Causality-based Verification

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Andrey Kupriyanov

Saarbrücken, 2016

Tag des Kolloquiums	28.06.2016
Dekan	Univ.-Prof. Dr. Frank-Olaf Schreyer

Prüfungsausschuss

Vorsitzender	Prof. Dr. Jörg Hoffmann
Berichterstattende	Prof. Dr. Bernd Finkbeiner Prof. Dr. Rupak Majumdar
Akademischer Mitarbeiter	Dr. Swen Jacobs

Abstract

Program verification is one of the central research topics in computer science since its inception – we can consider the field to be initiated as early as in 1949, with Alan Turing's pioneering paper "Checking a Large Routine." Yet, we are still far from the dream of automatically proving every computer program correct. Two aspects make this problem particularly challenging: concurrent program execution on parallel processors, and large, or even infinite, state spaces of data-manipulating programs. Nowadays, with concurrency entering everywhere, from smartphones to aircrafts, proving the correctness of infinite-state concurrent programs becomes increasingly more important: we do want to be sure that the program that controls the airplane we are flying in is correct.

In this thesis we propose a new approach to the verification of infinite-state concurrent programs. We call it *causality-based*, because it captures in an automatic proof system the "cause-effect" reasoning principles, which are often used informally in manual correctness proofs. While traditionally automatic methods are based on the state space exploration, our method is based on a new concurrency model, called *concurrent traces*, which are the abstractions of the history of a concurrent program to some key events and the relationships between them. Causality-based proof rules relate concurrent traces with each other, by formally tracking what are the necessary consequences (the "effects") from a particular analysis situation (the "cause"). The full correctness proof is then a composition of such primitive proof steps.

We study the syntactic and language-based properties of concurrent traces, and characterize the complexity of such operations as emptiness checking and language inclusion. Regarding the program correctness, we develop proof systems for the broad classes of safety and liveness properties, and provide algorithms for the automatic construction of correctness proofs. We demonstrate that for practically relevant classes of programs, such as multi-threaded programs with binary semaphores, the constructed proofs are of polynomial size, and can be also checked in polynomial time. The methods of the thesis have been implemented in Arctor, the first scalable termination prover for concurrent programs, which is able to handle programs with hundreds of non-trivial threads.

Zusammenfassung

Die Programmverifikation ist seit den Anfängen der Informatik eines ihrer zentralen Forschungsfelder. Als Beginn dieser Forschungsrichtung kann bereits das Jahr 1949 betrachtet werden, in dem Alan Turings bahnbrechende Arbeit "Checking a Large Routine" erschien. Der Traum, die Korrektheit von Programmen stets automatisch beweisen zu können, ist aber auch heute noch weit davon entfernt, Realität zu sein. Es gibt zwei Aspekte, die dieses Problem zu einer solch großen Herausforderung machen: die nebenläufige Ausführung von Programmen auf Parallelrechnern, und die großen, oder sogar unendlichen, Zustandsräume von datenverarbeitenden Programmen. Nebenläufige Programme werden in immer mehr Anwendungsbereichen, von Handys bis zur Luftfahrt, eingesetzt. Automatische Korrektheitsbeweise werden daher immer wichtiger: wenn wir mit dem Flugzeug reisen, möchten wir sicher sein, dass das Programm, das das Flugzeug steuert, auch tatsächlich korrekt ist.

In dieser Arbeit schlagen wir einen neuen Ansatz für die Verifikation von nebenläufigen Programmen mit unendlichem Zustandsraum vor. Wir nennen den Ansatz "kausalitätsbasiert", weil er im Rahmen eines automatischen Beweissystems die "Ursache-Wirkung"-Beziehungen erfasst, die sonst eher informell in manuellen Korrektheitsbeweisen benutzt werden. Anders als traditionelle automatische Methoden, die den Zustandsraums explorieren, baut unser Ansatz auf einem neuen nebenläufigen Berechnungsmodell, dem der "nebenläufigen Spuren", auf. Eine nebenläufige Spur ist eine Abstraktion der Vergangenheit eines nebenläufigen Programms im Hinblick auf bestimmte Schlüsselereignisse und die Beziehungen zwischen diesen Ereignissen. Kausalitätsbasierte Beweisregeln setzen nebenläufige Spuren zueinander in Bezug, indem die Konsequenzen (die "Wirkungen") einer bestimmten analytischen Situation (der "Ursache") auf eine formale Art und Weise verfolgt werden. Der vollständige Korrektheitsbeweis setzt sich dann aus solchen einfachen Beweisschritten zusammen.

Wir untersuchen die syntaktischen und sprachtheoretischen Eigenschaften von nebenläufigen Spuren, und charakterisieren die Komplexität von Operationen wie den Tests auf leere Sprache und Sprachinklusion. Wir entwickeln Beweissysteme zum Nachweis der Programmkorrektheit für die allgemeinen Klassen der Sicherheits- und Lebendigkeitseigenschaften, und stellen Algorithmen vor, die solche Beweise automatisch konstruieren. Für aus praktischer Sicht relevante Klassen von Programmen, wie Multi-Thread Programme mit binären Semaphoren, zeigen wir, dass die konstruierten Beweise polynomiell groß sind und auch in polynomieller Zeit geprüft werden können. Die in der Arbeit vorgestellten Methoden wurden im Verifikationswerkzeug ARCTOR implementiert. ARCTOR is der erste skalierbare Terminierungsbeweiser für nebenläufige Programme. Arctor kann Programme mit Hunderten nicht-trivialer Threads verarbeiten.

Acknowledgments

First and foremost, I would like thank my adviser Bernd Finkbeiner. I have used his open-door policy countless times, and also abused his semi-open one, to discuss my ideas. I thank him for his interest, patience, willingness to discuss everything up to and including implementation details, and for his ingenious remarks, capable of instantaneously rekindling ideas on the verge of abandonment and bringing back to life the most tangled and terrible mess. It is these qualities from which I have always drawn and will continue to draw inspiration and guidance throughout my career. It was only the infinite time Bernd devoted to me that permitted my ideas to crystallize into the present thesis.

I am grateful to Rupak Majumdar for reviewing my thesis, for joining my thesis committee, and for the interest he has shown in the topic of my research.

I want to thank all members of the Reactive Systems group at Saarland University, with whom I have had the pleasure to work together at different times: Rayna Dimitrova, Klaus Dräger, Rüdiger Ehlers, Peter Faymonville, Michael Gerke, Swen Jacobs, Felix Klein, Lars Kuhtz, Hans-Jörg Peter, Markus Rabe, Christa Schäfer, Leander Tentrup, Hazem Torfah, Martin Zimmermann. I will always cherish fond memories of our experiences together: our coffee breaks with endless opportunities to learn about German culture, from the language through the humor to politics, the delicious cakes you baked, our lovely walks in Saarbrücken, Freiburg, and Oldenburg, and many other things, far too numerous to list here.

I am grateful to the German Research Foundation (DFG) for supporting this work as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

I am indebted to my colleagues from the AVACS project – among them Matthias Heizmann, Jochen Hoenicke, Ernd-Rüdiger Olderog, Andreas Podelski, Mani Swaminathan, and all members of the Reactive Systems group – for the insightful discussions we had together.

Finally, I would like to thank my family – my wife Ksenia, and our daughter Uljana – for all the joy they bring into my life. Their unconditional love and unwavering support are the crucial foundation without which this thesis would never have materialized.

Contents

Chapter 1

Introduction

Concurrency fascinated many generations of computer scientists – the intriguing interdependence between concurrent events, and how they cooperate to achieve a common goal. A concurrent program can do many things *in parallel*, with its components either physically or logically distributed, thus achieving the processing speed unattainable by sequential programs. And precisely this power gives rise to problems: many more things can also go wrong.

This is where formal methods such as *verification* come in. They try to prove that a given program satisfies some specification, and, therefore, will serve its intended design purpose. Among verification methods, *model checking* attempts to construct a proof or to find a counterexample completely automatically – it provides a system designer with an appealing "push-button" approach to program verification. While very successful for sequential programs, model checking of concurrent programs suffers from the so-called *state space explosion problem*: the number of control states of a concurrent program grows exponentially fast with the addition of new components. The problem is exaggerated with large data domains for program variables.

In this thesis we focus on the verification of temporal properties for concurrent programs. The solution we describe aims to uncover the intricate interplay between dependency and independence. For concurrent programs, it is often the case that not many concurrent events depend on each other – most events are, in fact, independent, and precisely this allows concurrent programs to achieve better performance than sequential ones. On the other hand, the dependencies do exist. Moreover, they complicate the reasoning about concurrency to such extent that no general solution to the state space explosion problem is possible: there will always be programs requiring exponential resources for their analysis.

Despite this theoretical complexity, in many cases *a human reasoner* is able to devise either a short proof of correctness, or a small counterexample; this holds even for large systems, which are out of reach for any automatic method. Thus, the more general questions we want to answer are: "*why are systems built by humans often easy to reason about, despite high theoretical complexity?*", and "*can we, to some extent, capture the human intuition and reasoning power within a formal proof system?*". The more technical problem, which we seek a solution for, is "*can we produce polynomial proofs for concurrent programs?*"

Our approach aims to answer these questions by capturing *causality*. In most general terms, causality can be defined as the relation between two events, where the first event (the *cause*) is understood to be partly responsible for the second (the *effect*). Reasoning by causality is the usual style of constructing proofs: assume some situation (the effect) to be present, and derive all possible explanations (the causes). Consider the following proof from Leslie Lamport's paper [45] introducing the Bakery algorithm for mutual exclusion:

> **Assertion 1.** If processors i and k are in the bakery and i entered the bakery before k entered the doorway, then $number[i] < number[k]$.
>
> *Proof.* By hypothesis, $number[i]$ had its current value while k was choosing the current value of $number[k]$. Hence, k must have chosen $number[k] \geq 1 + number[i]$. □

Here, we assume the situation where the event "i entered the bakery" precedes the event "k entered the doorway", and, moreover, $number[i]$ preserves its value between two events. We derive from this situation another *necessary* fact (notice the words "*must have chosen*"): $(number[k] \geq 1 + number[i])$. In this thesis we propose a formal proof system as well as an automatic method of constructing concurrency proofs along the lines of causal reasoning.

Organization of the Thesis

- Chapter 2 introduces in more details the *verification problem*, puts it into a historical perspective, and discusses the related work.
- In Chapter 3 we formally define the necessary *preliminaries* such as SMT solving, temporal logic, and graph transformations. We also describe our model of *synchronized transition systems*, which we use to uniformly treat different flavors of concurrent programs.
- Chapter 4 introduces the model of *concurrent traces*, which captures sets of program computations in a succinct partial order representation. We discuss syntax, semantics, and different characterizations of concurrent traces, as well as computational complexity and algorithmic aspects of the standard language-theoretic notions such as emptiness, complementation and language inclusion.
- In Chapters 5 and 6 we show how transformations of concurrent traces, or *trace transformers*, can describe primitive proof steps, and how complete concurrency proofs are constructed as tableaux on top of them. We demonstrate that tableau-based proofs not only reflect the intuitive causal reasoning, but are often also of polynomial size. We specialize the the proof system and the verification algorithms for the two broad classes of temporal properties: *safety* and *liveness*.
- Chapter 7 briefly discusses the prototype implementation of the approach in a tool called ARCTOR, the first termination prover able to analyze termination of non-trivial concurrent programs with hundreds of processes.
- Finally, in Chapter 8 we conclude with the discussion of the possible directions for future research.

The results of the thesis were presented in the following publications:

- K Dräger, A. Kupriyanov, B. Finkbeiner and H. Wehrheim. SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems. In J. Esparza and R. Majumdar, eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010, LNCS 6015, pp. 271–274. Springer, 2010.

- A. Kupriyanov and B. Finkbeiner. Causality-based Verification of Multi-threaded Programs. In P. R. D'Argenio and H. C. Melgratti, eds., *Concurrency Theory (CONCUR)*, 2013, LNCS 8052, pp. 257–272. Springer, 2013.

- A.Kupriyanov and B.Finkbeiner. Causal Termination of Multi-threaded Programs. In A. Biere and R. Bloem, eds., *Computer Aided Verification (CAV)*, 2014, LNCS 8559, pp. 814–830. Springer, 2014.

Chapter 2

The Verification Problem

The problem we approach in the thesis is that of *verification*: given some program, probably written in a high-level programming language, and a specification of functional properties the program should meet, establish whether the program indeed satisfies the properties or not. The verification problem admits classification according to the class of programs and the class of properties we consider.

In this thesis we focus on the verification of *concurrent* programs. For the verification of *sequential* programs, known also as *software verification*, a wide variety of very successful techniques was developed, ranging from abstract interpretation to counterexample-guided abstraction refinement. The verification of concurrent programs is inherently more complex: already a control skeleton of a concurrent program is of exponential size with respect to the program description. Combined with usually large ranges of the program data variables, this gives rise to the infamous *state space explosion* problem, that limits the applicability of software verification methods.

With respect to the program properties we consider the standard classification into *safety* and *liveness* properties due to Lamport [46]. Informally, a safety property means that "something bad will *not* happen". A liveness property, on the other hand, expresses that "something good will *eventually* happen". This classification very clearly manifests itself in the shape of counterexamples: a counterexample to a safety property is always a *finite* program computation. On the other hand, a counterexample to a liveness property can be only an *infinite* program computation, as any finite counterexample computation, not containing the required good event, can be turned into a satisfying computation by appending the corresponding event.

The safety-liveness classification can be also approached from the point of view of *canonical* problems that belong to the corresponding property class. The canonical safety property is that of *reachability*: to decide whether some program state is reachable from the initial one. The canonical liveness property is *non-termination*: to decide, whether an infinite program computation (probably, with some additional restrictions) exists.

Finally, the language of *Linear Temporal Logic (LTL)* [64] combines both safety and liveness properties in a unified logical framework. As it is shown in [41], every temporal property can be represented as the intersection of a safety property and a liveness property.

2.1 Safety

The main characteristics that distinguishes safety from liveness is that properties in the former class can be specified as a set of *bad prefixes*. This set describes all violations to a safety properties, and contains only finite prefixes of program computations. Any extension of a prefix from the set remains a violation of the safety property. Here are some examples of safety properties and their corresponding language of bad prefixes:

- for a **local assertion** "l: `assert` φ" at location l, the language of bad prefixes is the set of program computations that can reach location l, and the formula φ evaluates to `false`.
- a more general case of a **safety invariant** asks whether some formula φ holds at all reachable program states. The bad prefixes are those program computations that reach a state where $\neg\varphi$ holds. The property of **reachability** is the exact opposite of invariance: here, for a formula φ, we ask whether there exist a reachable program state where φ holds.
- an example of a temporal safety property is **absence of unsolicited response**: whenever some response event β happens, a request event α should have happened before. The bad prefixes are exactly the program computations where event β happened, and there was no preceding event α.

Any safety property can be transformed to reachability by using *monitor automata* [42]. A monitor automaton for a safety property encodes the set of bad prefixes for the property: an error location in the monitor automaton is reached only by violating program computations. A monitor automaton can be then combined synchronously with the system; checking reachability of the monitor error location for the combined system is then equivalent to checking the safety property for the original system. This construction justifies calling reachability the canonical safety property: any method that can solve the reachability problem efficiently can be adapted for checking general safety properties.

Historically, the first approaches for safety analysis of programs are axiomatic proofs by Robert Floyd in 1967 [30] and Tony Hoare in 1969 [37], the so-called *Floyd-Hoare logic*. In this approach a proof of the shape $\{P\}S\{Q\}$ is constructed, where P is a pre-condition, S is a program, and Q is a post-condition. The above proof means that program S started in a state satisfying P, if it terminates, will end in a state that satisfies Q. The key points in the program control flow are annotated with state assertions, and the logic provides inference rules to derive new intermediate assertions and discharge proof obligations.

The Floyd-Hoare logic is targeted at sequential programs, where a clearly defined control flow of linear size is readily available. A concurrent program has a control flow of exponential size, and an efficient extension is not straightforward. This extension was nicely done by Susan Owicki and David Gries in 1976 [61, 60]. In their method control flows of individual processes are annotated with assertions similarly to the Floyd-Hoare logic. But in order for the proof to be valid, an additional condition of *non-interference* is imposed: an assertion in a given process should remain valid under arbitrary interference with all statements in concurrently evolving processes.

Floyd-Hoare and Owicki-Gries methods allow us to construct proofs of correctness for sequential and concurrent programs, respectively, which are small (of linear size), and efficiently verifiable (in linear and quadratic time, respectively). The only drawback of the approaches is that the assertions should be provided manually; this can be automated to some degree, but requires human ingenuity in the general case.

Addressing this drawback, *automatic* analysis of program properties, often referred to as *model checking*, started around 1980 by several authors; a nice account of its history is described in [14]. In model checking, we ask, given a program model M (often obtained by abstracting the real program), and a property in some temporal logic φ, whether the model satisfies the property, written $M \models \varphi$. On the theoretical side, the pioneering works of Edmund M. Clarke and E. Allen Emerson characterized correctness properties using fixed points [27, 15]. Approximately at the same time, Gerard Holzmann created at AT&T the network protocol analyzer *Pan* [39], which evolved later into the famous explicit-state model checker *SPIN* [40]. *Partial order reduction* methods such as [72, 63, 32] modify the search procedure of explicit-state model checking, and allow it to explore a reduced state space, by ignoring the order between independent transitions. *Symbolic model checking* [13] brought explicit-state model checking to the new level of efficiency by symbolically characterizing sets of states using BDDs. While partial order reduction and symbolic model checking do allow to analyze some concurrent designs in polynomial time (see, e.g., [55]), in most cases they still require exponential resources.

Concurrently to the model checking community, the *Abstract Interpretation* framework was evolving, starting with the foundational works of Patrick Cousot and Radhia Cousot [22, 23]. Traditionally, abstract interpretation has been limited to sequential programs and safety properties; recently, extensions to concurrent programs [57] and more general temporal properties [71] appeared. In this framework, the semantics of a program is abstracted by constructing some *abstract domain*, where the abstract semantics can be analyzed efficiently. Examples of abstract domains include the domain of intervals and the domain of octagons. The analysis in an abstract domain is usually very fast and scales to millions lines of code. The abstract program semantics overapproximates the concrete semantics (the method is incomplete): a proof in the abstract semantics is a valid proof for the program, but there may be false negatives, as an abstract error doesn't necessarily correspond to a real error.

Predicate abstraction [33] is a particular instance of an abstract domain, where the abstract states are valuations of a given set of predicates. Abstraction of a given program gives a finite-state boolean program [6], which can be efficiently model checked. As the initial set of predicates is often insufficient, predicate abstraction is usually equipped with the automatic predicate discovery in the framework of *Counterexample-guided abstraction refinement* (CEGAR) [16], thus making the method complete. This very successful combination was many times refined by different authors; here we would like to mention three refinements. *Lazy abstraction* [35] coined the idea of local refinement, when a new predicate is applied at a particular program location where it is relevant; this reduces the abstract state space substantially. Another refinement is the usage of Craig interpolants for predicate discovery [56], allowing to employ CEGAR-based model checking for infinite-state systems. Both of these works were limited to sequential programs; in the *Slicing Abstractions* framework [12] these

two lines of research were extended to concurrent programs. Author's own work on the model checker SLAB [25], which implements the ideas of [12], influenced to a large degree the development of the present thesis.

2.2 Liveness

Liveness properties are of a very distinct nature compared to safety properties: they can be violated only by an infinite computation. *Any* finite computation, even if the liveness property is not yet fulfilled, can be extended in such a way that the property is satisfied. Examples of liveness properties include:

- **Termination**: for a program P that performs some computational task, we expect that it finally terminates and delivers the computation result.
- Consider a set of processes, that coordinate their activity through the use of critical sections. We expect that any process wanting to enter its critical section will eventually succeed – this is the property of **accessibility**.
- For processes communicating over an unreliable channel, the property of **eventual reliability** requires that if some process keeps sending a message, it will be eventually delivered.

Similarly to invariance/reachability for safety, termination/non-termination can be referred to as the canonical liveness property. Proving termination relies on the discovery of *ranking functions*, which map the program state into a wellfounded domain. The classical approach due to Alan M. Turing is to build a single ranking function that strictly decreases in each step of the program [70]. Ranking function synthesis is a well-developed area; in particular, there are methods for the synthesis of ranking functions for such theories as linear arihmetic [17][65] and bit-vectors [18]. For quite some time, the common belief was that Counterexample-guided abstraction refinement (CEGAR) is limited to safety — until a new generation of CEGAR-based model checkers, notably the termination checkers *Terminator* [19] and *T2* [10, 20], proved capable of verifying the termination of difficult recursive functions, such as McCarthy's 91 function [50], as well as of reasonably complicated industrial software, such as device drivers. The CEGAR-based termination provers *Terminator* and *T2* build on the *Ramsey*-based approach, introduced by Podelski and Rybalchenko [66], which searches for a termination argument in the form of a *disjunction* of wellfounded relations. If the transitive closure of the transition relation is contained in the union of these relations, we call the disjunction a *transition invariant*; Ramsey's theorem then implies that the transition relation is wellfounded as well. The approach is attractive, because it is quite easy to find individual relations: one can look at the available program statements and take any decreasing transitions as hints for new relations. Unlike the model checkers for safety, however, the termination provers have been targeted to sequential programs only, and experiments show that they indeed scale badly for multi-threaded programs.

The first approach for proving liveness properties of concurrent programs was presented by Leslie Lamport in [47], where he has introduced the notion of proof lattices. Lattices are constructed from predicates, and arrows from some predicate P to the set of predicates $Q_1, \ldots, Q_n$ means that after a state

Algorithm 1: Verification as Proof Search

```
Input  : program P, property φ
Output: property holds/counterexample
Data: proof queue Q
begin
    Q ⟵ InitialAbstraction(P, φ)
    while not FixedPoint(Q) do
        take some q from Q
        if CheckSatisfiability(q, φ) then
            return counterexample
        else
            Q ⟵ Q ∪ GenerateSuccessors(q)
    return property holds
```

satisfying P appears in the computation, a state that satisfies one of $Q_1, \ldots, Q_n$ should appear thereafter. This work was generalized by Leslie Lamport and Susan Owicki in [62], where they replaced predicates in the lattice by arbitrary temporal formulas. The work of Zohar Manna and Amir Pnueli [51] extends the approach of [62] by allowing to use well-founded induction principles in diagram proofs, thus enabling proofs of liveness properties for infinite-state programs. The method has been automated in the interactive theorem prover STeP [8]. These works come closest to the approach developed in this thesis.

2.3 Verification as Proof Search

A substantial number of verification frameworks can be loosely modeled as a *proof search procedure*: see Algorithm 1. Here, given a program P and a property φ, we are interested in establishing the truth or finding a violation of φ by P. To make the discussion more concrete, we assume that the program P has a large but finite state space, and φ is a safety property. In the proof search view, we choose a *proof object*, and accumulate the set of objects seen so far in a *proof queue* Q. The search starts with some initial abstraction of the system with respect to the property. At each search iteration we select some object from the proof queue, and check whether it represents a property violation. If yes, we report a counterexample; otherwise we generate successors of the object, and put them into the proof queue. The search continues until either a counterexample is found, or our proof queue has reached a fixed point, when no new successors can be generated.

