 112.)