The concrete parameters of the above scheme can be instantiated differently, giving varying performance characteristics. We are interested in comparing the complexity of verification approaches for the case of concurrent programs; see Table 2.1. Here, for each verification method, we list what kind of proof objects it uses, what is the typical size of the proof with respect to the program description (**P** stands for polynomial, and **EXP** for exponential), and how expensive (in which complexity class) is the fixed point check with respect to the combined size of the program and the proof. We should note that this comparison is neither complete nor formal: it reflects the author's view on strengths and

Method	**Proof**	**Size**	**Fixed Point**
Explicit State	set of states	**EXP**	**P**
Predicate Abstraction	set of predicates	**P**	**PSPACE**
Lazy Abstraction	product of predicates	**EXP**	**P**
Automata-based	set of automata	**P**	**PSPACE**
Causality-based	tableau of traces	**P**	**NP / GI**

Table 2.1: Proof size and complexity for different verification methods

weaknesses of particular approaches. For example, there are some cases when *partial order reduction* for explicit-state model checking can reduce the state space of a concurrent program from exponential to polynomial; yet, there are even more cases when the state space after the reduction remains exponential.

For *explicit state model checking*, as represented, e.g., by the model checker SPIN [40], the proof object is an explicit program state (a valuation of its variables), and the proof queue is the set of states explored so far. Explicit state model checkers typically do explore the exponential number of states, but they do it very quickly (millions of states per second): this is because the proof object is so simple.

In the *predicate abstraction* framework, represented by such model checkers as SLAM [5] and ARMC [67], the proof object is a set of predicates, and the proof queue is the set of reachable abstract states represented by predicate valuations. Typically, the set of predicates is refined iteratively using the CEGAR paradigm [16]. The set of predicates is usually small, but at each iteration the whole set of reachable abstract states is constructed anew, and it has exponential size.

Lazy abstraction as introduced in [35], later refined by using Craig interpolants in [56], and brought to the concurrent setting in [12], is implemented in such model checkers as BLAST [7] and SLAB [25]. In this framework the abstract state space is not thrown away after each iteration, but preserved in order not to redo unnecessary work (as the refinement usually happens locally). As a result, the work done at each iteration is minimal, but the proof size as kept in memory becomes exponential.

In *automata-based* frameworks such as *Refinement of trace abstraction* [34] and *Inductive data flow graphs* [29], the proof object is either a set of automata, or a single automaton over the alphabet of program statements, which describes the set of program traces. The verification is successful, when the full set of program traces is included into the intersection of languages represented by the automata. Unfortunately, even if the automata themselves are of polynomial size, the later fixed point check is computationally expensive.

There is one particular point we want to make using the above comparison of different verification methods: no matter which proof object is used, there is always the same tradeof – either the proof has exponential size or its validation requires exponential time. Can we find a proof object and a verification method such that: a) most proofs for concurrent programs will have polynomial size, and b) the proof validation can be done in polynomial time?

The *causality-based* method of the present thesis is our (partial) answer to this question. It uses as a proof object *concurrent traces*, and the proof is a tableau built from them. In the following chapters we show that, on the one hand, concurrent traces are sufficiently succinct so that most concurrent pro-

grams do have polynomial proofs, and, on the other hand, the proof validation can be done in non-deterministic polynomial or quasipolynomial time. More specifically, the fixed point check for the tableau of concurrent traces amounts either to the precise language inclusion test, which, as we show later, can be done in **NP** under some restrictions, or to the under-approximating *concurrent trace inclusion*, which is an instance of the complexity class **GI** (for Graph Isomorphism). Although no polynomial algorithm is still known, recently a quasipolynomial algorithm for the problems in this class was described [4]. In practice, problems in **GI** can be solved very efficiently.

Chapter 3

Preliminaries

3.1 Assertion Language and SMT Solving

We are interested in the analysis of programs that comprise *data* such as integers, floats, bit-vectors, or arrays. In the verification literature such programs are generally described as *infinite-state*, although the infinity here is more of an abstraction, justified by the fact that the state space of such programs is too large to be handled explicitly. We abstract from concrete program statements and assume that their semantics can be expressed in some first-order assertion language; in the following we denote the set of quantifier-free formulas from the assertion language over some set of variables $\mathcal{V}$ by $\Phi(\mathcal{V})$.

To specify program semantics we need to define program transitions that modify program state, and, thus, connect two states: the current and the next one. For a set of variables $\mathcal{V}$, we define a set of *primed variables* $\mathcal{V}' \triangleq \{ x' \mid x \in \mathcal{V} \}$; primed variables represent the state of the program after executing a transition. We call formulas from the sets $\Phi(\mathcal{V})$ and $\Phi(\mathcal{V} \cup \mathcal{V}')$ *state predicates* and *transition predicates*, respectively. We denote by $\top$ and $\bot$ the boolean constants *true* and *false*, respectively (the valid and the unsatisfiable predicates).

We extend the primed variables notation to an arbitrary number of primes: this will allow us to specify sequences of program states of arbitrary length. For a finite set of variables $\mathcal{V}$, we define a countable family of primed variable sets $\mathcal{V}^k \triangleq \{ x^k \mid x \in \mathcal{V} \}$; we have that $\mathcal{V}^1 = \mathcal{V}'$, $\mathcal{V}^2 = \mathcal{V}''$, and so on. We denote the set of all primed variables up to some number k of primes by $\mathcal{V}_k = \bigcup_{i \in 0..k} \mathcal{V}^i$.

For the methods we develop further we rely on the existence of a *Satisfiability Modulo Theories* (SMT) solver that is able to handle assertions from the assertion language. A number of very efficient SMT solvers exist, and their performance is steadily improving; SMT competition [2] can serve as a good reference for solver comparison. We assume that an SMT solver satisfies the following requirements:

- For any $\varphi \in \Phi(\mathcal{V}_k)$, its satisfiability can be efficiently tested, i.e. there exists function $sat : \Phi(\mathcal{V}_k) \to \mathbb{B}$. We define auxiliary function $unsat(\varphi) \triangleq \neg sat(\varphi)$. We extend these functions to sets: for a set of assertions $\Gamma \subseteq \Phi(\mathcal{V}_k)$, let $sat(\Gamma) \triangleq sat(\bigwedge_{\varphi \in \Gamma} \varphi)$; similarly for *unsat*.
- For a set of assertions $\Gamma \subseteq \Phi(\mathcal{V}_k)$ such that their conjunction is unsatisfiable, an *unsatisfiable core*, i.e. a subset of assertions that is also

unsatisfiable, can be efficiently extracted. Formally, we assume the existence of the function $\mathit{unsat\_core} : \mathcal{P}\big(\Phi(\mathcal{V}_k)\big) \to \mathcal{P}\big(\Phi(\mathcal{V}_k)\big)$ such that $\mathit{unsat\_core}(\Gamma) \subseteq \Gamma \wedge \mathit{unsat}(\Gamma) \implies \mathit{unsat}(\mathit{unsat\_core}(\Gamma))$. We extend *unsat_core* to formulas: in that case, $\mathit{unsat\_core}(\varphi)$, where $\varphi \in \Phi(\mathcal{V}_k)$, is meant to be applied to the set of top-level conjuncts of φ.

- For any two formulas $\varphi_A \in \Phi(\mathcal{V}_{kA}), \varphi_B \in \Phi(\mathcal{V}_{kB})$ such that their conjunction is unsatisfiable, it is possible to compute a *Craig interpolant*: a formula φ that is implied by φ_A, contradicts φ_B, and is expressed using only the variables shared by both formulas. Formally, we assume the function $\mathit{interpolate} : \Phi(\mathcal{V}_{kA}) \times \Phi(\mathcal{V}_{kB}) \to \Phi(\mathcal{V}_{kA} \cap \mathcal{V}_{kB})$ such that if $\varphi = \mathit{interpolate}(\varphi_A\,;\,\varphi_B)$ then a) $\varphi_A \implies \varphi$, and b) $\mathit{unsat}(\varphi \wedge \varphi_B)$.

The above set of functions is enough for the purpose of the analysis of safety properties. For the analysis of liveness properties we additionally assume the existence of a *Ranking Function Synthesis* tool. This is also a well-developed area; in particular, there are methods for the synthesis of ranking functions for such theories as linear arihmetic [17][65] and bit-vectors [18]. Still, there is a big difference to the SMT solving: as the halting problem even for such simple models as counter machines is undecidable [58], it can be shown by an easy reduction that the ranking function synthesis is undecidable as well. Thus, we require the existence of a semi-decision procedure *rank*, described as follows:

- For any two formulas $\varphi_s, \varphi_c \in \Phi(\mathcal{V}_k)$, which collectively represent an SMT encoding of a lasso-shaped path, with φ_s encoding a *stem*, and φ_c encoding a *cycle*, a ranking function synthesis tool either produces a ranking function for the cycle, or it fails (e.g., after a timeout). Formally, we assume that there is a function $\mathit{rank} : \Phi(\mathcal{V}_k) \times \Phi(\mathcal{V}_k) \to \Phi(\mathcal{V} \cup \mathcal{V}') \cup \bot$ such that if $\psi = \mathit{rank}(\varphi_s, \varphi_c) \in \Phi(\mathcal{V} \cup \mathcal{V}')$, then for any sequence of states $s_0, s_1, \ldots, s_{i_1}, \ldots, s_{i_2}, \ldots$ we have the following. If a) $\varphi_s(s_0, s_1, \ldots, s_{i_1})$ holds, and b) for all $j \geq 1$, $\varphi_c(s_{i_j}, \ldots, s_{i_{j+1}})$ holds, then we have that for all $k \geq i_1$, $\psi(s_k, s_{k+1})$ holds, and ψ is a well-founded relation, i.e., there is no infinite chain $s_0, s_1, s_2, \ldots$ such that $\psi(s_k, s_{k+1})$ holds for all k.

We also use the following notations concerning assertions in the language:

- $\varphi_{[\mathcal{V}/\mathcal{V}']}$ denotes the substitution of $\mathcal{V}$ variables in formula φ by corresponding variables $\mathcal{V}'$.
- $\mathit{pres}(\mathcal{V}) \triangleq \bigwedge_{v \in \mathcal{V}} (v' = v)$ encodes the preservation of values for variables from $\mathcal{V}$.
- For a transition predicate $\tau \in \Phi(\mathcal{V} \cup \mathcal{V}')$, and a state predicate $\varphi \in \Phi(\mathcal{V})$, we denote by $\mathit{post}_\tau(\varphi) = \big(\exists V \,.\, \varphi(V) \wedge \tau(V, V')\big)_{[\mathcal{V}'/\mathcal{V}]}$ the post-condition of φ with respect to τ.
- Similarly, we denote by $\mathit{pre}_\tau(\varphi) = \exists V' \,.\, \varphi(V') \wedge \tau(V, V')$ the pre-condition of φ with respect to τ.

In this thesis we concentrate on the complexity stemming from concurrency, and want to abstract from the complexity associated with solving logical formulas. Therefore, we assume the existence of a *polynomial SMT oracle*: an SMT

solver that requires only polynomial time to answer a conjunctive SMT formula (with respect to the formula size). Note that the assumption holds strictly for simple theories, such as the quantifier-free theories of Equality and Uninterpreted Functions (EUF) or Difference Logic (DL), for which polynomial decision procedures exist.

3.2 Graphs and Graph Transformations

A *directed graph*, or simply a *graph*, is a tuple $G = \langle N, E \rangle$, where N is a set of nodes, and $E \subseteq N \times N$ is a set of edges. The source and target functions $s, t : E \to N$ map each edge to its first and second component, respectively. In this thesis we use exclusively *directed acyclic graphs* (*DAG*s): a DAG is such a graph $G = \langle N, E \rangle$ that no node is reachable from itself by following the edges: $\forall n \in N \,.\, (n, n) \notin E^*$.

Graph transformations define formal rules for transforming one graph into another by adding and deleting elements of the graph. We borrow the notation from [21, 26], where the authors describe the so-called *single-pushout (SPO)* and *double-pushout (DPO)* approaches to graph transformations.

Given two graphs $G = \langle N, E \rangle$ and $G' = \langle N', E' \rangle$, a *graph morphism* $f : G \to G'$ is a pair $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ of functions, preserving sources and targets: $f_N \circ t = t' \circ f_E$, and $f_N \circ s = s' \circ f_E$.

For our purposes, a *graph production* $p : (L \xrightarrow{r} R)$ is an injective graph morphism $r : L \to R$. The graphs L and R are called the *left-hand side* and the *right-hand side* of p, respectively. A given production $p : (L \xrightarrow{r} R)$ can be applied to a graph G if there is an occurrence of L in G, i.e. an injective graph morphism $m : L \to G$, called a *match*. In this case the resulting graph H can be obtained from G by adding all elements of R with no pre-image in L, removing all elements of L with no image in R, and contracting all elements of L with the same image in R. The application of a production p to a graph G with a match m is called a *direct derivation*; we will denote it with $p^m(G)$.

3.3 Linear Temporal Logic

As a specification language for system properties we use *Linear Temporal Logic* (*LTL*). Traditionally, state formulas are used as atomic propositions in LTL (state-based semantics); in the present work we use transition predicates (action-based semantics). For the finite-state case, formulas in one of the semantics can be encoded into another; for the infinite-state case, the action-based semantics is strictly more expressive than the state-based one.

For $\psi \in \Phi(\mathcal{V} \cup \mathcal{V}')$, LTL formulas are given by the following grammar:

$$\varphi \;::=\; \psi \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \varphi_1 \vee \varphi_2 \;\mid\; \neg\varphi \;\mid\; \bigcirc\varphi \;\mid\; \Box\varphi \;\mid\; \Diamond\varphi \;\mid\; \varphi_1 \,\mathcal{U}\, \varphi_2 \;\mid\; \varphi_1 \,\mathcal{R}\, \varphi_2$$

In the following we denote the set of LTL formulas over $\Phi(\mathcal{V} \cup \mathcal{V}')$ by Λ.

The temporal operators $\bigcirc$ (*Next*) and $\mathcal{U}$ (*Until*) form the temporal basis of LTL: the other temporal operators can be encoded by them. Nevertheless, the temporal operators $\Box$ (*Globally*), $\Diamond$ (*Finally*), and $\mathcal{R}$ (*Release*) are widely used in practice; therefore, we treat them explicitly. Having both $\mathcal{U}$ and $\mathcal{R}$ operators allows us to employ the so-called *Release positive normal form*: negations can

be propagated to the level of atomic formulas at the price of at most linear increase in the formula size.

LTL formulas are interpreted over a *model*, which is in our case a system computation $\pi = s_1, s_2, \ldots$ (we define transition systems formally in the next section; for now it is enough to see a transition system S as a generator of its computations). Given a model π, an atomic formula ψ, and temporal formulas φ_1, φ_2, we have the following inductive definition for the notion of the formula φ holding at position $j \geq 1$ in π, denoted by $(\pi, j) \models \varphi$:

$(\pi, j) \models \psi$	iff	$\psi(s_j, s_{j+1})$ holds
$(\pi, j) \models \varphi_1 \wedge \varphi_2$	iff	$(\pi, j) \models \varphi_1$ and $(\pi, j) \models \varphi_2$
$(\pi, j) \models \varphi_1 \vee \varphi_2$	iff	$(\pi, j) \models \varphi_1$ or $(\pi, j) \models \varphi_2$
$(\pi, j) \models \neg\varphi$	iff	$(\pi, j) \not\models \varphi$
$(\pi, j) \models \bigcirc\varphi$	iff	$(\pi, j+1) \models \varphi$
$(\pi, j) \models \Box\varphi$	iff	$\forall i \geq j\ (\pi, i) \models \varphi$
$(\pi, j) \models \Diamond\varphi$	iff	$\exists i \geq j$ s.t. $(\pi, i) \models \varphi$
$(\pi, j) \models \varphi_1 \mathcal{U} \varphi_2$	iff	$\exists k \geq j$ s.t. $(\pi, k) \models \varphi_2$ and $\forall j \leq i < k\ (\pi, i) \models \varphi_1$
$(\pi, j) \models \varphi_1 \mathcal{R} \varphi_2$	iff	$\forall i \geq j\ (\pi, i) \models \varphi_2$, or $\exists k \geq j$ s.t. $(\pi, k) \models \varphi_1$ and $\forall j \leq i \leq k\ (\pi, i) \models \varphi_2$.

We say that a transition system S satisfies the LTL specification φ, written $S \models \varphi$, if all computations π of S satisfy φ at position 1: $(\pi, 1) \models \varphi$. We denote by $\mathcal{L}_{\neg\varphi}(S)$ the set of *counter-models* of φ in S, i.e. the set of computations of S that *do not* satisfy φ at position 1. We verify that $S \models \varphi$ by proving that the set $\mathcal{L}_{\neg\varphi}(S)$ is empty; otherwise we provide a counterexample from $\mathcal{L}_{\neg\varphi}(S)$.

3.4 Transition Systems

For describing programs we use the simple and general formalism of *transition systems* [52, 53]. It serves as a basic model capable to directly represent sequential programs as well as concurrent programs with interleaving semantics. Later we extend it to the more general model of *synchronized transition systems*.

We assume the *vocabulary* $\mathcal{V}$ of variables, which is partitioned into the set of *flexible* variables V and *rigid* variables U. Flexible variables describe program control and data, and are allowed to change their value from one point in time to another. Rigid variables are used for specification purposes and are assumed to preserve their value throughout the program lifetime. Variables in $\mathcal{V}$ are typed, and each variable in $\mathcal{V}$ has its corresponding domain of values.

Definition 3.1 (**Transition system**)**.** A *transition system* is a tuple $S = \langle V, T, \Theta \rangle$, where:

- V is a finite set of (flexible) system variables;
- $T \subseteq \Phi(V \cup V')$ is a finite set of system transitions;
- $\Theta \in \Phi(V')$ is the *initial condition*: a predicate that characterizes the set of states where the program can start. Θ can be interpreted as a transition which brings the system into its initial state.

A *state* of S is a valuation of the system variables V, that assigns each flexible variable a value from its domain.

We call a sequence of system states $s_0, s_1, s_2, \ldots$ a *computation*. We call it a *system computation* of S, if:

1. $\Theta(s_1)$ holds, i.e. the computation starts in some initial state; and
2. for all $i \geq 1$, there is a system transition $t_i \in T$ such that $t_i(s_i, s_{i+1})$ holds.

We denote the set of system computations of S by $\mathcal{L}(S)$.

For illustrative purposes we use the following transition system of a specialized form that allows us to make comparisons of concurrent traces with finite automata.

Definition 3.2 (**Event transition system**)**.** An *event transition system* over a finite alphabet of events $\Sigma = \{\hat{a}, \hat{b}, \hat{c}, \ldots\}$ is a tuple $S = \langle V, T, \Theta \rangle$, where:

- $V = \{\, v_i \in \{0,1\} \mid i \in \Sigma \,\}$ is the set of variables, one per event. We use the notation $x \equiv v'_{\hat{x}} \neq v_{\hat{x}}$, $\neg x \equiv v'_{\hat{x}} = v_{\hat{x}}$, for an event $\hat{x} \in \Sigma$. That is, event $\hat{x}$ happens iff the corresponding variable x changes its value.
- $T = \{\, x \wedge \bigwedge_{\hat{y} \neq \hat{x}} \neg y \mid \hat{x} \in \Sigma \,\}$ is the set of transitions; at each transition exactly one event happens;
- $\Theta \equiv \top$.

The language of the event transition system is in one-to-one correspondence with the set $\Sigma^* \cup \Sigma^\omega$ of all (finite or infinite) strings over the event alphabet Σ.

Often programs have some fixed skeleton, called *control flow*: there is a finite set of locations L with a given initial location; each transition t is assumed to go from some start location s to some end location e, denoted graphically $s \xrightarrow{t} e$. We allow to extend any transition system with this construction.

Definition 3.3 (**Transition system with control flow**)**.** A transition system $S = \langle V, T, \Theta \rangle$, with a *control flow* over the set of locations L with initial location pc_0, implicitly defines another transition system $S' = \langle V', T', \Theta' \rangle$, where:

- $V' = V \cup \{pc\}$, for a fresh variable pc (program counter), defined over L;
- $T' = \{\, t \wedge (pc = s) \wedge (pc' = e) \mid \text{for all } t \in T \text{ such that } s \xrightarrow{t} e \,\}$;
- $\Theta' \equiv \Theta \wedge (pc' = pc_0)$.

We use the following shorthand notation: for a location l, we write l to denote the formula $pc = l$, and l' to denote the formula $pc' = l$.

We represent control flow graphically in figures, where locations are drawn as circles, and transitions are represented as arrows between locations annotated with transition predicates.

3.5 Synchronized Transition Systems

Our aim is the analysis of *concurrent programs* that comprise several components running in parallel and interacting in different ways to achieve a common goal. We model individual components using transition systems as defined above, and call them *processes* in this context. Interaction between processes may be organized using various mechanisms such as shared variables, binary/broadcast/lock-step synchronization, or communication channels. The methods we develop in the next chapters are sufficiently general to handle various types of interaction; therefore we want to abstract from a concrete interaction mechanism and propose a unified representation, which we call a *synchronized transition system*. The formulation we use is inspired by the representation of *labeled formal concurrent systems* of Godefroid in [32], and by *products of transition systems* of Esparza and Heljanko in [28], but we extend it in this thesis to be able to handle programs with infinite data.

Definition 3.4 (**Synchronized transition system**)**.** A *synchronized transition system* is a tuple $\mathbb{S} = \langle S_1, \ldots, S_n, \mathbb{T}, \Delta \rangle$, where:

- $S_i = \langle V_i, T_i, \Theta_i \rangle$, with $i \in \{1, \ldots, n\}$, are the constituent processes; we assume that for all $i, j \in \{1, \ldots, n\}$, T_i and T_j are disjoint sets. We define $V = V_1 \cup \ldots \cup V_n$ and $V' = V_1' \cup \ldots \cup V_n'$;
- $\mathbb{T} \subseteq \mathcal{P}(T_1 \cup \ldots \cup T_n)$ is a *synchronization constraint* such that for all $\tau \in \mathbb{T}$ we have $|\tau| > 0$ and $|\tau \cap T_i| \leq 1$;
- $\Delta : \mathbb{T} \to \Phi(V \cup V')$ is a *synchronization mechanism*. We require that if $\tau = \Delta\big(\{\tau_1, \ldots, \tau_k\}\big)$, then for all τ_i it holds that $\tau \Longrightarrow \tau_i$.

A synchronization constraint specifies which transitions from different processes can participate in a joint transition. At least one process should always participate, and each process can participate with at most one of its transitions. A synchronization constraint is usually represented implicitly, using such methods as synchronization of events in the common alphabet of the processes.

A synchronization mechanism, given a set of transitions selected by the synchronization constraint, defines how the relation of the composite transition is constructed from the relations of individual process transitions. The additional requirement ensures that the composite transition relation agrees with the changes that are allowed by its constituent transitions. A simple way to satisfy the requirement is to construct the composite transition relation as a conjunction of the constituent relations and the preservation constraint for the variables outside of the alphabets of participating processes.

Example 3.5 (**Synchronization methods**)**.** Here we show what synchronization constraints and mechanisms correspond to different ways of interaction. For a transition τ that synchronizes over a synchronization event a, we assume that synchronization is encoded as part of the transition relation (e.g., using a dedicated boolean variable $\hat{a}$), and define the shorthand $a \in \tau \triangleq sat(\tau \wedge \hat{a})$.

- with **interleaving semantics** a transition from any process can be executed independently; each transition preserves variables that are not in the alphabet of its process:

  - $\mathbb{T} = \big\{\, \{\tau\} \mid \tau \in T_1 \cup \ldots \cup T_n \,\big\}$;
  - $\Delta\big(\{\tau\}\big) = \tau \wedge \mathit{pres}(V \setminus V_i)$, for $\tau \in T_i$.

  Processes that are composed using interleaving semantics and operate on shared memory objects are often called *threads* in the literature; the corresponding programs are called *multi-threaded*. We will also use these notions for processes and programs with interleaving composition.

- with **binary synchronization** over an alphabet Σ, any two transitions from different processes that share complementary communication symbols, say $a!$ and $a?$, evolve synchronously, or a transition from a single process is executed, if it doesn't require a communication partner:

  - $\mathbb{T} = \big\{\, \{\tau_i \cup \tau_j\} \mid \tau_i \in T_i \wedge \tau_j \in T_j \wedge i \neq j \wedge a! \in \tau_i \wedge a? \in \tau_j \,\big\}$
    $\cup \big\{\, \{\tau\} \mid \tau \in T_1 \cup \ldots \cup T_n \,.\, \forall a \in \Sigma \,.\, a! \notin \tau \wedge a? \notin \tau \,\big\}$;
  - $\Delta\big(\{\tau_i, \tau_j\}\big) = \tau_i \wedge \tau_j \wedge \mathit{pres}(V \setminus V_i \setminus V_j)$, for $\tau_i \in T_i, \tau_j \in T_j$;
    $\Delta\big(\{\tau_i\}\big) = \tau \wedge \mathit{pres}(V \setminus V_i)$, for $\tau_i \in T_i$.

- related are **synchronous communication channels**. For a channel $\gamma \in \Sigma$, a sending action $\gamma \lessdot e$ and a receiving action $\gamma \gtrdot x$ complement each other, resulting in the assignment of expression e to variable x:

  - $\mathbb{T} = \big\{\, \{\tau_i \cup \tau_j\} \mid \tau_i \in T_i \wedge \tau_j \in T_j \wedge i \neq j \wedge \gamma \lessdot e \in \tau_i \wedge \gamma \gtrdot x \in \tau_j \,\big\}$
    $\cup \big\{\, \{\tau\} \mid \tau \in T_1 \cup \ldots \cup T_n \wedge \forall \gamma \in \Sigma \,.\, \gamma \lessdot e \notin \tau \wedge \gamma \gtrdot x \notin \tau \,\big\}$;
  - $\Delta\big(\{\tau_i, \tau_j\}\big) = \tau_i \wedge \tau_j \wedge x' = e \wedge \mathit{pres}(V \setminus V_i \setminus V_j)$, for $\tau_i \in T_i, \tau_j \in T_j$;
    $\Delta\big(\{\tau_i\}\big) = \tau \wedge \mathit{pres}(V \setminus V_i)$, for $\tau_i \in T_i$.

- **explicit synchronization** declares explicitly the constituent local transition for each global transition. Variables outside of the alphabets of participating processes preserve their values:

  - $\mathbb{T}$ is given as an explicit enumeration;
  - $\Delta(\tau) = \bigwedge_{\tau_i \in \tau}(\tau_i) \wedge \mathit{pres}\big(V \setminus \bigcup_{i.|\tau \cap T_i| = 0}(V_i)\big)$.

- in **lock-step synchronization** all processes do each step simultaneously. In this model each global transition corresponds to a conjunction of one transition from every process:

  - $\mathbb{T} = \big\{\, \{\tau_1 \cup \ldots \cup \tau_n\} \mid \forall i \,.\, \tau_i \in T_i \,\big\}$;
  - $\Delta\big(\{\tau_1 \cup \ldots \cup \tau_n\}\big) = \tau_1 \wedge \ldots \wedge \tau_n$.

A synchronized transition system induces a standard transition system in a natural way; we call such an induced transition system *fully expanded*.

Definition 3.6 (**Fully expanded system**)**.** For a synchronized transition system $\mathbb{S} = \langle S_1, \ldots, S_n, \mathbb{T}, \Delta \rangle$, where $S_i = \langle V_i, T_i, \Theta_i \rangle$ for $i \in \{1, \ldots, n\}$, we define a *fully expanded transition system* $S = \langle V, T, \Theta \rangle$ as follows:

- $V = V_1 \cup \ldots \cup V_n$;
- $T = \{\, \Delta(\tau) \mid \tau \in \mathbb{T} \,\}$;

- $\Theta = \Theta_1 \wedge \ldots \wedge \Theta_n$.

Please note that in some cases, such as lock-step synchronization, the number of transitions of the fully expanded system may be exponential compared to the description of the synchronized system. On the other hand, this definition allows us first to concentrate on generic analysis methods, applicable to any transition system, and only afterward extend them to more specialized methods for particular synchronization mechanisms.

Chapter 4

Concurrent Traces

In abstraction-based verification [16, 5, 7, 25], infinite state spaces are partitioned and represented compactly by using predicates, thus providing a reduction from infinite to finite. Yet, there is another source of difficulties, the state explosion problem [24] that is due to the combinatorial explosion of the number of states of a concurrent program.

A very natural way to describe analysis situations that arise in concurrency is to employ partial orders over events in a concurrent history. Examples of such representations include Event Structures of Glynn Winskell [73], Dependency Graphs in the Trace Theory of Antoni Mazurkiewicz [54], and, more recently, Concurrent Kleene Algebra of Tony Hoare et. al. [38]. The distinctive features of all these formalisms are the finiteness assumption for the set of events as well as the static dependency relation between events.

We combine both approaches of predicate abstraction and partial orders in a unified model that captures the concurrent evolution of a potentially infinite set of events; we call the structures we propose for that purpose *concurrent traces*.

In the next chapters we outline the methods for the verification of both safety and liveness properties. To deal with the former class of properties, *finite* concurrent traces are adequate; for the latter class we represent infinite program computations using *infinite* concurrent traces; they will be described in the next chapter.

We start by introducing the formal notion of a concurrent trace, define its language, and consider its syntactic properties. Then we consider, in order, the problems of intersection, union, emptiness, complementation, and language inclusion for concurrent traces. In order to provide efficient algorithms for these problems we formulate some restrictions on the general concurrent trace model.

4.1 Syntax and Semantics

Definition 4.1 (**Finite concurrent trace**). A *finite concurrent trace* is a tuple $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E}$ is a set of *events*; we assume that $\mathcal{E}$ always contains two special events $e_{\triangleleft}$ (*entry*) and $e_{\triangleright}$ (*exit*);

- $\mathcal{C} \subset \mathcal{E} \times \mathcal{E}$ is a set of *causal links*; for $(e, e') \in \mathcal{C}$ we write $e \preceq e'$; we call $\preceq$ the *causality relation*. We require that $\langle \mathcal{E}, \mathcal{C} \rangle$ is simple directed acyclic graph; i.e., there is at most one link between each pair of events, and $\preceq$ is acyclic. We require also that for all $e \in \mathcal{E}$ we have $e_\triangleleft \preceq e \preceq e_\triangleright$.

- $\lightning \subseteq \mathcal{E} \times \mathcal{E}$ is a symmetric *conflict relation*; we require that all events are in conflict with the exit event: for all $e \in \mathcal{E} \,.\, e \neq e_\triangleright \Longrightarrow e \lightning e_\triangleright$.

- $\lambda_\mathcal{E} : \mathcal{E} \to \Phi(\mathcal{V} \cup \mathcal{V}')$ and $\lambda_\mathcal{C} : \mathcal{C} \to \Phi(\mathcal{V} \cup \mathcal{V}')$ are labelings of events and causal links with transition predicates; we restrict $\lambda_\mathcal{E}(e_\triangleright) \in \Phi(\mathcal{V})$.

We use the source and target functions $src, tgt : \mathcal{C} \to \mathcal{E}$, which map each causal link to its first and second component, respectively. We denote the set of finite concurrent traces by $\mathcal{F}$, and call them *concurrent traces* or simply *traces* within this and next chapters. For a particular concurrent trace $F \in \mathcal{F}$, we write $\mathcal{E}(F)$ to denote its set of events, and $\mathcal{C}(F)$ to denote its set of causal links.

Note that we use the words "event" and "link" to describe concurrent traces, which are essentially labeled DAGs, instead of "node" and "edge": these are reserved for the later use to describe composite structures built on top of concurrent traces.

A concurrent trace describes a set of program computations. For a particular concurrent trace its events specify what state changes should necessarily occur in a computation, while its causal links represent the partial ordering between such changes and constrain the ones that occur in-between. While it is possible that several logically distinct state changes are performed by a single system transition, the conflict relation may restrict such transition sharing.

We postpone the formal description of concurrent trace language in favor of some examples to build the intuition.

Graphical Notation. We show event identities in circles, and labeling formulas in squares. Causal links are shown as solid lines with arrows, and conflicts as crossed zigzag lines. We omit any of these parts when they are not important or would create clutter in the current context.

Example 4.2 (Some concurrent traces). Consider the Figure 4.1; here we use the event transition system (see Example 3.2) with events a, b, c.

In the trace F_1, we require that events a, b, and c all happen in the computation, at some unspecified order. When there are several states in the computation that satisfy a, then the corresponding event can be matched to any such position in the computation.

In the trace F_2, we again require that all of the events a, b, and c happen, but this time we additionally require that, after we match event a, no other event a occurs in the computation; similarly for b and c. Thus, whenever a, b, and c occur in a computation, the trace F_2 can be matched in a *unique* way to the last positions where these events happened. Trace F_2 is *deterministic*: we will return to this notion later.

Formally, we describe the set of computations that a concurrent trace represents as follows.

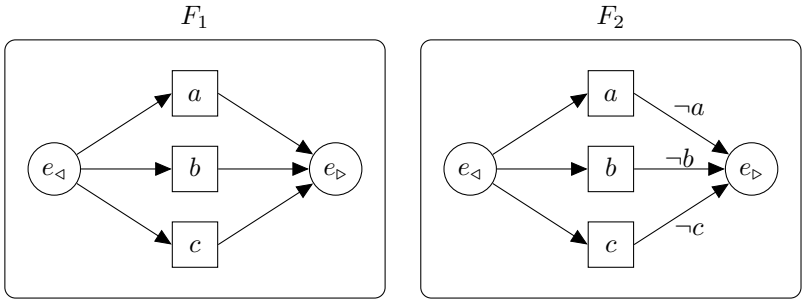


Figure 4.1: Examples of concurrent traces

Definition 4.3 (**Trace language**)**.** The *language* of a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ is defined as a set $\mathcal{L}(F)$ of finite computations such that for each computation $\pi = s_0, \ldots, s_n \in \mathcal{L}(F)$ there exists a mapping $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$, called a *run* of F on π, such that:

1. for each event $e \in \mathcal{E}$ and $s_i = \sigma(e)$ the formula $\lambda_\mathcal{E}(e)\big(s_i, s_{i+1}\big)$ holds; moreover, $s_0 = \sigma(e_\triangleleft)$, $s_n = \sigma(e_\triangleright)$;
2. for each causal link $c = (e_1, e_2) \in \mathcal{C}$, and $s_i = \sigma(e_1)$, $s_j = \sigma(e_2)$, we have
   a) $i \leq j$, and
   b) for all $i < k < j$, the formula $\lambda_\mathcal{C}(c)\big(s_k, s_{k+1}\big)$ holds;
3. for each pair of conflicting events $e_1 \not\downarrow e_2$, and $s_i = \sigma(e_1)$, $s_j = \sigma(e_2)$, we have $i \neq j$.

We call a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ *contradictory* if any of its events is labeled with an unsatisfiable predicate, i.e. if there exists $e \in \mathcal{E}$ such that $\mathit{unsat}(\lambda_\mathcal{E}(e))$ holds. Obviously, the language of such a trace is empty. We call a concurrent trace F *malformed*, if any condition in definition 4.1 is violated for F. We consider the language of such a trace empty as well. Malformed traces may result from applications of some transformation rules that are described later in this thesis.

The main motivation for introducing the computational model of concurrent traces is succinctness: concurrent traces can succinctly represent natural situations, arising in concurrency. Formally, we can state the following:

Proposition 4.4 (**Succinctness of concurrent traces**)**.** There is an indexed family of computation languages $\mathcal{L}^c(n)$, such that for each n the set of computations $\mathcal{L}^c(n)$ can be represented by a concurrent trace of size $O(n)$, but a minimal non-deterministic finite automaton accepting $\mathcal{L}^c(n)$ has 2^n states.

Proof. For a given n, let $A = \{a_1, \ldots, a_n\}$ be the alphabet of events. Let $\mathcal{L}^c(n)$ be the language of words containing all events from A:

$$\mathcal{L}^c(n) = \{\, s_1, \ldots, s_k \mid \forall a \in A \,.\, \exists i \in [1, k] \,.\, s_i \models a \,\}$$

Consider the following concurrent trace $F^c_n = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, where:

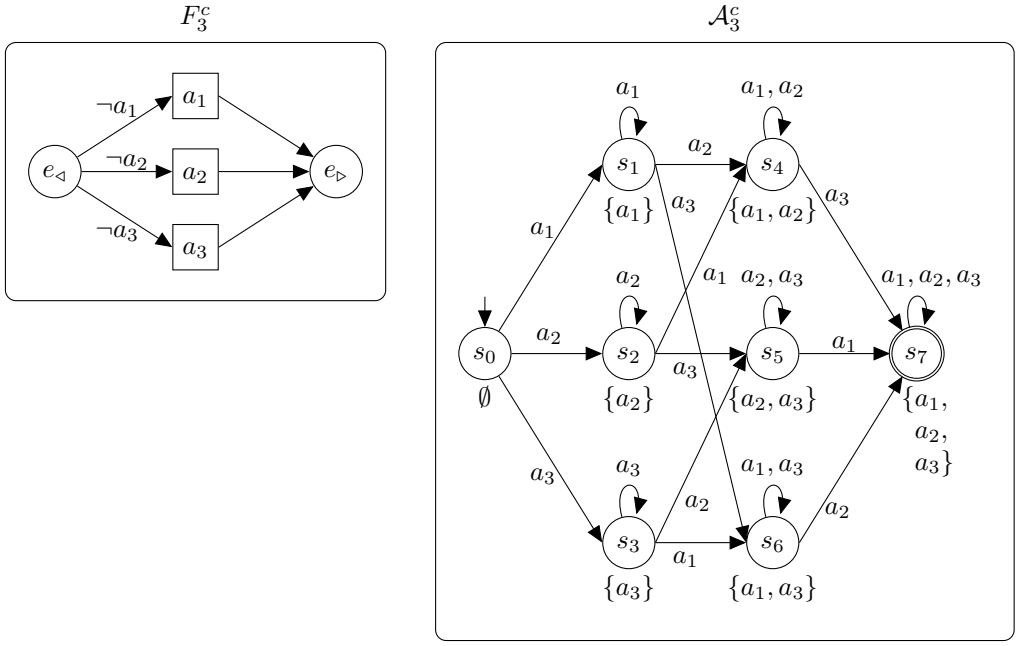


Figure 4.2: Succinctness of concurrent traces. The concurrent trace on the left and the finite automaton on the right accept the same language.

- $\mathcal{E} = \{e_\triangleleft, a^1, \ldots, a^n, e_\triangleright\}$;
- $\mathcal{C} = \{\, (e_\triangleleft, a), (a, e_\triangleright) \mid a \in \{a^1, \ldots, a^n\} \,\}$;
- $\not\downarrow \; = \mathcal{E} \times \mathcal{E} \setminus \{\, (e, e) \mid e \in \mathcal{E} \,\}$;
- $\lambda_\mathcal{E} = \{e_\triangleleft \to \top, e_\triangleright \to \top\} \cup \big\{\, a^i \to a_i \mid a^i \in \{a^1, \ldots, a^n\} \,\big\}$;
- $\lambda_\mathcal{C} = \big\{\, (e_\triangleleft, a^i) \to \neg a_i,\ (a^i, e_\triangleright) \to \top \mid a^i \in \{a^1, \ldots, a^n\} \,\big\}$.

An instance of F_n^c for $n = 3$ is shown in the left part of Figure 4.2. The trace F_n^c has $n + 2$ events, and it can be easily verified that it accepts the language $\mathcal{L}^c(n)$.

On the other hand, the minimal finite automaton for $n = 3$ has $2^n = 8$ states, and is shown in the right part of Figure 4.2. There, each automaton state is marked additionally with the set of events seen so far in the computation, and there should be one state for every subset from n events.

Formally, we prove, for arbitrary n, that the minimal finite automaton accepting $\mathcal{L}^c(n)$ cannot have less than 2^n states. Suppose the opposite, namely that there is an automaton $\mathcal{A}'_n$ with $N < 2^n$ states, which accepts $\mathcal{L}^c(n)$. Then, by pigeonhole principle, there should exist two computations π_1, π_2, with different sets $A_1 \neq A_2$ of events that occurred in them, but both computations leading from some initial state to the same state s of $\mathcal{A}'_n$. W.l.o.g., assume that $A_1 \setminus A_2 \neq \emptyset$. Now, consider the computation π', which is an arbitrary permutation of events from $A \setminus A_1$. Given π', $\mathcal{A}'_n$ should lead from s to an accepting state, because in the full computation $\pi_1 \cdot \pi'$ the whole set A of events is present.

But then the computation $\pi_2 \cdot \pi'$ should be also accepted, and the set of events in it is $A_2 \cup (A \setminus A_1) \subsetneq A$. Thus, computation $\pi_2 \cdot \pi'$ is accepted, but is not in the language $\mathcal{L}^c(n)$, a contradiction. □

We turn to the language-based characteristics of concurrent traces such as *emptiness* or *language inclusion* later in this chapter; here, we first define some syntactic characterizations and transformations.

Definition 4.5 (**Trace equality**)**.** We say that two traces $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ and $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$ are equal, denoted $F = F'$, if

1. $\mathcal{E}' = \mathcal{E}$, $\mathcal{C}' = \mathcal{C}$, $\not\downarrow' = \not\downarrow$;
2. $\forall e \in \mathcal{E} \,.\, \lambda_\mathcal{E}(e) \iff \lambda'_\mathcal{E}(e)$, and $\forall (e_1, e_2) \in \mathcal{C} \,.\, \lambda_\mathcal{C}\big((e_1, e_2)\big) \iff \lambda'_\mathcal{C}\big((e_1, e_2)\big)$.

Two traces are equal if they share the same set of events and causal links which are labeled by logically equivalent formulas in different traces. This notion is very restrictive, but useful in some contexts; obviously it can be decided in linear time. Less restricted notions are *trace inclusion* and *trace isomorphism*; to define them we first need to define the notion of *trace morphism*.

Definition 4.6 (**Trace morphism**)**.** Given two concurrent traces $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ and $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$, a *trace morphism* $\mu : F \to F'$ is a pair $\mu = \langle \mu_\mathcal{E} : \mathcal{E} \to \mathcal{E}', \mu_\mathcal{C} : \mathcal{C} \to \mathcal{C}' \rangle$ of injective mappings for events and causal links of one trace to those of another such that:

1. it preserves sources and targets: $\mu_\mathcal{E} \circ \mathit{tgt} = \mathit{tgt}' \circ \mu_\mathcal{C}$, and $\mu_\mathcal{E} \circ \mathit{src} = \mathit{src}' \circ \mu_\mathcal{C}$;
2. it maps entry and exit events of F to entry and exit events of F', i.e. it holds that $\mu_\mathcal{E}(e_\triangleleft) = e'_\triangleleft$ and $\mu_\mathcal{E}(e_\triangleright) = e'_\triangleright$.

Definition 4.7 (**Trace inclusion**)**.** For two concurrent traces $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ and $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$ we define the *trace inclusion relation* $\subseteq$ as follows: $F' \subseteq F$ iff there exists a trace morphism $\mu = \langle \mu_\mathcal{E} : \mathcal{E} \to \mathcal{E}', \mu_\mathcal{C} : \mathcal{C} \to \mathcal{C}' \rangle$ such that:

1. event labels of F' are stronger than those of F: for all $e \in \mathcal{E}$ we have $\lambda'_\mathcal{E}\big(\mu_\mathcal{E}(e)\big) \Longrightarrow \lambda_\mathcal{E}(e)$;
2. labels of causal links of F' are stronger than those of F: for all $c \in \mathcal{C}$ we have $\lambda'_\mathcal{C}\big(\mu_\mathcal{C}(c)\big) \Longrightarrow \lambda_\mathcal{C}(c)$;
3. conflicting events in F are mapped to conflicting events in F': for all $e_1 \not\downarrow e_2$ we have $\mu_\mathcal{E}(e_1) \not\downarrow' \mu_\mathcal{E}(e_2)$.

We write $F' \subseteq_\mu F$ if trace inclusion holds for a particular trace morphism μ.

Trace inclusion $F' \subseteq F$ holds if we can embed F into F'. When we require inclusion in both directions, we get trace isomorphism, a weaker variant of trace equality.

Definition 4.8 (**Trace isomorphism**)**.** We say that two traces F and F' are isomorphic, denoted $F \approx F'$, if both $F' \subseteq F$ and $F \subseteq F'$.

Both trace inclusion and trace isomorphism belong to the class **GI** (Graph Isomorphism). Although no polynomial algorithm is still known, recently a quasipolynomial algorithm for the problems in this class was described [4]. In practice, problems in **GI** can be solved very efficiently.

For a concurrent trace we can often deduce some deterministic facts about it. Consider an example, when a causal link (e_1, e_2) is *in scope* of some other link (e_1, e_3), i.e. when the following constellation of links exists: $(e_1, e_2), (e_2, e_3), (e_1, e_3) \in \mathcal{C}$. Then, whenever some event is in the scope of (e_1, e_2), it is also in the scope of (e_1, e_3). Thus, we can *deterministically* conjunct the label $\lambda_{\mathcal{C}}\big((e_1, e_3)\big)$ to the label $\lambda_{\mathcal{C}}\big((e_1, e_2)\big)$. We identify such cases as *deterministic proof rules*: these rules do not introduce any case distinctions. These rules are extensively described in Section 5.3; here we give a preliminary definition, which is sufficient for our immediate purposes.

Definition 4.9 (**Deterministic proof rules**)**.** We call *deterministic* any of the following proof rules which transform a trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ to another trace $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$. All elements of F', not mentioned explicitly, are equal to the corresponding elements of F.

- *LinkRestriction*: if $(e_1, e_2), (e_2, e_3), (e_1, e_3) \in \mathcal{C}$, then
  set $\lambda'_{\mathcal{C}}\big((e_1, e_2)\big) \longleftarrow \lambda_{\mathcal{C}}\big((e_1, e_2)\big) \wedge \lambda_{\mathcal{C}}\big((e_1, e_3)\big)$.

- *EventRestriction*: if $(e_1, e), (e, e_2), (e_1, e_2) \in \mathcal{C}$, $(e_1, e), (e, e_2) \in \not\downarrow$, then
  set $\lambda'_{\mathcal{E}}(e) \longleftarrow \lambda_{\mathcal{E}}(e) \wedge \lambda_{\mathcal{C}}\big((e_1, e_2)\big)$.

- *CausalTransitivity*: if $(e_1, e_2), (e_2, e_3) \in \mathcal{C}$, then
  set $\mathcal{C}' \longleftarrow \mathcal{C} \cup (e_1, e_3)$,
  $\lambda'_{\mathcal{C}}\big((e_1, e_3)\big) \longleftarrow \lambda_{\mathcal{C}}\big((e_1, e_2)\big) \vee \lambda_{\mathcal{E}}(e_2) \vee \lambda_{\mathcal{C}}\big((e_2, e_3)\big)$.

- *ConflictTransitivity*: if $(e_1, e_2), (e_2, e_3) \in \mathcal{C}$, $(e_1, e_2) \in \not\downarrow$ then
  set $\not\downarrow' \longleftarrow \not\downarrow \cup (e_1, e_3)$.

We denote the set of deterministic proof rules by *DPR*. A proof rule $R \in DPR$ can be applied in several places for a particular trace; therefore, $R : \mathcal{F} \to \mathcal{P}(\mathcal{F})$ maps traces to sets of traces. The set of deterministic proof rules induce a partial ordering between concurrent traces: $F \preccurlyeq_{DPR} F'$ when there is a rule $R \in DPR$ such that $F' \in R(F)$.

Deterministic proof rules increase the trace size at most quadratically; we assume that they are applied whenever possible till saturation. We call such deterministically saturated traces to be in *normal form*. In the graphical notation though, we draw only the minimal set of trace elements, which is enough to derive the trace normal form.

Definition 4.10 (**Normal form**)**.** A concurrent trace F is said to be in *normal form* when any application of a deterministic proof rule R doesn't change the trace, i.e., when for all $F' \in R(F)$ we have that $F' = F$. A trace in normal form is maximal in the partial order $\preccurlyeq_{DPR}$.

Deterministic proof rules do not change the language of a concurrent trace. We consider also the proof rules that may *restrict* the language of a trace to some subset; we call them *restricting proof rules* and denote *RPR*.

Definition 4.11 (**Restricting proof rules**)**.** We call *restricting* any of the following proof rules which transform a trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ to another trace $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$. All elements of F', not mentioned explicitly, are equal to the corresponding elements of F.

- *EventOrdering*: if $(e_1, e_2), (e_2, e_1) \notin \mathcal{C}^*$, then
  set $\mathcal{C}' \longleftarrow \mathcal{C} \cup (e_1, e_2)$, $\lambda'_{\mathcal{C}}\big((e_1, e_2)\big) \longleftarrow \top$.

- *EventContraction*: if $(e_1, e_2) \notin \not\downarrow$, then
  set $\mathcal{E}' \longleftarrow \mathcal{E} \setminus e_1 \setminus e_2 \cup e'$, $\lambda'_{\mathcal{E}}(e') \longleftarrow \lambda_{\mathcal{E}}(e_1) \wedge \lambda_{\mathcal{E}}(e_2)$,
  $\mathcal{C}' \longleftarrow \{\, (x, y) \in \mathcal{C} \mid x, y \notin \{e_1, e_2\} \,\} \cup$
  $\{\, (e', y) \in \mathcal{C} \mid x \in \{e_1, e_2\}, y \notin \{e_1, e_2\} \,\} \cup$
  $\{\, (x, e') \in \mathcal{C} \mid x \notin \{e_1, e_2\}, y \in \{e_1, e_2\} \,\}$,
  $\not\downarrow' \longleftarrow \{\, (x, y) \in \not\downarrow \mid x, y \notin \{e_1, e_2\} \,\} \cup$
  $\{\, (e', y) \in \not\downarrow \mid x \in \{e_1, e_2\}, y \notin \{e_1, e_2\} \,\} \cup$
  $\{\, (x, e') \in \not\downarrow \mid x \notin \{e_1, e_2\}, y \in \{e_1, e_2\} \,\}$,
  $\lambda'_{\mathcal{C}}$ is equal to the corresponding elements of $\lambda_{\mathcal{C}}$.

The set of restricting proof rules induce a partial ordering between concurrent traces: $F \preccurlyeq_{RPR} F'$ when there is a rule $R \in RPR$ such that $F' \in R(F)$.

Application of restricting proof rules till saturation leads to two subclasses of concurrent traces: *linear* and *compact*.

Definition 4.12 (**Linear trace, linearization**)**.** A *linear trace* is a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where the transitive closure of the causality relation $\mathcal{C}$ is total: for all $e, e' \in \mathcal{E}$ we have that $(e, e') \in \mathcal{C}^*$ or $(e', e) \in \mathcal{C}^*$. We denote the set of finite linear traces by $\mathcal{F}_L$.

We say that a linear trace $F' \in \mathcal{F}_L$ is a *linearization* of F, if F' can be obtained from F as a result of a sequence of applications of the rule *EventOrdering*. We denote the set of linearizations of F by *Linearizations*(F).

Definition 4.13 (**Compact trace, contraction**)**.** A *compact trace* is a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where the conflict relation $\not\downarrow$ is total: for all $e, e' \in \mathcal{E}$ we have that $(e, e') \in \not\downarrow$. We denote the set of finite compact traces by $\mathcal{F}_C$.

We say that a compact trace $F' \in \mathcal{F}_C$ is a *contraction* of F, if F' can be obtained from F as a result of a sequence of applications of *EventContraction*. We denote the set of contractions of F by *Contractions*(F).

Note that there can be exponentially many possible linearizations and contractions of a given concurrent trace. For the former, take the trace that consists of n unordered events (not counting $e_\triangleleft$ and $e_\triangleright$): there are $n!$ linearizations. For the latter, take n triples of events such that all events in different triples are in conflict, and within each triple e_1, e_2, e_3, only e_1 and e_2 are in conflict. Then e_3 can be contracted with either e_1 or e_2; this choice can be done independently for each triple, and we have 2^n possible contractions.

One obvious notion of size for a concurrent trace is the number of events $|\mathcal{E}|$ in it. But, as events can be contracted, this measure doesn't allow to assess, e.g., the shortest computation which a trace accepts. Contractions allow us to define this kind of measure.

Definition 4.14 (**Trace size**)**.** We define the *size* $|F|$ of a concurrent trace F as the minimum of the number of events between all contractions of F:

$$|F| \triangleq \mathit{min}\ \{\ |\mathcal{E}(F')|\ \big|\ F' \in \mathit{Contractions}(F)\ \}.$$

Linearizations, on the other hand, allow us to partition the language of a concurrent trace.

Proposition 4.15. For any concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$,

$$\mathcal{L}(F) = \bigcup \{\ \mathcal{L}(F')\ |\ F' \in \mathit{Linearizations}(F)\ \}.$$

Proof. Take a computation $\pi = s_0, \ldots, s_n \in \mathcal{L}(F)$; then there exists a run $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$ of F on π. Construct a linearization F' of F by adding all causal links (e_1, e_2) such that $\sigma(e_1) = s_i$ and $\sigma(e_2) = s_{i+1}$, for some i, but $(e_1, e_2) \notin \mathcal{C}^*$. Obviously, σ is also a run of F' on π.

Now, take a computation $\pi \in \mathcal{L}(F')$ for some linearization F' of F: then any run σ of F' on π is also a run of F, because F imposes less constraints on σ. □

A linear trace admits a particularly useful and simple variant of the normal form. For an arbitrary linear trace $F \in \mathcal{F}_L$ we can construct it as follows: apply the deterministic proof rules *EventRestriction* and *LinkRestriction* till saturation, and then keep only the links between the neighboring events in the trace.

Definition 4.16 (**Linear normal form**)**.** Let $F \in \mathcal{F}_L$ be a linear trace. The events of F are ordered; thus, we can write them as a sequence $\mathcal{E}(F) = \{e_1, e_2, \ldots, e_n\}$. Let F' be the result of applying the rules *EventRestriction* and *LinkRestriction* to F till saturation. We define the *linear normal form* $N_L(F)$ as a sequence of formulas $\varphi_1, \psi_1, \varphi_2, \psi_2, \ldots, \psi_{n-1}, \varphi_n$, where $\varphi_i = \lambda_{\mathcal{E}}(e_i)$, and $\psi_i = \lambda_{\mathcal{C}}\big((e_i, e_{i+1})\big)$.

4.2 Language Intersection, Union, Emptiness

Here we study the language-based properties of the concurrent trace model. The first one is *language intersection*: given two concurrent traces F_1 and F_2, find another concurrent trace F such that $\mathcal{L}(F) = \mathcal{L}(F_1) \cap \mathcal{L}(F_2)$. Concurrent traces are in their nature *conjunctive* objects: *all* trace events should be present in the computation, and *all* state changes in the scope of a causal link should satisfy its label. Taking this into account, it is clear that juxtaposition of two concurrent traces will represent the intersection of their languages.

Proposition 4.17. Given two concurrent traces $F_1 = \langle \mathcal{E}_1, \mathcal{C}_1, \not\downarrow_1, \lambda_{\mathcal{E}_1}, \lambda_{\mathcal{C}_1} \rangle$ and $F_2 = \langle \mathcal{E}_2, \mathcal{C}_2, \not\downarrow_2, \lambda_{\mathcal{E}_2}, \lambda_{\mathcal{C}_2} \rangle$, the intersection of their languages is represented by a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \mathcal{E}_1 \uplus \mathcal{E}_2$, with $e_\triangleleft$ and $e_\triangleright$ from $\mathcal{E}_1$ and $\mathcal{E}_2$ identified;
- $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$, with a single entry for $(e_\triangleleft, e_\triangleright)$;
- $\not\downarrow\ = \not\downarrow_1 \cup \not\downarrow_2$, with a single entry for $(e_\triangleleft, e_\triangleright)$;
- $\lambda_{\mathcal{E}} = \lambda_{\mathcal{E}_1} \cup \lambda_{\mathcal{E}_2}$, with $\lambda_{\mathcal{E}}(e_\triangleleft) = \lambda_{\mathcal{E}_1}(e_\triangleleft) \wedge \lambda_{\mathcal{E}_2}(e_\triangleleft)$, $\lambda_{\mathcal{E}}(e_\triangleright) = \lambda_{\mathcal{E}_1}(e_\triangleright) \wedge \lambda_{\mathcal{E}_2}(e_\triangleright)$;

- $\lambda_{\mathcal{C}} = \lambda_{\mathcal{C}_1} \cup \lambda_{\mathcal{C}_2}$, with $\lambda_{\mathcal{C}}\big((e_\triangleleft, e_\triangleright)\big) = \lambda_{\mathcal{C}_1}\big((e_\triangleleft, e_\triangleright)\big) \wedge \lambda_{\mathcal{C}_2}\big((e_\triangleleft, e_\triangleright)\big)$.

Proof. Given a computation $\pi = s_0, \ldots, s_n \in \mathcal{L}(F_1) \cap \mathcal{L}(F_2)$, there exists a pair of runs $\sigma_1 : \mathcal{E}_1 \to \{s_0, \ldots, s_n\}$ and $\sigma_2 : \mathcal{E}_2 \to \{s_0, \ldots, s_n\}$ of F_1 and F_2 on π. Then $\sigma_1 \cup \sigma_2$ is a run of F on π. Conversly, given a computation $\pi = s_0, \ldots, s_n \in \mathcal{L}(F)$, the run $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$ gives a pair of runs for both traces F_1 and F_2. □

For an example of language intersection, consider Figure 4.3; x, y, z, u, v, w are boolean variables. Concurrent trace F_1 specifies that states satisfying x and y should happen in order, and all states in between should satisfy z. Similarly, F_2 specifies that states satisfying u and v should occur sequentially, and all intermediate states should satisfy w. A trace on the right represents the intersection of the languages of both traces, and it is achieved by putting the traces side by side.

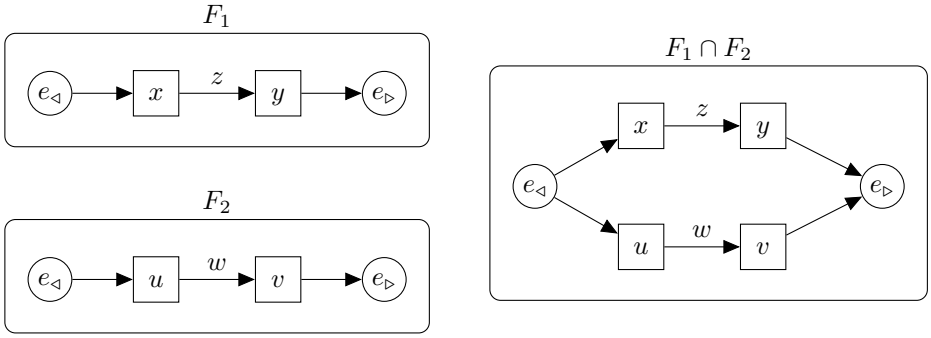


Figure 4.3: Illustration of language intersection for concurrent traces

Now we turn to *language union*. As concurrent traces are conjunctive objects, it is easy to show that a union of languages of two concurrent traces cannot be represented by a single concurrent trace.

Proposition 4.18. There are concurrent traces F_1, F_2 such that the union of their languages $\mathcal{L}(F_1) \cup \mathcal{L}(F_2)$ is not representable by any concurrent trace.

Proof. Take as F_1 and F_2 the traces from Figure 4.4. Suppose there is a concurrent trace F that accepts the union of their languages. Then, it should accept the computation $\pi_1 = (\neg x \wedge \neg y \wedge \neg u), (x \wedge \neg y \wedge \neg u), (\neg x \wedge \neg y \wedge \neg u), (\neg x \wedge y \wedge \neg u)$, and reject the computations $\pi_2 = (\neg x \wedge \neg y \wedge \neg u), (\neg x \wedge \neg y \wedge \neg u), (\neg x \wedge y \wedge \neg u)$, and $\pi_3 = (\neg x \wedge \neg y \wedge \neg u), (x \wedge \neg y \wedge \neg u), (\neg x \wedge \neg y \wedge \neg u)$. It can be easily seen, that for that F should contain at least two distinct events e_1, e_2 such that $\lambda_{\mathcal{E}}(e_1) \implies x$, $\lambda_{\mathcal{E}}(e_2) \implies y$, and $e_1 \preceq e_2$. By a similar reasoning, the trace should contain an event e_3 such that $\lambda_{\mathcal{E}}(e_3) \implies u$. By considering different cases with respect of e_3 being the same as or different from e_1, e_2, we obtain that either F accepts some computation not from $\mathcal{L}(F_1) \cup \mathcal{L}(F_2)$, or doesn't accept some computations from that language. □

From the above proposition it is clear that a set of traces is necessary, in general, to represent the union of languages of concurrent traces.

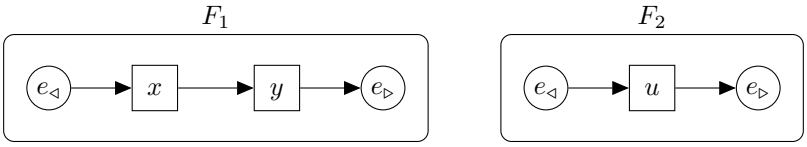


Figure 4.4: Illustration of language union for concurrent traces

Now we turn our attention to the most fundamental question for verification, namely *emptiness checking*: given a concurrent trace F, decide whether its language $\mathcal{L}(F)$ is empty. Our traces are parametrized by the logic used for the labels of events and causal links; but it is easy to show that checking emptiness becomes undecidable already for very simple logics.

Theorem 4.19. *The emptiness problem for concurrent traces over any logic that includes atoms of the form $x - y = c$, for variables x, y ranging over $\mathbb{N}$, and constants $c \in [0, 1]$, is undecidable.*

Proof. We show that we can reduce the halting problem of an arbitrary Minsky machine to the emptiness problem of a concurrent trace over the outlined logic. As the halting problem is undecidable already for Minsky machines with two counters [58], we get the desired result. Let a Minsky machine $M = \langle C, P \rangle$ with a set C of counters (or registers) and a list P of numbered instructions be given. We proceed in two steps.

1. We construct a transition system $S = \langle V, T, \Theta \rangle$ that encodes all computations of M. In the following, for any $x, y \in V$, we write $x = y$ instead of $x - y = 0$; this allows us, for example, to use the *pres*() predicate.

We take $V = \{\mathbf{0}, pc, init, halt\} \uplus V_C \uplus V_P$, where $|V_C| = |C|$ and $|V_P| = |P|$; all variables range over $\mathbb{N}$. $\mathbf{0}$ is a variable that encodes constant 0, pc is a program counter, *init* is a special initialization variable, and *halt* is assigned 1 when the machine halts. We have a dedicated variable $c \in V_C$ for each counter from C, with the same meaning, as well as a dedicated variable $l_i \in V_P$ for each instruction label i from P, encoding its constant value. W.l.o.g., we assume that the instruction labels range over $1 \ldots |P|$. In the following, for an arbitrary variable $v \in V \cup V'$, $(v = 0)$ denotes the constraint $(v - \mathbf{0} = 0)$, and $(v = 1)$ denotes the constraint $(v - \mathbf{0} = 1)$.

The initial condition is:

$$\Theta \equiv (pc' = 1) \wedge (l_1' = 1) \wedge (init' = 0) \wedge (halt' = 0) \wedge \bigwedge_{l_i \in V_P \setminus l_1} (l_i' = 0) \wedge \bigwedge_{c \in V_C} (c' = 0)$$

We have the set of transitions $T = T_{init} \uplus \{t_{start}\} \uplus T_{main}$. The transitions T_{init} of the initialization phase are as follows:

$$T_{init} = \left\{ \begin{array}{l} (init = 0) \wedge (l_{i+1}' - l_i = 1) \\ (pc = l_i) \wedge (pc' - l_i = 1) \end{array} \wedge pres(V \setminus \{pc, l_{i+1}\}) \;\middle|\; l_i \in 1 \ldots |P| - 1 \right\}$$

In this phase each variable $l_i \in V_P$ gets shifted by i with respect to $\mathbf{0}$.

The starting transition t_{start} finishes the initialization phase, and starts executing the main program:

$$t_{start} \equiv (pc = l_{|P|}) \wedge (pc' = l_1) \wedge (init' = 1) \wedge pres(V \setminus \{pc, init\}).$$

The transitions of the main program T_{main} are in one to one correspondence with the instructions of P. We encode each instruction as follows:

- `i:INC(r,j)`: the instruction is labeled with `i`, increments the register `r`, and jumps to `j`. This is encoded as

$$(init = 1) \wedge (pc - l_i = 0) \wedge (pc' - l_j = 0) \wedge (c'_r - c_r = 1) \wedge pres(V \setminus \{pc, c_r\}).$$

- `i:JZDEC(r,j,k)`: the instruction is labeled with `i`, and tests whether the register `r` equals 0. If yes, it jumps to `j`; otherwise it decrements `r` and jumps to `k`. This is encoded as

$$\begin{array}{l}(init = 1) \\ \wedge\, (pc - l_i = 0)\end{array} \wedge \left[\begin{array}{l}\big((c_r = 0) \wedge (pc' - l_j = 0) \wedge pres(V \setminus \{pc\})\big) \vee \\ \big(\neg(c_r = 0) \wedge (pc' - l_k = 0) \wedge (c_r - c'_r = 1) \wedge pres(V \setminus \{pc, c_r\})\big)\end{array}\right]$$

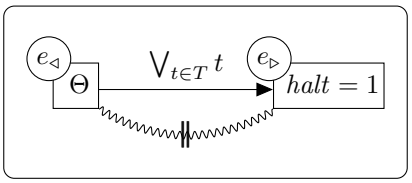
- `i:HALT`: the instruction is labeled with `i` and halts the machine. We encode it as

$$(init = 1) \wedge (pc - l_i = 0) \wedge (halt' = 1) \wedge pres(V \setminus \{halt\}).$$

This finishes the construction of the transition system S. The transition system is constructed in such a way that all transitions from T_{init} should be executed in order, followed by the execution of t_{start}, before any transition from T_{main} can be executed. By construction, machine M halts if and only if transition system S reaches a state satisfying $(halt = 1)$.

2. In the second step we encode the reachability problem of the halting state of $S = \langle V, T, \Theta\rangle$ as a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}}\rangle$, where:

- $\mathcal{E} = \{e_\triangleleft, e_\triangleright\}$;
- $\mathcal{C} = \{(e_\triangleleft, e_\triangleright)\}$;
- $\not\downarrow\ = \{(e_\triangleleft, e_\triangleright)\}$;
- $\lambda_{\mathcal{E}} = \{e_\triangleleft \to \Theta,$
  $e_\triangleright \to (halt = 1)\}$;
- $\lambda_{\mathcal{C}} = \{(e_\triangleleft, e_\triangleright) \to \bigvee_{t\in T} t\}$.



F describes all computations $s_0, \ldots, s_n$ where (s_0, s_1) satisfies Θ, s_n satisfies $(halt = 1)$, and all state pairs (s_j, s_{j+1}), $1 \le j < n$, satisfy one of the transitions from T. It is easy to see that the language of F is not empty if and only if M reaches a halting instruction. Thus, we reduced the problem of termination of M to the language emptiness of F.

□

The natural question to ask is: what use is a model for which even such a basic problem as language emptiness is undecidable? The answer is that we develop concurrent traces to describe not the complete behavior of programs, but rather some basic scenarios that allow to split the set of program behaviors into cases, and then proceed with the reasoning for each case separately. In order to do this we impose a set of natural restrictions that concurrent traces should satisfy. The first such restriction, allowing us to reduce the complexity of emptiness checking, is *transitivity*.

Definition 4.20 (**Transitive concurrent trace**)**.** We call a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ *transitive* if all causal links are labeled with transitive formulas; formally, if the following holds:

$$\forall c \in \mathcal{C},\ \forall s_0, s_1, s_2\ .\ (s_0, s_1) \models \lambda_{\mathcal{C}}(c) \wedge (s_1, s_2) \models \lambda_{\mathcal{C}}(c) \implies (s_0, s_2) \models \lambda_{\mathcal{C}}(c).$$

If a causal link is labeled with a transitive formula $\varphi(V, V')$, then the end points of any sequence of states in the scope of the link satisfy φ. Examples of transitive formulas are $\top$; $x' > x$; $x' = x$. Formulas which talk only about the present-state variables $\varphi(V, \emptyset)$, or only about the next-state variables $\varphi(\emptyset, V')$ are also transitive for any choice of φ. Another class of transitive formulas includes those which are satisfiable only over a computation consisting of a single step, such as $x = 0 \wedge x' = 1 \wedge \varphi$, for any formula φ. A conjunction of transitive formulas is always transitive as well.

It turns out that under the transitivity restriction concurrent trace emptiness becomes tractable, as the following proposition shows.

Theorem 4.21. *The emptiness problem for transitive concurrent traces is* **NP***-complete.*

Proof. First, we show that the problem is in **NP**. Fix some concurrent trace F. Remember that in Proposition 4.15 we have shown that the language of a concurrent trace is equal to the union of languages of its linearizations. Guess some linearization F' of F: this requires at most quadratic number of binary ordering choices. Let $\varphi_1, \psi_1, \varphi_2, \psi_2, \ldots, \psi_{n-1}, \varphi_n$ be a linear normal form of F'; here φ_i are event labels, and ψ_i are link labels. For each $1 \leq i \leq n-1$, we guess a) whether there are state changes in the scope of causal link ψ_i; if not, then b) whether the events φ_i and φ_{i+1} are mapped to different states in the computation or not. Note that these guesses partition the set of computations possibly accepted by F', and there is a linear number of such guesses. Now construct an SMT formula Ψ as follows:

1. initialize Ψ to $\top$, and j to 0.
2. Repeat for each $i \in [1, n-1]$:
   - set $\Psi \longleftarrow \Psi \wedge \varphi_i(V^j, V^{j+1})$
   - if a) holds, then set $\Psi \longleftarrow \Psi \wedge \psi_i(V^{j+1}, V^{j+2})$, set $j \longleftarrow j + 2$
   - if a) doesn't hold, but b) holds, then set $j \longleftarrow j + 1$
3. set $\Psi \longleftarrow \Psi \wedge \varphi_n(V^j, V^{j+1})$

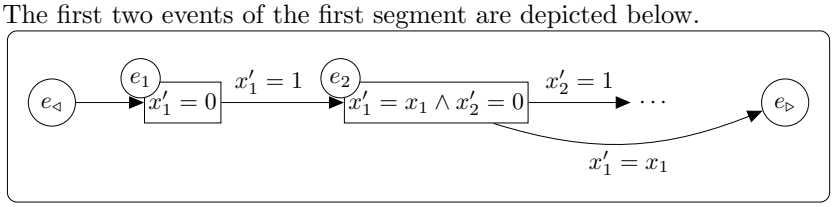
The algorithm outlined above constructs a conjunctive SMT formula in SSA form of at most linear size in n. Due to F being transitive, this formula is satisfiable if and only if there is a computation π such that F has a run on π under the guesses made. Indeed, for each causal link, say ψ_i, if a sub-computation with more than one state change, say $s_k, s_{k+1}, s_{k+2}, \ldots, s_l$ is in scope of ψ_i, then a shorter sub-computation s_k, s_l also satisfies ψ_i. Thus, if F has a run on some computation, then it has a run on a computation where at most one state change occurs within each causal link. Due to the assumption on the existence of a polynomial SMT solver, the satisfiability of Ψ can be checked in polynomial time.

Now we turn to **NP**-hardness: we prove it by a reduction from 3-SAT. Let a 3-CNF formula over the variables $x_1, \ldots, x_m$, with clauses $C_1, \ldots, C_n$ be given. We construct a linear trace F with $2 + (k+1) + 4n$ events over the variables $V = \{ x_i \mid i \in [1,m] \} \cup \{ c_i \mid i \in [1,n] \}$ such that $\mathcal{L}(F)$ is not empty if and only if the given 3-CNF formula is satisfiable. F contains 2 fixed entry ($e_\triangleleft$) and exit ($e_\triangleright$) events, and two segments between them. In the first segment the values for the variables are chosen, in the second segment the clauses are encoded.

The first segment consists of the following:

- events e_i, for each x_i, and event e_{m+1}, such that $\lambda_\mathcal{E}(e_1) \equiv (x'_1 = 0)$, $\lambda_\mathcal{E}(e_i) \equiv (x'_{i-1} = x_{i-1} \wedge x'_i = 0)$, for $1 < i \leq m$, $\lambda_\mathcal{E}(e_{m+1}) \equiv (x'_m = x_m)$;
- link $(e_\triangleleft, e_1)$ such that $\lambda_\mathcal{C}\big((e_\triangleleft, e_1)\big) \equiv \top$;
- links (e_i, e_{i+1}), for $1 \leq i \leq m$, such that $\lambda_\mathcal{C}\big((e_i, e_{i+1})\big) \equiv (x'_i = 1)$;
- links $(e_i, e_\triangleright)$, for $1 < i \leq m+1$, such that $\lambda_\mathcal{C}\big((e_i, e_\triangleright)\big) \equiv (x'_{i-1} = x_{i-1})$.

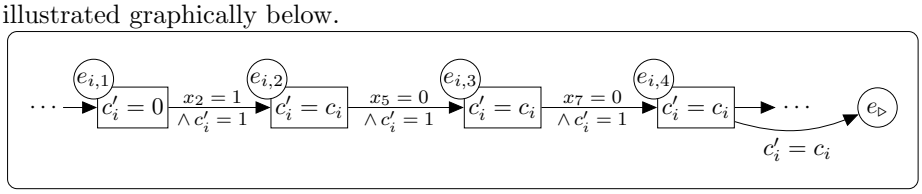
The first two events of the first segment are depicted below.



The second segment of F follows after the first, and contains the following:

- four events $e_{i,k}$, where $1 \leq k \leq 4$, for each clause C_i such that $\lambda_\mathcal{E}(e_{i,1}) \equiv (c'_i = 0)$, and $\lambda_\mathcal{E}(e_{i,k}) \equiv (c'_i = c_i)$, for $2 \leq k \leq 4$;
- link $(e_{m+1}, e_{1,1})$ such that $\lambda_\mathcal{C}\big((e_\triangleleft, e_1)\big) \equiv \top$;
- links $(e_{i,k}, e_{i,k+1})$, for each $1 \leq i \leq n$ and $1 \leq k \leq 3$, such that $\lambda_\mathcal{C}\big((e_{i,k}, e_{i,k+1})\big) \equiv (x_j = l \wedge c'_i = 1)$, where $l = 1$ if the k-th literal of C_i is x_j, and $l = 0$ if the k-th literal of C_i is $\neg x_j$;
- links $(e_{i,4}, e_{i+1,1})$, for each $1 \leq i < n$, such that $\lambda_\mathcal{C}\big((e_{i,4}, e_{i+1,1})\big) \equiv \top$;
- links $(e_{i,4}, e_\triangleright)$, for each $1 \leq i \leq n$, such that $\lambda_\mathcal{C}\big((e_{i,4}, e_\triangleright)\big) \equiv (c'_i = c_i)$.

The i-th quadruple of events for the example clause $C_i \equiv x_2 \vee \neg x_5 \vee \neg x_7$ is illustrated graphically below.



Finally, the exit event $e_\triangleright$ is labeled as follows: $\lambda_\mathcal{E}(e_\triangleright) \equiv \bigwedge_{i \in [1,n]} (c_i = 1)$. This finishes the construction of F.

Let the 3-CNF formula be satisfiable, and let $l_1, \ldots, l_m$ be the satisfying assignment. Then F has an accepting run on the computation where there is a state change within each link (e_i, e_{i+1}) if and only if $l_i = \mathit{true}$. In this computation, the first segment chooses $x_i = 1$ iff $l_i = \mathit{true}$; otherwise $x_i = 0$.

Function SSA(F)

In : concurrent trace F
Out: formula in SSA form
begin
  select $F' \in \mathit{Compactizations}(F)$
  let $N_C(F') = \varphi_1, \ldots, \varphi_n$
  set $\psi \longleftarrow \top$
  foreach $i \in [1, n]$ **do**
    set $\psi \longleftarrow \psi \wedge \varphi_i(V^{i-1}, V^i)$
  return ψ

Function Satisfiable(F)

In : concurrent trace F
Out: *true* / *false*
begin
  set $\psi \longleftarrow \mathit{SSA}(F)$
  return $\mathit{sat}(\psi)$

Function UnsatSubtrace(F)

In : concurrent trace F
Out: trace $F' \supseteq F$
begin
  set $\psi \longleftarrow \mathit{SSA}(F)$
  set $\Psi = \mathit{unsat\_core}(\psi)$
  set $F' \longleftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$
  let $\mathit{add}(e) \equiv$ **if** $e \notin \mathcal{E}'$ **then**
    set $\mathcal{E}' \longleftarrow \mathcal{E}' \cup e$
  foreach $\psi_i \in \Psi$ **do**
    if ψ_i *is from* $\lambda_{\mathcal{E}}(e_j)$ **then**
      $\mathit{add}(e_j)$
      $\lambda'_{\mathcal{E}}(e_j) \longleftarrow \psi_i$
    if ψ_i *is from* $\lambda_{\mathcal{C}}\big((e_j, e_k)\big)$ **then**
      $\mathit{add}(e_j)$; $\mathit{add}(e_k)$
      $\mathcal{C}' \longleftarrow \mathcal{C}' \cup (e_j, e_k)$
      $\lambda'_{\mathcal{C}}\big((e_j, e_k)\big) \longleftarrow \psi_i$
  set $\not\downarrow' \longleftarrow \not\downarrow \cap\ \mathcal{E}' \times \mathcal{E}'$

Figure 4.5: Utility functions for emptiness checking

After the choice is made, the values of x_i are preserved for the rest of the computation.

In the second segment there is a state change within each link $(e_{i,j}, e_{i,j+1})$ if and only if the j-th literal of clause C_i is equals *true*. Because there is at least one true literal for each clause, the value of c_i is assigned to 1 at least once, and this value is preserved for the rest of the computation. Thus, the label of event $e_{\triangleright}$ is satisfied, and F accepts the computation.

By a similar reasoning, if there is a computation which is accepted by F, then we can construct a satisfying assignment $l_1, \ldots, l_m$ for the 3-CNF formula. We take $l_i = \mathit{true}$ if and only if there is a state change within link (e_i, e_{i+1}), where x_i is assigned 1. By construction of the second segment, each variable c_i is assigned 1 if and only if the corresponding clause C_i has at least one true literal in it. This condition is checked by the label of event $e_{\triangleright}$. □

Note that the trace we used in the proof above is *linear*: thus, emptiness checking is **NP**-hard already for linear transitive traces. The essential property of linear traces (i.e., concurrent traces without any concurrency aspects) which makes their emptiness checking hard is the existence of causal links: they allow to model disjunction. In the algorithms for system analysis, which we describe in the next chapters, we use an even simpler model than linear traces, *compactization*. For a given trace F, we obtain it by first producing some *compact linearization* of F (here is where the name stems from), and then "gluing" together the events by prohibiting any state changes to occur in between. The trace obtained in such a way can be considered as an underapproximation of F: if the language of compactization of F is not empty, then the language of F is not empty as well.

Definition 4.22. For a given concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, we call its *compactization* any trace $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$ such that:

- there are traces $F_L \in \mathit{Linearizations}(F)$ and $F_{CL} \in \mathit{Contractions}(F_L)$; let $\varphi_1, \psi_1, \varphi_2, \psi_2, \ldots, \psi_{n-1}, \varphi_n$ be the linear normal form of F_{CL};
- $\mathcal{E}' = \mathcal{E}$ is an ordered set of events $\{e_1, e_2, \ldots, e_n\}$;
- $\mathcal{C}' = \{ (e_i, e_{i+1}) \mid 1 \leq i < n \}$;
- $\not\downarrow' = \{ (e_i, e_j) \mid 1 \leq i \leq n, i \neq j \}$;
- $\lambda'_{\mathcal{E}} = \{ e_i \to \varphi_i \mid 1 \leq i \leq n \}$;
- $\lambda'_{\mathcal{C}} = \{ (e_i, e_{i+1}) \to \bot \mid 1 \leq i < n \}$.

We call the sequence $\varphi_1, \varphi_2, \ldots, \varphi_n$ a *compactization normal form* of F', $N_C(F')$. We denote the set of all compactizations of F by $\mathit{Compactizations}(F)$.

A compactization of a trace F is some linear sequence of events, which cannot be "pumped": the given compactization F' of F can accept only a computation where the number of state changes equals exactly to the number of events in F'. This makes its emptiness checking trivial: the language of F' is not empty exactly when the SSA-shaped conjunction of formulas from its normal form is satisfiable. But what happens when such an SSA formula is unsatisfiable? In that case we can use the *unsatisfiable core* of the formula to pinpoint the events and links of the *original* concurrent trace which lead to the unsatisfiability.

In Figure 4.5 we outline the three utility functions, *SSA*, *Satisfiable* and *UnsatSubtrace*, which will be used in the next chapters. *SSA* builds an SSA formula from some compactization of a concurrent trace, while *Satisfiable* checks this SSA formula for satisfiability. As for *UnsatSubtrace*, note that $\mathit{SSA}(F)$ is a conjunction of formulas such that each can be easily identified as a label of either an event or a causal link from F. Indeed, the conjuncts result from applications of the rules *EventRestriction*, *LinkRestriction*, or *EventContraction*. Under this observation, the behavior of *UnsatSubtrace* is clear: given these conjuncts, it reconstructs the corresponding events, links, and conflicts of the original trace.

4.3 Complementation and Language Inclusion

We have defined the syntactic trace inclusion relation before (see Definition 4.7). Here we show that it can be used to underapproximate the language inclusion between concurrent traces.

Proposition 4.23. Structural inclusion between concurrent traces implies also their language inclusion: if $F' \subseteq F$ then $\mathcal{L}(F') \subseteq \mathcal{L}(F)$.

Proof. Let $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$ be two concurrent traces. Suppose that for some trace morphism $\mu = \langle \mu_{\mathcal{E}} : \mathcal{E} \to \mathcal{E}', \mu_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}' \rangle$ we have $F' \subseteq_{\mu} F$, and there is a computation $s_0, \ldots, s_n \in \mathcal{L}(F')$. Then, by definition 4.3 we have an injective mapping $\sigma' : \mathcal{E}' \to \{s_0, s_1, \ldots, s_n\}$, for which all conditions of the definition are satisfied. We show that for the mapping $\sigma = \sigma' \circ \mu_{\mathcal{E}} : \mathcal{E} \to \{s_0, s_1, \ldots, s_n\}$ these conditions hold as well:

1. By assumption, for each $e \in \mathcal{E}$ we have some $e' = \mu_{\mathcal{E}}(e) \in \mathcal{E}'$ and $s_i = \sigma'(e') = \sigma(e)$ such that the formula $\lambda'_{\mathcal{E}}(e')\big(s_i, s_{i+1}\big)$ holds. By the definition of trace inclusion, $\lambda'_{\mathcal{E}}(e') \Longrightarrow \lambda_{\mathcal{E}}(e)$, thus the formula $\lambda_{\mathcal{E}}(e)\big(s_i, s_{i+1}\big)$ holds as well. Moreover, we have $s_0 = \sigma'(e'_{\triangleleft})$, $s_n = \sigma'(e'_{\triangleright})$ by definition 4.3, and $e'_{\triangleleft} = \mu_{\mathcal{E}}(e_{\triangleleft})$, $e'_{\triangleright} = \mu_{\mathcal{E}}(e_{\triangleright})$ by definition 4.6. Therefore, we have that $s_0 = \sigma(e_{\triangleleft})$ and $s_n = \sigma(e_{\triangleright})$.

2. By assumption, for each $c = (e_1, e_2) \in \mathcal{C}$ we have some $c' = (e'_1, e'_2) \in \mathcal{C}'$ such that $c' = \mu_{\mathcal{C}}(c)$ and there are states $s_i = \sigma'(e'_1)$, $s_j = \sigma'(e'_2)$, for which $i \leq j$, and for all $i < k < j$ the formula $\lambda'_{\mathcal{C}}(c')\big(s_k, s_{k+1}\big)$ holds. By definition 4.6 we have $e'_1 = \mu_{\mathcal{E}}(e_1)$ and $e'_2 = \mu_{\mathcal{E}}(e_2)$, therefore $s_i = \sigma(e_1)$ and $s_j = \sigma(e_2)$. By the definition of trace inclusion, $\lambda'_{\mathcal{C}}(c') \Longrightarrow \lambda_{\mathcal{C}}(c)$, thus the formula $\lambda_{\mathcal{C}}(c)\big(s_k, s_{k+1}\big)$ holds.

3. By assumption, for each $e_1, e_2 \in \mathcal{E}$ we have some $e'_1, e'_2 \in \mathcal{E}'$ such that $e'_1 = \mu_{\mathcal{E}}(e_1)$ and $e'_2 = \mu_{\mathcal{E}}(e_2)$. Then, there are states in the computation $s_i = \sigma'(e'_1) = \sigma(e_1)$ and $s_j = \sigma'(e'_2) = \sigma(e_2)$. If $e_1 \not\frown e_2$ then $e'_1 \not\frown' e'_2$, and, by definition 4.3, $i \neq j$.

The above conditions show that $s_0, \ldots, s_n \in \mathcal{L}(F)$. □

Proposition 4.23 demonstrates that structural inclusion between concurrent traces represents a sufficient condition for the inclusion of their languages. Is it also a necessary condition? Unfortunately, the answer is negative, as the following counterexample shows.

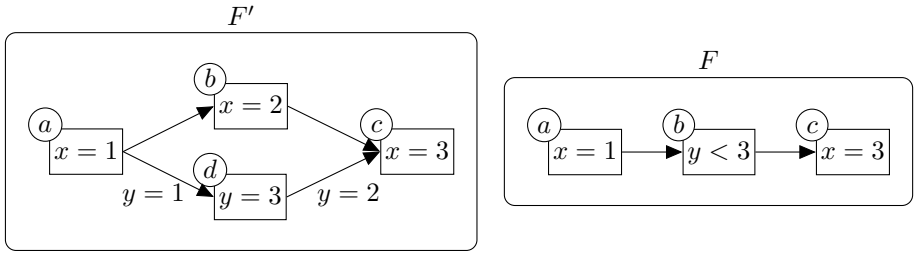


Figure 4.6: Counterexample to trace inclusion (all events are in conflict)

Example 4.24 (**Counterexample to trace inclusion**)**.** Consider the concurrent traces in Figure 4.6. All events are in conflict; we do not show conflict edges to avoid clutter. The right trace, F, accepts all computations where there are at least three events, satisfying the predicates $x = 1$, $y < 3$, $x = 3$ in that order. The language of the left trace F' is the union of languages of two possible linearizations: where event b comes before d, and vice versa. The first linearization contains at least the events $x = 1$, $x = 2 \wedge y = 1$, $y = 3$, $x = 3$; while the second contains at least the events $x = 1$, $y = 3$, $x = 2 \wedge y = 2$, $x = 3$. It can be easily seen that both linearizations belong to the language of F. Nevertheless, the structural language inclusion $F' \subseteq F$ doesn't hold, because $x = 2 \implies y < 3$ and $y = 3 \implies y < 3$; thus, no trace morphism $F \to F'$ exists.

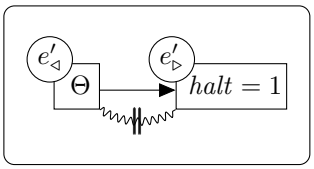
In fact, we can show that language inclusion is undecidable even for transitive concurrent traces. Remember how in the proof of Theorem 4.19 we use a simple concurrent trace encoding all computations of a Minsky machine. For that we labeled the single causal link of the trace with the disjunction of all transitions corresponding to the Minsky machine instructions; this encoding represented a universal constraint for an unbounded number of events. The transitivity restriction disallows such disjunctive labels, and reduces the complexity of emptiness checking to **NP**. For language inclusion we can use the second trace to encode the universal constraint without violating the transitivity restriction.

Theorem 4.25. *The language inclusion problem for transitive concurrent traces over any logic that includes atoms of the form $x-y=c$, for variables x, y ranging over $\mathbb{N}$, and constants $c \in [0,1]$, is undecidable.*

Proof. Let M be the Minsky machine, and $S = \langle V, T, \Theta \rangle$ – the transition system as defined in the proof of Theorem 4.19. We encode the reachability of a halting state of M as a language inclusion problem between two transitive concurrent traces F' and F: M reaches a halting state exactly when $\mathcal{L}(F') \not\subseteq \mathcal{L}(F)$. Define F' and F as follows.

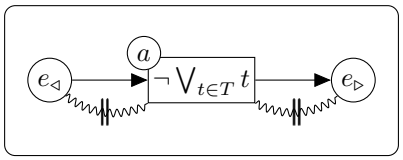
$F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, where:

- $\mathcal{E}' = \{e'_{\triangleleft}, e'_{\triangleright}\}$;
- $\mathcal{C}' = \{(e'_{\triangleleft}, e'_{\triangleright})\}$;
- $\not\downarrow' = \{(e'_{\triangleleft}, e'_{\triangleright})\}$;
- $\lambda'_{\mathcal{E}} = \{e'_{\triangleleft} \to \Theta,$ $e'_{\triangleright} \to (\mathit{halt} = 1)\}$;
- $\lambda'_{\mathcal{C}} = \{(e'_{\triangleleft}, e'_{\triangleright}) \to \top\}$.



$F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{e_{\triangleleft}, a, e_{\triangleright}\}$;
- $\mathcal{C} = \{(e_{\triangleleft}, a), (a, e_{\triangleright})\}$;
- $\not\downarrow = \{(e_{\triangleleft}, a), (a, e_{\triangleright})\}$;
- $\lambda_{\mathcal{E}} = \{e_{\triangleleft} \to \top, a \to \neg \bigvee_{t \in T} t,$ $e_{\triangleright} \to \top\}$;
- $\lambda_{\mathcal{C}} = \{(e_{\triangleleft}, a) \to \top, (a, e_{\triangleright}) \to \top\}$.



The language inclusion between F' and F is violated precisely when there is a computation that starts in a state satisfying the initial condition Θ, ends in a halting state, satisfying $(\mathit{halt} = 1)$, and where all intermediate state changes satisfy the formula $\bigvee_{t \in T} t$ (the negated label of event a); i.e., each state change is one of the transitions of S. As was observed already in the proof of Theorem 4.19, such a computation corresponds to the halting computation of the machine M. □

An easy way to recover decidability for language inclusion is to find the way to *complement* a concurrent trace. As we have seen already in the previous section, a set of concurrent traces is needed to represent the language union, which naturally arises in complementation. Therefore, we define the *complementation problem* as follows: given a concurrent trace F, find a set of concurrent traces $F_1, \ldots, F_n$ such that $\mathcal{L}(F_1) \cup \ldots \cup \mathcal{L}(F_n) = \overline{\mathcal{L}(F)}$. We call the set of trace $\mathcal{L}(F_1) \cup \ldots \cup \mathcal{L}(F_n)$ a *complementation set.*

Unfortunately, even very simple concurrent traces cannot be complemented. The reason, intuitively, is that a concurrent trace specifies a constraint of the shape $\exists^*\forall^*$: there exists a set of events such that for all events in between some condition hold. A complement of the above represents a reversed quantification prefix, namely $\forall^*\exists^*$: for all sets of events, there is some event in between that violates the condition. Such constraints cannot be represented by concurrent traces.

Before continuing with complementation, we prove the following useful characterization of concurrent traces: an analogue of pumping lemmas for regular and context-free languages.

Lemma 4.26 (**Pumping lemma for concurrent traces**). *For every concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ there is a function $f : \mathbb{N} \to \mathbb{N}$ such that for any natural $p > 0$ and any computation π of length at least $f(p)$ accepted by F, there is a formula φ such that π can be written as $\pi = \pi_1\pi_2\pi_3$, where $\pi_i = s^i_0, \ldots, s^i_{n_i}$, and the following holds:*

1. *$|\pi_1\pi_2| \leq f(p)$ and $|\pi_2| \geq p$;*
2. *$(s^2_j, s^2_{j+1}) \models \varphi$ for all $0 \leq j < n_2$, and $(s^1_{n_1}, s^2_0) \models \varphi$ and $(s^2_{n_2}, s^3_0) \models \varphi$*
3. *F accepts any "pumped" computation $\pi' = \pi_1\pi'_2\pi_3$, where π_2 is replaced by π'_2 in π, and π'_2 satisfies the above condition.*

Proof. The idea is that for any number p, in any sufficiently long computation from $\mathcal{L}(F)$ we can find a subcomputation of length at least p that "falls in scope" of some causal link of F without any events from F mapped to this subcomputation.

Let $m = |\mathcal{E}|$, and $\pi = s_0, \ldots, s_n \in \mathcal{L}(F)$ be some computation from the language of F; then there is a run $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$ of F on π. Order events of F according to their occurrence in π: $\mathcal{E} = \{e_1, \ldots, e_m\}$ such that if $\sigma(e_i) = s_{i'}$, $\sigma(e_j) = s_{j'}$, and $i' < j'$, then $i < j$. We always have that $\sigma(e_1) = s_0$ and $\sigma(e_m) = s_n$. Each event, except the last one, "controls" two states: a label of an event e_i, mapped into $s_{i'}$, is a formula over $s_{i'}$ and $s_{i'+1}$. Thus, all m events "control" at most $2(m-1)+1$ states. The remaining states all fall in scope of some causal links. The worst case happens when these states are distributed evenly between $m-1$ intervals between events. In that case each interval contains at least $p = \frac{(n+1)-2(m-1)+1}{m-1} = \frac{n}{m-1} - 2$ states. Thus, $f(p) = n + 1 = (p+2)(|\mathcal{E}| - 1) + 1$ is the desired function. In a computation of length at least $f(p)$ there will be a subcomputation $\pi_2 = s^2_0, \ldots, s^2_{n_2}$ of length at least p, where no events are mapped into states of π_2. Then take as φ a conjunction of labels of all causal links that "span" π_2, i.e. of all causal links (e_i, e_j) such that $\sigma(e_i) < s^2_0$ and $\sigma(e_j) > s^2_{n_2}$. □

Now we use the pumping lemma to prove that some concurrent traces cannot be complemented.

Proposition 4.27. There is a concurrent trace F such that the complement of its language cannot be represented by any finite union of concurrent traces: $\nexists F_1, \ldots, F_n$ such that $\mathcal{L}(F_1) \cup \ldots \cup \mathcal{L}(F_n) = \overline{\mathcal{L}(F)}$.

Proof. Take as F the trace shown in Figure 4.7; it is the same as the trace F_1 from Figure 4.3. It specifies the set of computations $s_0, \ldots, s_n$ such that

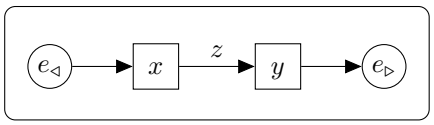


Figure 4.7: A non-complementable concurrent trace

$\exists\, s_i, s_j \,.\, i \leq j \wedge s_i \models x \wedge s_j \models y \wedge \forall\, k \in (i,j) \,.\, s_k \models z$. The complement can be represented as follows: $\big(\exists\, s_i, s_j \,.\, i \leq j \wedge s_i \models x \wedge s_j \models y\big) \implies \big(\exists\, k \in (i,j) \,.\, s_k \models \neg z\big)$.

Suppose there is a set of concurrent traces that accepts the complement of the language of F. Then, it should also accept the computation π, which is a cyclic repetition of the following sequence of five states $\pi_0 = x, z, z, \neg z, y$. Take $p = 11$, and repeat the previous sequence $f(p)$ times: $\pi = \pi_0^{f(p)}$. Then, from the pumping lemma, we have that there should be a subcomputation π_2 of length at least 11, and a formula φ such that the conditions of the pumping lemma hold. We have that π_2 should contain π_0 in exactly the described order at least once such that π_0 occurs in the middle of π_2, and that φ should be sufficiently weak to accept all pairs of states from π_0. Then the computation where the middle π_0 is replaced by $\pi_0' = x, z, z, y$ should also be accepted. But this computation belongs to the language of F; a contradiction. □

Finite automata are divided into deterministic and non-deterministic according to whether they have a unique run on any input word or not. As it was shown by Alur and Dill in case of timed automata, determinization helps to recover decidibility of the complentation and language inclusion. Here we draw a similar distinction with respect to concurrent traces. Finite automata admit a simple syntactic characterization of determinism: a given finite automaton is deterministic exactly when for all its states there is at most one transition going to another state for each input letter. In a similar way, whether a concurrent trace is deterministic can be described syntactically; but in that case the characterization is slightly more complex, and uses the notion of *event anchoring*.

Definition 4.28 (**Event anchoring**)**.** For a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, we call an event $e' \in \mathcal{E}$ *relatively left-anchored* with respect to an event e, if the following two conditions hold:

1. e' is in conflict with e: $(e, e') \in \not\downarrow$;
2. there is a causal link from e to e', and its label is unsatisfiable with the label of e: $(e, e') \in \mathcal{C} \;\wedge\; \lambda_\mathcal{C}\big((e, e')\big) \implies \neg\lambda_\mathcal{E}(e')$.

We call an event $e' \in \mathcal{E}$ *(absolutely) left-anchored* if one of the following conditions hold:

1. $e' = e_\triangleleft$, i.e. it is an entry event, or
2. e' is relatively left-anchored with respect to some absolutely left-anchored event e.

We define *relatively right-anchored* and *(absolutely) right-anchored* events in a symmetric way.

According to the above definition the set of left-anchored events is defined recursively, starting from the entry event: an event is left-anchored if we can establish that it is *relatively* left-anchored to the event that is itself established *absolutely* left-anchored. Left- or right anchoring of an event uniquely determines the position where it can be matched in any given computation; a concurrent trace where all events are anchored is deterministic.

Definition 4.29 (**Deterministic trace**)**.** We call a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ *deterministic* if every event in $\mathcal{E}$ is either left- or right-anchored.

Proposition 4.30. If a concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ is deterministic, then for any computation $\pi = s_0, \ldots, s_n$ there is at most one run $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$ of F on π.

Proof. Suppose that every event is left- or right-anchored. Fix a computation $\pi = s_0, \ldots, s_n$ and suppose there is a run $\sigma : \mathcal{E} \to \{s_0, \ldots, s_n\}$. We prove, by induction, that in any alternative run $\sigma' : \mathcal{E} \to \{s_0, \ldots, s_n\}$, any left- or right-anchored event is mapped by σ' to the same state as by σ.

Induction basis: $e_{\triangleleft}$ is always mapped to s_0. Induction step: suppose that an event e' is relatively left-anchored with respect to another event e. By induction hypothesis, we have $\sigma'(e) = \sigma(e) = s_i$ for some i. Suppose that $\sigma(e') = s_j$, $\sigma'(e') = s_{j'}$. We have that $i < j$ and $i < j'$, because $e \preceq e'$ and $e \not\downarrow e'$. Suppose that $j' < j$; we have that for all k such that $i < k < j$, the formula $\neg\lambda_{\mathcal{E}}(e')(s_k)$ holds, due to the condition of e' being relatively left-anchored. This shows that it can't be the case that $\sigma'(e') = s_{j'}$, for $j' < j$. Similarly, it's impossible that $j < j'$. The only remaining case is $j' = j$. We can prove, in a similar way, that any relatively right-anchored event $e' \in \mathcal{E}$ is mapped by any σ' to the same state as by σ. Because any event is either left- or right-anchored, there is a at most one run for any given computation.

□

The main purpose of introducing deterministic concurrent traces is to make complementation possible. If a trace is deterministic, then we can list one by one the conditions under which some computation *does not* belong to the language of the original trace. We illustrate the idea with a small example, and then formally prove its correctness.

Example 4.31 (**Complementation of a deterministic trace**)**.** Consider the concurrent trace F shown in Figure 4.8. It requires that two events satisfying a and b are present in the trace, that they are in conflict, and, moreover, all events before and after a satisfy, respectively, φ_1 and φ_2, and all events before and after b satisfy, respectively, φ_3 and φ_4. The trace is deterministic, as indicated by the boldface formulas: that is, we have $\varphi_1 \implies \neg a$, and $\varphi_4 \implies \neg b$.

The complementation process starts with the most basic trace, consisting only of entry and exit events, and gradually refines it by introducing new components. Each new component describes one particular situation, where the satisfying computation does not belong to the language of F.

The first round of refinements is related to event a. Because it is anchored with respect to $e_{\triangleleft}$, we can describe the refinements as a search for a state that satisfies a. There are several possibilities. The first possibility, represented by trace F_1, is when we have searched through the whole computation, and all states satisfy φ_1, which implies that no state satisfying a exists. An alternative

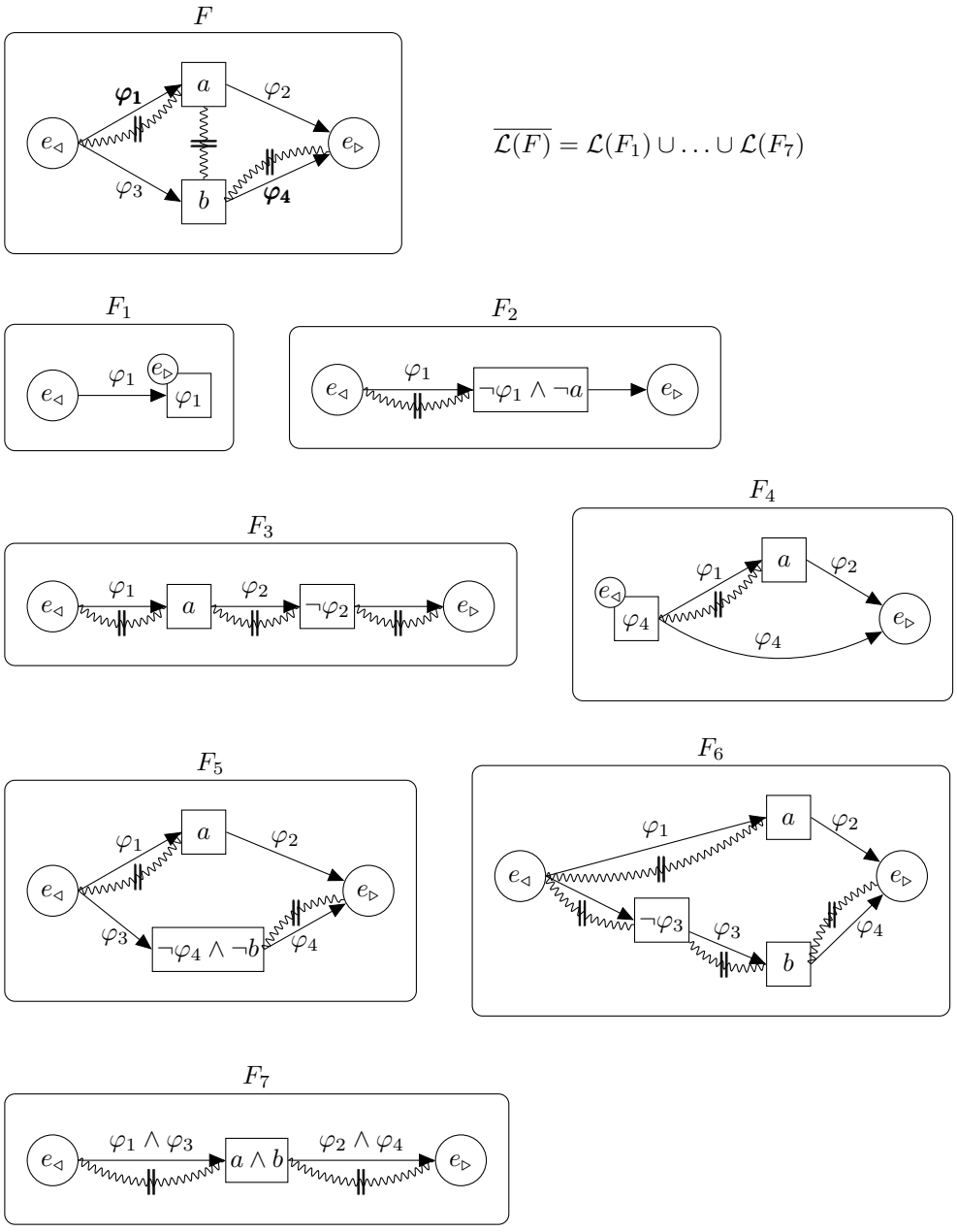


Figure 4.8: Example of a deterministic trace complementation

situation is when we have encountered a state, satisfying $\neg\varphi_1$. Then we can consider two other possibilities: either this state satisfies $\neg a$, which means that the constraint φ_1 of the causal link is violated (this is represented by trace F_2); or this state satisfies a; i.e., we have found a in the computation. After we have fixed the position for a, we can continue with the remaining components of the original trace that relate a to other, already present, events. Such a component is the causal link labeled with φ_2. With respect to it we again consider two possibilities: either it's constraint is violated, which means that a state satisfying $\neg\varphi_2$ should exist (see trace F_3); or all states in the scope of the link satisfy φ_2, and the refinement continues.

Now, we continue with the next event, labeled with b, which is still missing from the last trace. Here we repeat the same steps as for a. Trace F_4 represents the situation where all states satisfy φ_4, which implies that a state satisfying b doesn't exist. Another possibility is when we encounter a state satisfying $\neg\varphi_4$ before a state satisfying b (trace F_5). Finally, when we have fixed the position of b, we consider the missing components relating it to other events. The first component is the causal link labeled with φ_3: trace F_6 represents a situation when this constraint is violated. The last missing component is the conflict edge between a and b. Trace F_7 describes computations when the positions of a and b coincide in a computation. This is the last possibility for a computation to violate the conditions imposed by trace F.

The complementation algorithm, sketched above for an example, is described formally in Figure 4.9. Throughout the algorithm execution, two data structures are maintained: the current set S of of the traces in the complementation set, and the last refined trace F', which is always a subtrace of the original trace. The set is initialized with two traces, each one characterized by the negated label for the entry or exit events. Trace F' is initialized to contain only the appropriately labeled entry and exit events. In the main loop of the algorithm three checks are executed till saturation, which insert missing links, conflicts, and events. From the algorithm it is clear that the final complementation set is at most of linear size with respect to the size of the original trace.

Proposition 4.32. The complementation algorithm of Figure 4.9 is correct.

Proof. There are two possibilities for a computation not to belong to the language of F. The first one is when the first or the last states of the computation do not satisfy the labels of the entry or exit events. The traces, describing such situations, are contained in the initial set S. All other possibilities can be described with the following induction scheme:

Induction basis. there are events in the prefix and in the suffix of the trace, which can be mapped properly and deterministically to the computation (entry and exit events constitute the basis).

Induction step. With respect to these fixed events, there are again several possibilities:

- some of the events occur in the computation in the order, which violates some causal link of F. The first trace added to S in function *InsertLink* describes such situation.

Algorithm 2: Complement(F)

In : deterministic concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$
Out: set of concurrent traces $F_1, \ldots, F_n$ such that $\mathcal{L}(F_1) \cup \ldots \cup \mathcal{L}(F_n) = \overline{\mathcal{L}(F)}$
Data: set of concurrent traces S, last refined trace $F' = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$
`/* W.l.o.g. assume that all events are relatively left-anchored; */`
`/* relatively right-anchored events are processed in a symmetric way */`
begin
  set $S \longleftarrow \big\{ \langle \{e_\triangleleft, e_\triangleright\}, \emptyset, \{(e_\triangleleft, e_\triangleright)\}, \{e_\triangleleft \to \neg\lambda_\mathcal{E}(e_\triangleleft), e_\triangleright \to \top\}, \emptyset \rangle,$
  $\langle \{e_\triangleleft, e_\triangleright\}, \emptyset, \{(e_\triangleleft, e_\triangleright)\}, \{e_\triangleleft \to \lambda_\mathcal{E}(e_\triangleleft), e_\triangleright \to \neg\lambda_\mathcal{E}(e_\triangleright)\}, \emptyset \rangle \big\}$
  set $F' \longleftarrow \langle \{e_\triangleleft, e_\triangleright\}, \emptyset, \{(e_\triangleleft, e_\triangleright)\}, \{e_\triangleleft \to \lambda_\mathcal{E}(e_\triangleleft), e_\triangleright \to \lambda_\mathcal{E}(e_\triangleleft)\}, \emptyset \rangle$
  while $F' \neq F$ **do**
    if $\exists\, e, e' \in \mathcal{E} \cap \mathcal{E}'$ *and* $(e, e') \in \mathcal{C} \setminus \mathcal{C}'$ **then**
      set $\langle S, F' \rangle \longleftarrow$ *InsertLink*(F, F', S, e, e')
    else if $\exists\, e, e' \in \mathcal{E} \cap \mathcal{E}'$ *and* $(e, e') \in \lightning \setminus \lightning'$ **then**
      set $\langle S, F' \rangle \longleftarrow$ *InsertConflict*(F, F', S, e, e')
    else if $\exists\, e' \in \mathcal{E}' \setminus \mathcal{E}$ *such that* e' *is relatively anchored to some* $e \in \mathcal{E}'$ **then**
      set $\langle S, F' \rangle \longleftarrow$ *InsertEvent*(F, F', S, e, e')
  return S

Function InsertLink(F, F', S, e, e')

In : trace F, last refined trace F', set of traces S, events e, e'
Out: updated set of traces S, updated refined trace F'
begin
  let φ be the label of link (e, e'): $\varphi \equiv \lambda_\mathcal{C}\big((e, e')\big)$
  set $S \longleftarrow S \cup \big\{ \langle \mathcal{E}', \mathcal{C}' \cup \{(e', e)\}, \lightning' \cup \{(e', e)\}, \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \cup \{(e, e') \to \top\} \rangle,$
  $\langle \mathcal{E}' \cup \{e''\}, \mathcal{C}' \cup \{(e, e''), (e'', e')\}, \lightning' \cup \{(e, e''), (e'', e')\},$
  $\lambda'_\mathcal{E} \cup \{e'' \to \neg\varphi\}, \lambda'_\mathcal{C} \cup \{(e, e'') \to \varphi\}, (e', e) \to \top \rangle \big\}$
  set $F' \longleftarrow \langle \mathcal{E}', \mathcal{C}' \cup \{(e, e')\}, \lightning', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \cup \{(e, e') \to \varphi\} \rangle$
  return $\langle S, F' \rangle$

Function InsertConflict(F, F', S, e, e')

In : trace F, last refined trace F', set of traces S, events e, e'
Out: updated set of traces S, updated refined trace F'
begin
  let φ be the label of link (e, e'): $\varphi \equiv \lambda_\mathcal{C}\big((e, e')\big)$
  set $S \longleftarrow S \cup \big\{ \mathit{EventContraction}(F', e, e') \big\}$
  set $F' \longleftarrow \langle \mathcal{E}', \mathcal{C}', \lightning' \cup \{(e, e')\}, \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$
  return $\langle S, F' \rangle$

Function InsertEvent(F, F', S, e, e')

In : trace F, last refined trace F', set of traces S, events e, e'
Out: updated set of traces S, updated refined trace F'
begin
  let φ be the label of link (e, e'): $\varphi \equiv \lambda_\mathcal{C}\big((e, e')\big)$
  let $e'' \in \mathcal{E} \cap \mathcal{E}'$ be the smallest in the order $\preceq$ between all events $\{\, x \mid (e', x) \in \mathcal{C} \,\}$
  set $S \longleftarrow S \cup \big\{$
  $\langle \mathcal{E}', \mathcal{C}' \cup \{(e, e'')\}, \lightning', \lambda'_\mathcal{E} \cup \{e'' \to \lambda'_\mathcal{E}(e'') \wedge \varphi\}, \lambda'_\mathcal{C} \cup \{(e, e'') \to \varphi\} \rangle,$
  $\langle \mathcal{E}' \cup \{e'\}, \mathcal{C}' \cup \{(e, e')\}, \lightning' \cup \{(e, e')\}, \lambda'_\mathcal{E} \cup \{e' \to \neg\varphi \wedge \neg\lambda_\mathcal{E}(e')\}, \lambda'_\mathcal{C} \cup \{(e, e') \to \varphi\} \rangle$
  $\big\}$
  set $F' \longleftarrow \langle \mathcal{E}' \cup \{e'\}, \mathcal{C}' \cup \{(e, e')\}, \lightning' \cup \{(e, e')\},$
  $\lambda'_\mathcal{E} \cup \{e' \to \lambda_\mathcal{E}(e')\}, \lambda'_\mathcal{C} \cup \{(e, e') \to \varphi\} \rangle$
  return $\langle S, F' \rangle$

Figure 4.9: Deterministic trace complementation

- some computation state in between two of the fixed events violates the label of some causal link of F. The second trace added to S in function *InsertLink* describes such scenario.
- two events are mapped to the same computation state, although they are declared to be in conflict. Function *InsertConflict* takes care of this possibility.
- there is another event of the trace, besides the currently fixed ones, which can't be properly mapped to the computation. Function *InsertEvent* describes all possible cases how this is possible.

It is also clear that the algorithm terminates, because with each iteration of the main loop at least one new component is added to the refined trace F', and the number of the components is limited by the original trace F.

□

Notice that if the original concurrent trace is deterministic and transitive, then, by construction, all traces in the complementation set are also deterministic and transitive. This opens the way perform the language inclusion test. To check if $\mathcal{L}(F') \subseteq \mathcal{L}(F)$, for some transitive deterministic traces F and F', it is enough to do the following:

1. Compute the complementation set of F; let it be $\{F_1, \ldots, F_n\}$;
2. For each $i \in 1 \ldots n$, test $F' \cap F_i$ for emptiness. If, for some i, $\mathcal{L}(F' \cap F_i)$ is not empty, then answer the original question negatively; otherwise, answer positively.

To conclude, we now have two ways to test language inclusion for concurrent traces. If the traces are transitive and deterministic, then the precise language inclusion can be checked as outlined above. If the traces do not satisfy these conditions, then we still have the under-approximating language inclusion test via the syntactic trace inclusion relation; see Proposition 4.23. In the following chapters we use both ways to check language inclusion.

Chapter 5

Causality-based Verification: Safety

In this and the subsequent chapter we outline our solution to the verification problem described in Chapter 2. Our approach aims to answer these questions by capturing *causality*. In most general terms, causality can be defined as the relation between two events, where the first event (the *cause*) is understood to be partly responsible for the second (the *effect*). Anyone, who ever searched for an error in his or her program, should remember chains of reasoning as the following:

1. Why is variable a equal to 0 at location l_{13}, when I expect it to be 1? Because some instruction should have assigned 0 to a.
2. Which instruction is that? It should be one of $l_3 : a = x - 1$ or $l_7 : a = y$; other instructions do not change a.
3. What if l_3 has changed a? Then x should be equal to 1. How ...?
4. What if l_7 has changed a? Then y should be equal to 0. Why ...?

We can easily decompose this chain of reasoning, or, how we call it, the *causal chain*, into primitive components. In step 1 we define the effect: $(a{=}0 \wedge pc{=}13)$. In step 2 we ask what are the possible causes for the effect? They should have happened in the past, i.e., we build causal chains backwards in time. We identify instructions l_3 and l_7 as the possible causes for the effect. Then our reasoning bifurcates, or performs a *case split*: we consider each of the causes separately. We continue building causal chains, till one of them allows us to identify the ultimate cause for the observed error.

This informal example illustrates well the underlying principles of causality-based verification. As a formal foundation we employ the model of concurrent traces: it allows us to capture both dependency (though causal links), and independence (through absence of ordering). Concurrent traces describe well one particular analysis situation, like the ordering between instructions at locations l_3 and l_{13} above. For building causal chains we apply what we call *trace transformers*: primitive transformation rules which, given one concurrent trace (the premise), derive a set of possible conclusions, described also as concurrent

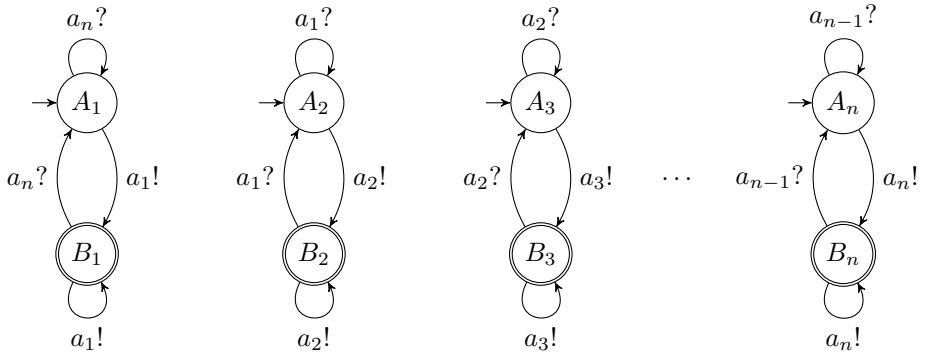


Figure 5.1: Chain of n automata with binary synchronization over $a_1, \ldots, a_n$

traces. These transformation rules capture case splits. Finally, we build composite structures on top of trace transformers, which we call *trace unwindings* or *trace tableaux*: they capture the whole proof or counterexample derivation.

The example above describes counterexample search; but how can we use trace tableaux to build proofs? The answer lies in the notion of *causal loops*. A causal loop is a circular dependency between events that are all necessary to achieve some effect. If event a requires event b to happen before, but event b, in turn, requires event a to precede it, then there is a causal loop between events a and b, and no *finite* computation containing these events can exist.

Consider the example in Figure 5.1. It consists of n automata, each with two locations A_i and B_i; the automata employ binary synchronization over the alphabet $a_1, \ldots, a_n$. The i-th automaton starts in its top location A_i; we want to check whether the system state where each automaton is in its bottom location B_i is reachable. The following causal loop proves that this state is unreachable:

- The first automaton needs to execute $a_1!$ to go from A_1 to B_1. The synchronization partner for $a_1!$ is $a_1?$; it brings the second automaton to A_2.
- Similarly, the second automaton needs to execute action $a_2!$ to go from A_2 to B_2, but this brings the third automaton to A_3.
- $\ldots$
- The last automaton needs to execute action $a_n!$ to go from A_n to B_n, but this brings the first automaton again to A_1.

The causal loop is now closed, and it constitutes the proof that system state $(B_1, \ldots, B_n)$ is not reachable from $(A_1, \ldots, A_n)$. Note that most transitions in the system are independent: automata can go forth and back between locations A_i and B_i almost unconstrained. In fact, exactly $2^n - 1$ states are reachable, and we are not aware of any automatic method that doesn't require exponential time and space to analyze this system. On the contrary, the proof with causal loops has only *linear* size, and it concisely captures human intuition about the system behavior. The presented proof is informal; we return to the example in Section 5.6, and formalize it in the form of a trace tableau.

5.1 Proofs with Trace Transformers

We generalize graph productions described in Section 3.2 to concurrent traces.

Definition 5.1 (**Trace production**). A *trace production* $p : (L \xrightarrow{r} R)$ is a trace morphism $r : L \to R$, where $L, R \in T$ are concurrent traces. The traces L and R are called the *left-hand side* and the *right-hand side* of p, respectively. A given production $p : (L \xrightarrow{r} R)$ can be applied to a trace A if there is an occurrence of L in A, i.e. a trace morphism $m : L \to A$, called a *match*. The resulting trace A' can be obtained from A by adding all elements of R with no pre-image in L, removing all elements of L with no image in R, and contracting all elements of L with the same image in R. The application of a production p to a trace A with a match m is called a *direct derivation*; we denote it with $p^m(A)$. The set of trace productions is denoted by Π.

Later we will need an additional property of trace productions which we call *context-boundedness*. All trace productions we employ are context-bounded.

Definition 5.2 (**Context-bounded trace production**). We say that a trace production $p : (L \xrightarrow{r} R)$ is *context-bounded* if all new events introduced in R are bound to occur in the scope of the context defined by L. Formally, for every event $e' \in \mathcal{E}(R)$ with no pre-image in L, $\nexists e \in \mathcal{E}(L) \,.\, e' = r(e)$, we require that:

1. there are $e_1, e_2 \in \mathcal{E}(L)$, and $e'_1, e'_2 \in \mathcal{E}(R)$, such that $e'_1 = r(e_1), e'_2 = r(e_2)$;
2. there are causal links $(e'_1, e'), (e', e'_2) \in \mathcal{C}(R)$.

For the purpose of system analysis we use combinations of trace productions; we call them *trace transformers*.

Definition 5.3 (**Trace transformer**). For a given transition system $S = \langle V, T, \Theta \rangle$, a *trace transformer* $\tau : \{\tau_1, \ldots, \tau_n\}$ is an ordered set of trace productions $\tau_i : (L \xrightarrow{r_i} R_i)$, where all productions share the same left-hand side L; we denote L by $pre(\tau)$, and call it the transformer *premise*; we call the set $post(\tau) = \{R_1, \ldots, R_n\}$ the transformer *conclusions*. We can consider a trace transformer as one proof step, describing a *case split*. Given some analysis situation (its premise), expressed as a concurrent trace, it makes a case distinction, and transforms the given trace into a set of other traces (its conclusions).

Definition 5.4 (**Sound, precise, and exact trace transformer**). We say that a trace transformer τ is *sound* if the condition below holds:

$$\forall F \in \mathcal{F} \,.\, F \subseteq_m pre(\tau) \implies \mathcal{L}(F) \cap \mathcal{L}(S) \subseteq \bigcup_{\tau_i \in \tau} \mathcal{L}\big(\tau_i^m(F)\big) \cap \mathcal{L}(S).$$

We say that a trace transformer τ is *precise* if the inverse inclusion holds:

$$\forall F \in \mathcal{F} \,.\, F \subseteq_m pre(\tau) \implies \mathcal{L}(F) \cap \mathcal{L}(S) \supseteq \bigcup_{\tau_i \in \tau} \mathcal{L}\big(\tau_i^m(F)\big) \cap \mathcal{L}(S).$$

We say that a trace transformer τ is *exact* if it is both sound and precise.

A sound trace transformer constructs over-approximations for the set of system computations; thus, it is suitable for building proofs. A precise trace

transformer, on the other hand, constructs under-approximations, and can be used to find refutations for properties, i.e., errors in programs. An exact trace transformer can be used for both purposes because it exactly transforms the set of computations of its premise into the set of runs of its conclusions. Most trace transformers we consider in this thesis are exact.

We first specialize the causality-based verification to the most basic class of *safety properties*; the historical perspective on the previous approaches to safety verification is presented in Section 2.1. We continue with the description of how concurrent traces can be used to represent violations of typical safety properties (Section 5.2), define safety-specific trace transformers (Section 5.3), and outline a sequence of increasingly more powerful proof structures as well as algorithms for their construction in Sections 5.4–5.7. Finally, in Section 5.8 we present a class of multi-threaded semaphore programs for which our causality-based verification algorithm can prove safety in polynomial time.

5.2 Representation of Safety Properties

As a first step towards the verification algorithm for safety properties, we use concurrent traces to represent their possible violations. We assume the existence of the function *Abstract*, which, given a transition system and a safety property, described, e.g., in LTL, provides a set of concurrent traces that encode all possible system computations that violate the property.

Definition 5.5 (Abstract). For a transition system $S = \langle V, T, \Theta \rangle$ and a safety property φ, the function $Abstract(S, \varphi) \in \mathcal{P}(\mathcal{F})$ gives a set of concurrent traces such that:

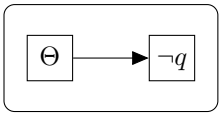
$$\mathcal{L}(S) \cap \mathcal{L}(\neg\varphi) \;=\; \bigcup_{F \in Abstract(S,\varphi)} \mathcal{L}(F)$$

Below we describe the encoding of some typical cases of safety LTL formulas; the example LTL formulas are taken from [52].

Global Invariants. Let q be a state predicate. A general safety property that q remains invariant throughout the computation is expressed by the temporal formula

$$\Box q.$$

We represent all violations of such property by the following concurrent trace:



Mutual Exclusion. Suppose a transition system S represents two processes P_1 and P_2 that require mutually exclusive access to their critical sections. In the schematic representation of Figure 5.2, each process P_i, $i = 1, 2$, consists of an endless loop. The body of the loop contains three sections: N_i ("noncritical"), T_i ("trying"), and C_i ("critical"). The property of mutual exclusion for such a program is described by the temporal formula

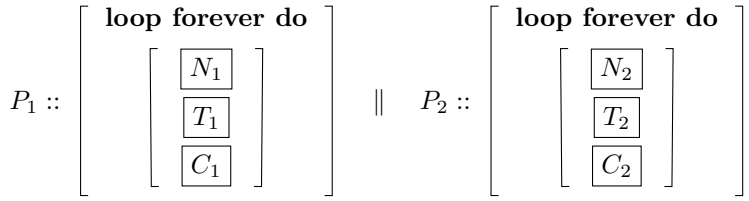
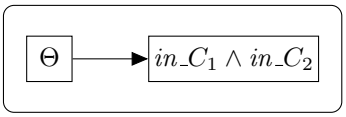


Figure 5.2: Schematic mutual exclusion program

$\Box\neg(in\_C_1 \wedge in\_C_2)$.

All violations of the formula are contained in the language of the trace



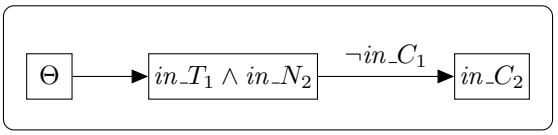
Strict Precedence. Consider again the program that provides mutually exclusive access to critical sections for processes P_1 and P_2. We may want to state strict precedence with respect to order of entries into the critical sections:

> if P_1 is in the trying section T_1 and has priority over P_2, then P_1 will precede P_2 in entering the critical section.

If we interpret P_1 having priority over P_2 as the situation in which P_1 is already in the trying section T_1, while P_2 is still in N_1, then the above requirement can be expressed by the formula

$$\Box\big(\,(in\_T_1 \wedge in\_N_2) \implies (\neg in\_C_2)\,\mathcal{W}\,in\_C_1\,\big).$$

We can represent all violations of such property by the following trace:



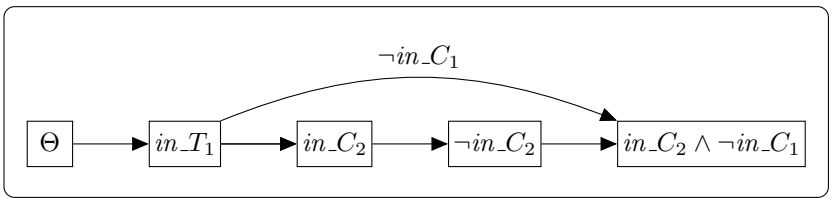
Bounded Overtaking. For the same program, we may want not to specify a strict precedence, but instead to provide upper bounds on the amount of overtaking, where overtaking means that one process enters the critical section ahead of its rival. For example, the property of 1-bounded overtaking states that

> from the time P_1 reaches T_1, P_2 may enter the critical section ahead of P_1 (overtake P_1) at most once.

This property may be expressed by the temporal formula

$$\Box\big(\,in\_T_1 \implies (\neg in\_C_2)\,\mathcal{W}(in\_C_2)\,\mathcal{W}(\neg in\_C_2)\,\mathcal{W}(in\_C_1)\,\big).$$

The violations of the property can be described by the concurrent trace

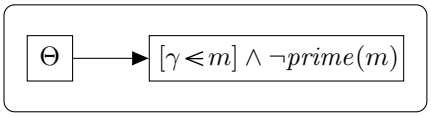


The concurrent trace above provides a clear intuition for a violation of the 1-bounded overtaking property. Indeed, it is violated exactly when process P_1 was trying to get access, i.e. it is in section T_1, but after that event there exist two accesses to the critical section C_2 by process P_2, while process P_1 is forced to stay outside of its critical section C_1.

Invariances over Communication Events. Let P be a nonterminating program that computes and prints the sequence of all prime numbers on channel γ. We can express the requirement "*nothing but primes are ever printed*" by the temporal formula

$$\Box\big(\,[\gamma{\lessdot}m] \implies prime(m)\,\big).$$

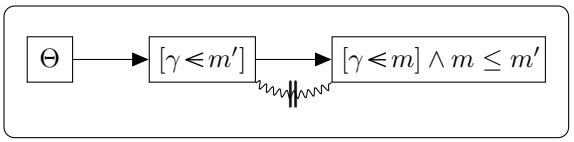
The formula states that, whenever the event $[\gamma{\lessdot}m]$ happens, a new value m that is printed to γ must be prime. We can capture all violations of this property by the trace



Monotonicity. For the same nonterminating prime-printing program, consider the property "*the printed primes form a strictly increasing sequence*". We can express this by the safety formula

$$\Box\big(\,[\gamma{\lessdot}m] \wedge \widehat{\diamondsuit}\,[\gamma{\lessdot}m']\,\big) \implies (m' < m).$$

We represent all violations of such property by the following concurrent trace:

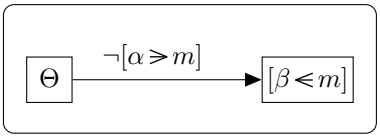


Absence of Unsolicited Response. Let S implement a buffer that has input channel α and output channel β:

The purpose of the system is to collect messages from α and eventually transmit them on β; we assume that all messages are distinct. An important property for such a system is that every message transmitted on β must have been previously received on α. This property is expressed by the formula

$$\Box\big(\,[\beta{\lessdot}m] \implies \diamondsuit[\alpha{\gtrdot}m]\,\big).$$

The violations of the property can be described by the concurrent trace



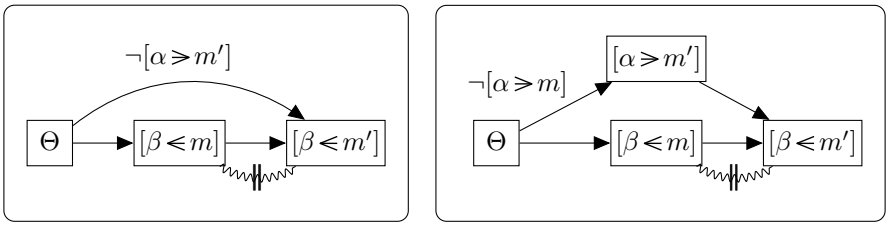
The trace represents all system computations where a message m is sent to the output channel β, and m was not received before on the input channel α.

First-In-First-Out Ordering. For the same buffer system consider the following property: messages are transmitted over β in the same order that they are transmitted over α. One possibility to express such a requirement is with the following formula:

$$\Box\Big(\,\big(\,[\beta{\lessdot}m'] \wedge \widehat{\diamondsuit}\,[\beta{\lessdot}m]\,\big) \implies \diamondsuit\big(\,[\alpha{\gtrdot}m'] \wedge \widehat{\diamondsuit}\,[\alpha{\gtrdot}m]\,\big)\,\Big).$$

This formula states that, if m' is sent on β at t'_β and m is sent on β at t_β, $t_\beta < t'_\beta$, then there exist t_α and t'_α, $t_\alpha < t'_\alpha \leq t'_\beta$, such that m' is sent on α at t'_α and m is sent on α at t_α.

The system computations that violate the above formula lie in the union of languages of two concurrent traces



The first trace describes a situation when message m' was not received on α before it was sent on β. The second trace describes another possible case: when message m' was received on α, but there was no previous event of receiving m. Notice that the first trace is a refined version of the trace that specifies violations for the absence of unsolicited response property, and can be safely omitted if that property is guaranteed.

5.3 Safety Trace Transformers

Here we describe the basic trace transformers appropriate for the analysis of safety properties. Our trace transformers are actually parametrized trace transformer *schemas*, which can be instantiated for arbitrary values of a predefined set of parameters. We write the schema parameters in parentheses after the trace transformer name.

All of the described trace transformers are precise in the sense of definition 5.4. The reason is that for any transformer (besides *ConflictSplit*) we do not remove events or links, and the labels of events and links of its conclusions

either stay the same as in the premise, or become stricter by adding new conjuncts. By combining that fact with the existence if a trace morphism from the premise to any conclusion, we have that all conditions of the definition 4.7 hold, and the language of the conclusion is included in the language of the premise. Therefore, the inclusion holds also for the union of languages of all conclusions, and the trace transformer is precise. For the *ConflictSplit* transformer we give a separate proof of its preciseness.

All of the described trace transformers are also sound. To save the space we introduce now some notation that will be used uniformly in all the soundness proofs further down. For any trace transformer τ, let $pre(\tau) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, and $F = \langle {}'\mathcal{E}, {}'\mathcal{C}, {}'\not\downarrow, {}'\lambda_{\mathcal{E}}, {}'\lambda_{\mathcal{C}} \rangle$ be a concurrent trace such that $F \subseteq_{\mu} pre(\tau)$ for the trace morphism μ. Let $\pi = s_0, s_1, s_2, \ldots \in \mathcal{L}(F)$ be some system computation from the language of F; then there are mappings ${}'\sigma : {}'\mathcal{E} \to \{s_0, s_1, \ldots, s_n\}$ and $\sigma = {}'\sigma \circ \mu_{\mathcal{E}} : \mathcal{E} \to \{s_0, s_1, \ldots, s_n\}$ satisfying the conditions of definition 4.3. In each soundness proof below we show that there exists a trace production $\tau_i \in \tau$ such that $\pi \in \mathcal{L}\big(\tau_i^{\mu}(F)\big)$. We let $\tau_i^{\mu}(F) = F'$, where $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$.

Order Split (Figure 5.3). The *OrderSplit*(a, b) trace transformer considers alternative orderings of two concurrent events a and b. Formally, we have

$pre\big(\mathit{OrderSplit}(a,b)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \top\}$;
- $\mathcal{C} = \not\downarrow = \lambda_{\mathcal{C}} = \emptyset$.

$post\big(\mathit{OrderSplit}(a,b)\big) = \{R_1, R_2\}$, where:

- $R_1 = \langle \mathcal{E}, \{(a,b)\}, \not\downarrow, \lambda_{\mathcal{E}}, \{(a,b) \to \top\} \rangle$;
- $R_2 = \langle \mathcal{E}, \{(b,a)\}, \not\downarrow, \lambda_{\mathcal{E}}, \{(b,a) \to \top\} \rangle$.

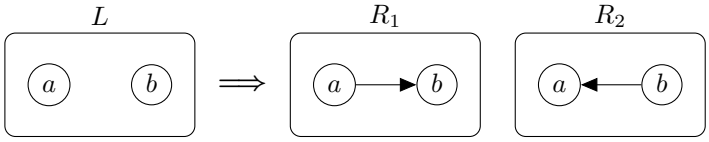


Figure 5.3: The *OrderSplit* trace transformer

Proposition 5.6. The *OrderSplit* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$ and $\sigma(b) = s_j$. If $i \leq j$, then $\pi \in \mathcal{L}\big(\tau_1^{\mu}(F)\big)$, otherwise $\pi \in \mathcal{L}\big(\tau_2^{\mu}(F)\big)$. □

Event Split (Figure 5.4). The *EventSplit*(a, φ, ψ) trace transformer, given some event a in the trace, labeled with a transition predicate ψ, and another arbitrary transition predicate φ, considers two alternatives: either a satisfies φ or not. Formally, we have

$pre\big(\mathit{EventSplit}(a,\varphi,\psi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a\}$;
- $\lambda_{\mathcal{E}} = \{a \to \psi\}$;
- $\mathcal{C} = \not\downarrow = \lambda_{\mathcal{C}} = \emptyset$.

$post\big(\mathit{EventSplit}(a, \varphi, \psi)\big) = \{R_1, R_2\}$, where:

- $R_1 = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \{a \to \psi \wedge \varphi\}, \lambda_{\mathcal{C}} \rangle$;
- $R_2 = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \{a \to \psi \wedge \neg\varphi\}, \lambda_{\mathcal{C}} \rangle$.

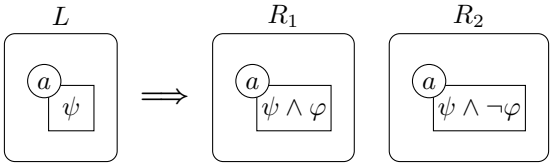


Figure 5.4: The *EventSplit* trace transformer

Proposition 5.7. The *EventSplit* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$. If the formula $\varphi(s_i, s_{i+1})$ holds, then $\pi \in \mathcal{L}\big(\tau_1^{\mu}(F)\big)$, otherwise $\pi \in \mathcal{L}\big(\tau_2^{\mu}(F)\big)$. □

Conflict Split (Figure 5.5). The $\mathit{ConflictSplit}(a, b, \varphi, \psi)$ trace transformer, given two events a and b in the trace, labeled with φ and ψ respectively, considers two possibilities: either these events coincide in time, or they are distinct, i.e., they are in conflict. Formally we have the following:

$pre\big(\mathit{ConflictSplit}(a, b, \varphi, \psi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\lambda_{\mathcal{E}} = \{a \to \varphi, b \to \psi\}$;
- $\mathcal{C} = \not\downarrow = \lambda_{\mathcal{C}} = \emptyset$.

$post\big(\mathit{ConflictSplit}(a, b, \varphi, \psi)\big) = \{R_1, R_2\}$, where:

- $R_1 = \langle \{ab\}, \mathcal{C}, \not\downarrow, \{ab \to \varphi \wedge \psi\}, \lambda_{\mathcal{C}} \rangle$;
- $R_2 = \langle \mathcal{E}, \mathcal{C}, \not\downarrow \cup \{(a, b)\}, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$.

ConflictSplit is the only rule that modifies the context in the trace where it is applied; therefore we need to specify this modification. Let *ConflictSplit* be applied to the trace $F = \langle {}'\mathcal{E}, {}'\mathcal{C}, {}'\not\downarrow, {}'\lambda_{\mathcal{E}}, {}'\lambda_{\mathcal{C}} \rangle$; we have that $F \subseteq_{\mu} L$. Let $\mathcal{E}_{a,b} = \{\mu(a), \mu(b)\}$ be the events of F, where a and b are mapped to. We denote by $\grave{\mathcal{C}} = \{\, (x, y) \in {}'\mathcal{C} \mid x \notin \mathcal{E}_{a,b}, y \in \mathcal{E}_{a,b} \,\}$ and $\acute{\mathcal{C}} = \{\, (y, x) \in {}'\mathcal{C} \mid x \notin \mathcal{E}_{a,b}, y \in \mathcal{E}_{a,b} \,\}$ the set of incoming and outgoing causal links to and from $\mathcal{E}_{a,b}$, without. Let $F' = \tau_1^{\mu}(F) = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$ be the result of applying the transformer $L \to R_1$ to F. We define F' as follows:

- $\mathcal{E}' = {}'\mathcal{E} \setminus \{\mu(a), \mu(b)\} \cup \{ab\}$;
- $\lambda'_{\mathcal{E}} = {}'\lambda_{\mathcal{E}} \setminus \{(\mu(a), \varphi), (\mu(b), \psi)\} \cup \{(ab, \varphi \wedge \psi)\}$;

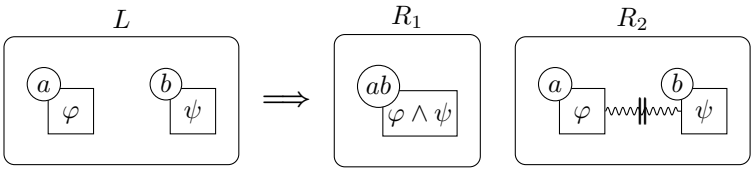


Figure 5.5: The *ConflictSplit* trace transformer

- $\mathcal{C}' = {}'\mathcal{C} \setminus \grave{\mathcal{C}} \setminus \acute{\mathcal{C}} \cup \{\ (x, ab) \mid (x, y) \in \grave{\mathcal{C}}\ \} \cup \{\ (ab, x) \mid (y, x) \in \acute{\mathcal{C}}\ \}$;
- $\lambda'_{\mathcal{C}}$.

Proposition 5.8. The *ConflictSplit* trace transformer is sound and precise.

Proof. Soundness. Let $\sigma(a) = s_i$ and $\sigma(b) = s_j$. if $i = j$ then $\pi \in \mathcal{L}\big(\tau_1^{\mu}(F)\big)$, otherwise $\pi \in \mathcal{L}\big(\tau_2^{\mu}(F)\big)$. *Preciseness.* Suppose that for some $i = 1, 2$ we have that $\pi = s_0, s_1, s_2, \ldots \in \mathcal{L}\big(\tau_i^{\mu}(F)\big)$. Then there is a mapping $\sigma : \mathcal{E}\big(\tau_i^{\mu}(F)\big) \to \{s_0, s_1, \ldots, s_n\}$. We modify σ to obtain mapping $\sigma' : \mathcal{E}(F) \to \{s_0, s_1, \ldots, s_n\}$ as follows: for $i = 1$, we remove $\big(\mu(ab), \sigma(ab)\big)$, and add $\big(\mu(a), \sigma(ab)\big)$, $\big(\mu(b), \sigma(ab)\big)$; for $i = 2$, we take $\sigma' = \sigma$. □

Conflict (Figure 5.6). The *Conflict*(a, b, φ) trace transformer, given two events a and b in the trace, labeled with the formulas φ_1, φ_2 such that *unsat*$(\varphi_1 \wedge \varphi_2)$ holds, establishes that these events are in conflict. Technically, this condition can be expressed by a single predicate $\varphi =$ *interpolate*$(\varphi_1\ ;\ \varphi_2)$, because in that case we have that $\varphi_1 \Longrightarrow \varphi$ and $\varphi_2 \Longrightarrow \neg\varphi$. Formally:

pre$\big($*Conflict*$(a, b, \varphi)\big) = \langle \mathcal{E}, \mathcal{C},\ \lightning\ , \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\lambda_{\mathcal{E}} = \{a \to \varphi, b \to \neg\varphi\}$;
- $\mathcal{C} = \lightning = \lambda_{\mathcal{C}} = \emptyset$.

post$\big($*Conflict*$(a, b, \varphi)\big) = \{\langle \mathcal{E}, \mathcal{C},\ \lightning'\ , \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle\}$, where $\lightning' = \{(a, b)\}$.

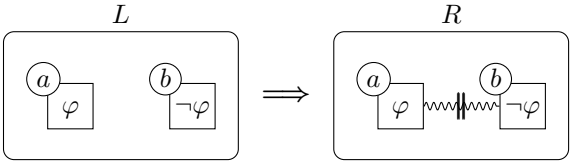


Figure 5.6: The *Conflict* trace transformer

Proposition 5.9. The *Conflict* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$ and $\sigma(b) = s_j$. It cannot be the case that both $\varphi(s_i, s_{i+1})$ and $\neg\varphi(s_i, s_{i+1})$ hold; therefore, $i \neq j$ and $\pi \in \mathcal{L}\big(R\big)$. □

Event Restriction (Figure 5.7). The *EventRestriction*(a, b, c, φ, ψ) trace transformer, given an event b, labeled with ψ, which is in the scope of a causal link (a, c), labeled with φ, allows to restrict b with φ. In a formal setting we have:

$pre\big(\mathit{EventRestriction}(a, b, c, \varphi, \psi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b, c\}$;
- $\mathcal{C} = \{(a, b), (b, c), (a, c)\}$;
- $\not\downarrow = \{(a, b), (b, c)\}$;
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \psi, c \to \top\}$;
- $\lambda_{\mathcal{C}} = \{(a, b) \to \top, (b, c) \to \top, (a, c) \to \varphi\}$.

$post\big(\mathit{EventRestriction}(a, b, c, \varphi, \psi)\big) = \{\langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda'_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle\}$, where

- $\lambda'_{\mathcal{E}} = \{a \to \top, b \to \psi \wedge \varphi, c \to \top\}$.

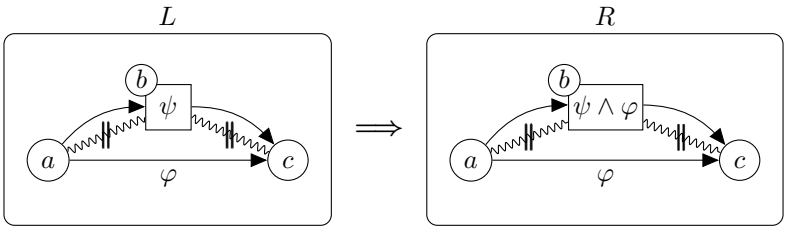


Figure 5.7: The *EventRestriction* trace transformer

Proposition 5.10. The *EventRestriction* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$, $\sigma(b) = s_j$, and $\sigma(c) = s_k$. We have that $i < j < k$, the formula $\psi(s_j, s_{j+1})$ holds, and for all $i < x < k$ the formula $\varphi(s_x, s_{x+1})$ holds. Thus, we have that the formula $(\psi \wedge \varphi)(s_j, s_{j+1})$ holds as well, and $\pi \in \mathcal{L}\big(R\big)$. □

Link Restriction (Figure 5.8). The *LinkRestriction*(a, b, c, φ, ψ) trace transformer, given a causal link (a, b), labeled with ψ, which is in the scope of another causal link (a, c), labeled with φ, allows to restrict (a, b) with φ. Formally, we have:

$pre\big(\mathit{LinkRestriction}(a, b, c, \varphi, \psi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b, c\}$;
- $\mathcal{C} = \{(a, b), (b, c), (a, c)\}$;
- $\not\downarrow = \emptyset$;
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \top, c \to \top\}$;
- $\lambda_{\mathcal{C}} = \{(a, b) \to \psi, (b, c) \to \top, (a, c) \to \varphi\}$.

$post\big(\mathit{LinkRestriction}(a, b, c, \varphi, \psi)\big) = \{\langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle\}$, where

- $\lambda'_{\mathcal{C}} = \{(a, b) \to \psi \wedge \varphi, (b, c) \to \top, (a, c) \to \varphi\}$.

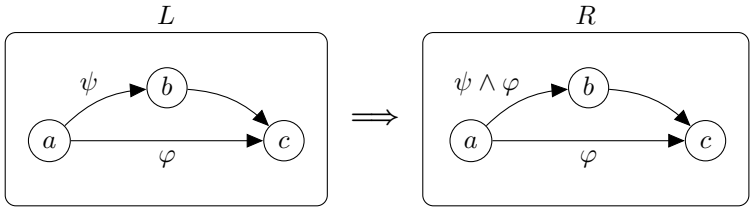


Figure 5.8: The *LinkRestriction* trace transformer

Proposition 5.11. The *LinkRestriction* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$, $\sigma(b) = s_j$, and $\sigma(c) = s_k$. We have that $i \leq j \leq k$, for all $i \leq x < j$ the formula $\psi(s_x, s_{x+1})$ holds, and for all $i \leq y < k$ the formula $\varphi(s_y, s_{y+1})$ holds. Therefore, for all $i \leq x < j$ the formula $(\psi \wedge \varphi)(s_x, s_{x+1})$ holds, and $\pi \in \mathcal{L}\big(R\big)$. □

Causal Transitivity (Figure 5.9). The *CausalTransitivity*$(a, b, c, \varphi_1, \psi, \varphi_2)$ trace transformer, given two causal links $(a, b), (b, c) \in \mathcal{C}$, labeled with φ_1 and φ_2 respectively, and an event b labeled with ψ, derives by transitivity a new causal link (a, c), labeled with $\varphi_1 \vee \psi \vee \varphi_2$.

Formally it is expressed as follows:

$pre\big(\mathit{CausalTransitivity}(a, b, c, \varphi_1, \psi, \varphi_2)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b, c\}$;
- $\mathcal{C} = \{(a, b), (b, c)\}$;
- $\not\downarrow = \emptyset$;
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \psi, c \to \top\}$;
- $\lambda_{\mathcal{C}} = \{(a, b) \to \varphi_1, (b, c) \to \varphi_2\}$.

$post\big(\mathit{CausalTransitivity}(a, b, c, \varphi_1, \psi, \varphi_2)\big) = \{\langle \mathcal{E}, \mathcal{C}', \not\downarrow, \lambda_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle\}$, where

- $\mathcal{C}' = \mathcal{C} \cup \{(a, c)\}$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \varphi_1 \vee \psi \vee \varphi_2\}$.

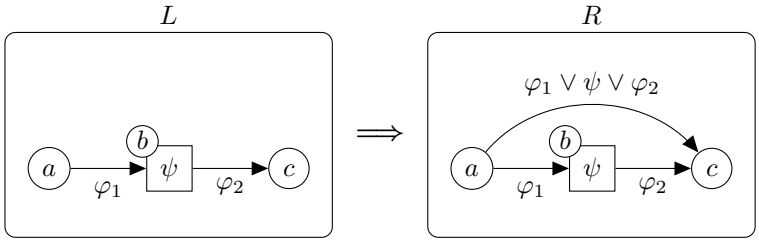


Figure 5.9: The *CausalTransitivity* trace transformer

Proposition 5.12. The *CausalTransitivity* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$, $\sigma(b) = s_j$, and $\sigma(c) = s_k$. We have that $i \leq j \leq k$, for all $i < x < j$ the formula $\varphi_1(s_x, s_{x+1})$ holds, for all $j < y < k$ the formula $\varphi_2(s_y, s_{y+1})$ holds, and the formula $\psi(s_j, s_{j+1})$ hold. Therefore for all $i < z < k$ the formula $(\varphi_1 \vee \psi \vee \varphi_2)(s_z, s_{z+1})$ holds, and $\pi \in \mathcal{L}\big(R\big)$. □

Conflict Transitivity (Figure 5.10). The *ConflictTransitivity*(a, b, c) trace transformer, given a conflict $a \not\frown b$ and causal links $(a, b), (b, c) \in \mathcal{C}$, derives by transitivity a new conflict $a \not\frown c$ (conflict propagates over causal links). Formally, we have:

$pre\big(\mathit{ConflictTransitivity}(a, b, c)\big) = \langle \mathcal{E}, \mathcal{C}, \not\frown, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b, c\}$;
- $\mathcal{C} = \{(a, b), (b, c)\}$;
- $\not\frown = \{(a, b)\}$;
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \top, c \to \top\}$;
- $\lambda_{\mathcal{C}} = \{(a, b) \to \top, (b, c) \to \top\}$.

$post\big(\mathit{ConflictTransitivity}(a, b, c)\big) = \{\langle \mathcal{E}, \mathcal{C}, \not\frown \cup \{(a, c)\}, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle\}$.

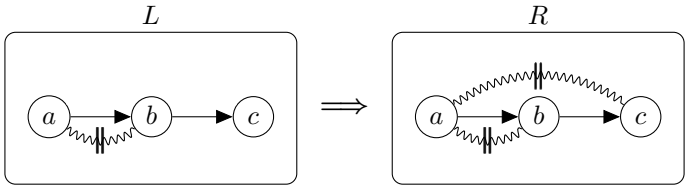


Figure 5.10: The *ConflictTransitivity* trace transformer

Proposition 5.13. The *ConflictTransitivity* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$, $\sigma(b) = s_j$, and $\sigma(c) = s_k$. We have that $i < j \leq k$; therefore, $i < k$ and $\pi \in \mathcal{L}\big(R\big)$. □

Necessary Event (Figure 5.11). The *NecessaryEvent*(a, b, φ) trace transformer, given two causally related and conflicting events a and b in a concurrent trace, and a state predicate $\varphi \in \Phi(\mathcal{V})$, such that the label of a implies φ', and the label of b implies $\neg\varphi$, introduces a new "bridging" event c in between. This condition can be interpreted as a contradiction between events a and b (a "ends" in the region φ, while b "starts" in the region $\neg\varphi$). In a formal setting we have:

$pre\big(\mathit{NecessaryEvent}(a, b, \varphi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\frown, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\mathcal{C} = \{(a, b)\}$;
- $\not\frown = \{(a, b)\}$;
- $\lambda_{\mathcal{E}} = \{a \to \varphi', b \to \neg\varphi\}$;

- $\lambda_{\mathcal{C}} = \{(a, b) \to \top\}$.

$post\big(NecessaryEvent(a, b, \varphi)\big) = \{\langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}}\rangle\}$, where

- $\mathcal{E}' = \mathcal{E} \cup \{c\}$;
- $\mathcal{C}' = \mathcal{C} \cup \{(a, c), (c, b)\}$;
- $\not\downarrow' \;=\; \not\downarrow \cup \{(a, c), (c, b)\}$;
- $\lambda'_{\mathcal{E}} = \lambda_{\mathcal{E}} \cup \{c \to \varphi \wedge \neg\varphi'\}$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \top, (c, b) \to \top\}$.

The $NecessaryEvent(a, b, \varphi)$ trace transformer can be applied to two arbitrary ordered and conflicting events a and b, in case the SSA formula built from their labels is unsatisfiable. The predicate φ is obtained by Craig interpolation between the labels of a and b as follows: if $unsat(\lambda_{\mathcal{E}}(a) \wedge \lambda_{\mathcal{E}}(b)')$, then we let $\varphi = interpolate(\lambda_{\mathcal{E}}(a)\,;\, \lambda_{\mathcal{E}}(b)')_{[\mathcal{V}'/\mathcal{V}]}$; we have $\lambda_{\mathcal{E}}(a) \Longrightarrow \varphi'$ and $\lambda_{\mathcal{E}}(b) \Longrightarrow \neg\varphi$.

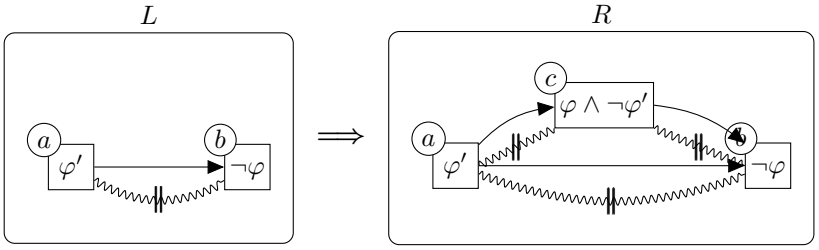


Figure 5.11: The *NecessaryEvent* trace transformer

Proposition 5.14. The *NecessaryEvent* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$ and $\sigma(b) = s_j$. We have that $i < j$, the formulas $\varphi'(s_i, s_{i+1})$ and $\neg\varphi(s_j, s_{j+1})$ hold. Because φ is a state formula, we have that the formulas $\varphi(s_{i+1})$ and $\neg\varphi(s_j)$ hold. They cannot hold for the same state, thus we have that $i + 1 < j$. We show that there is an index k such that $i + 1 \leq k < j$ and the formula $(\varphi \wedge \neg\varphi')(s_k, s_{k+1})$ holds. We proceed by well-founded induction over k. *Induction base:* $\varphi(s_k)$ holds for $k = i + 1$. *Induction step:* consider two alternatives: either $\neg\varphi(s_{k+1})$ or $\varphi(s_{k+1})$ should hold. In the first case we have found the desired k; in the second we have the base case for $k + 1$. The process cannot continue forever, because for $k = j - 1$ we have that $\neg\varphi(s_{k+1})$ holds. □

First/Last Necessary Event (Figure 5.12). The trace transformers $FirstNecessaryEvent(a, b, \varphi)$ and $LastNecessaryEvent(a, b, \varphi)$ have the same premise as $NecessaryEvent(a, b, \varphi)$, but in their conclusion they require, additionally, that the newly introduced event c is the *first* (resp. *last*) of the sequence of several such events that cross the half-space from φ to $\neg\varphi$ (see the figure on the right).

This can be achieved by marking the corresponding causal link with the predicate $\neg(\varphi \wedge \neg\varphi') = \neg\varphi \vee \varphi'$; but we can easily strengthen this requirement. Indeed, suppose we want event c to be the last in the sequence of possible

necessary events; the only events allowed by the above predicate are of the type u $(\varphi \wedge \varphi')$, v $(\neg\varphi \wedge \neg\varphi')$, or w $(\neg\varphi \wedge \varphi')$. But, if an event of type u or w happens, then an event of type c becomes necessary again, and it is not allowed: a contradiction. Thus, we can safely allow only events of type v to happen, and strengthen the above predicate to $\neg\varphi \wedge \neg\varphi'$. Similarly, if we want c to be the first necessary event, we can allow only events of type u, and strengthen the predicate to $\varphi \wedge \varphi'$. Formally, we have:

$$pre\big(\mathit{FirstNecessaryEvent}(a,b,\varphi)\big) = pre\big(\mathit{NecessaryEvent}(a,b,\varphi)\big)$$
$$post\big(\mathit{FirstNecessaryEvent}(a,b,\varphi)\big) = \{\langle \mathcal{E}', \mathcal{C}',\ \lightning',\ \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}}\rangle\}, \text{ where:}$$

- $\mathcal{E}'$, $\mathcal{C}'$, $\lightning'$, $\lambda'_{\mathcal{E}}$ are the same as in $post\big(\mathit{NecessaryEvent}(a,b,\varphi)\big)$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a,c) \to \varphi \wedge \varphi', (c,b) \to \top\}$.

$$pre\big(\mathit{LastNecessaryEvent}(a,b,\varphi)\big) = pre\big(\mathit{NecessaryEvent}(a,b,\varphi)\big)$$
$$post\big(\mathit{LastNecessaryEvent}(a,b,\varphi)\big) = \{\langle \mathcal{E}', \mathcal{C}',\ \lightning',\ \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}}\rangle\}, \text{ where}$$

- $\mathcal{E}'$, $\mathcal{C}'$, $\lightning'$, $\lambda'_{\mathcal{E}}$ are the same as in $post\big(\mathit{NecessaryEvent}(a,b,\varphi)\big)$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a,c) \to \top, (c,b) \to \neg\varphi \wedge \neg\varphi'\}$.

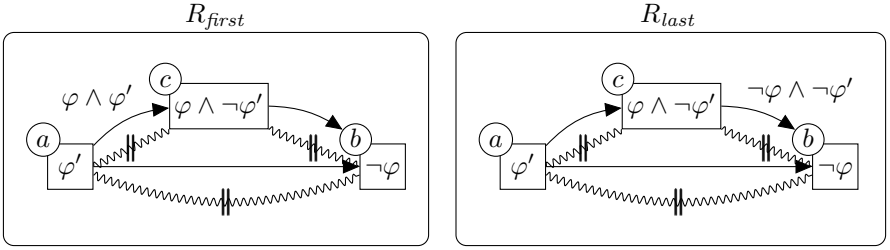


Figure 5.12: The $(\mathit{First}/\mathit{Last})\mathit{NecessaryEvent}$ trace transformers

Proposition 5.15. The $(\mathit{First}/\mathit{Last})\mathit{NecessaryEvent}$ trace transformers are sound.

Proof. For the *FirstNecessaryEvent* transformer we employ the same proof as for the *NecessaryEvent* transformer, and take as k the first index that contradicts the base case of the induction. Then, by construction, we have that for all n, $i < n < k$, the formula $(\varphi \wedge \varphi')(s_n, s_{n+1})$ holds. For the *LastNecessaryEvent* we can use the proof construction of *NecessaryEvent*, but going backwards from j. This way we will find an index k, $i+1 < k < j$, such that $(\varphi \wedge \neg\varphi')(s_k, s_{k+1})$ holds, and for all n, $k < n < j$, the formula $(\neg\varphi \wedge \neg\varphi')(s_n, s_{n+1})$ holds. □

Instantiate (Figure 5.13). The $\mathit{Instantiate}(a, \varphi, \psi)$ trace transformer, given some event a in a trace, labeled with a transition predicate φ, instantiates it with all possible system transitions that satisfy φ. An additional predicate ψ can be used to restrict the potentially large set of conclusions of this trace transformer. In that case we instantiate all transitions satisfying $\varphi \wedge \psi$, and do

not instantiate the ones satisfying $\varphi \wedge \neg\psi$. This additional restriction can be used to limit the instantiation scope, for example, to transitions of a particular process i (by using $\psi \equiv pc_i' \neq pc_i$), or to transitions that modify a particular variable x (with $\psi \equiv x' \neq x$). Formally, given a transition system $S = \langle V, T, \Theta \rangle$, this is expressed as follows:

$pre\big(\mathit{Instantiate}(a, \varphi, \psi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a\}$;
- $\lambda_{\mathcal{E}} = \{a \to \varphi\}$;
- $\mathcal{C} = \not\downarrow = \lambda_{\mathcal{C}} = \emptyset$.

$post\big(\mathit{EventSplit}(a, \varphi, \psi)\big) = \{R_0, R_1, \ldots, R_k\}$, where:

- $R_0 = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \{a \to \varphi \wedge \neg\psi\}, \lambda_{\mathcal{C}} \rangle$;
- let $\{t_1, \ldots, t_k\} = \{\, t \in T \mid sat(t \wedge \varphi \wedge \psi) \,\}$;
  then $R_i = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \{a \to t_i \wedge \varphi \wedge \psi\}, \lambda_{\mathcal{C}} \rangle$.

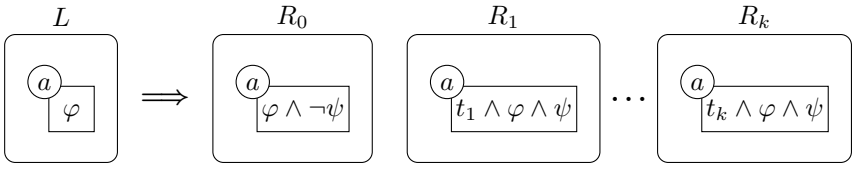


Figure 5.13: The *Instantiate* trace transformer

Proposition 5.16. The *Instantiate* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$; we have that $\varphi(s_i, s_{i+1})$ holds. If $\neg\psi(s_i, s_{i+1})$ holds, then $\pi \in \mathcal{L}\big(\tau_0^{\mu}(F)\big)$. Otherwise $\psi(s_i, s_{i+1})$ holds. Because π is a system computation, it should be the case that there exists a system transition $t_i \in T$ such that $t_i(s_i, s_{i+1})$ holds. Thus, $sat(t \wedge \varphi \wedge \psi)$ holds, and $\pi \in \mathcal{L}\big(\tau_i^{\mu}(F)\big)$. □

Forward Unrolling (Figure 5.14). Similarly to the case of *NecessaryEvent*, we have two ordered and conflicting events a and b that cannot follow immediately one after another: the label of a implies φ', and the label of b implies $\neg\varphi$. In that case the $\mathit{ForwardUnrolling}(a, b, \varphi)$ trace transformer explores all system transitions that can follow after event a. Formally, given a transition system $S = \langle V, T, \Theta \rangle$, let $\{t_1, \ldots, t_k\} = \{\, t \in T \mid sat(\varphi \wedge t) \,\}$ be the set of transitions that can follow immediately after a. Then we have:

$pre\big(\mathit{ForwardUnrolling}(a, b, \varphi)\big) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\lambda_{\mathcal{E}} = \{a \to \varphi', b \to \neg\varphi\}$;
- $\not\downarrow = \{(a, b)\}$;
- $\mathcal{C} = \{(a, b)\}$;
- $\lambda_{\mathcal{C}} = \emptyset$.

$post\big(ForwardUnrolling(a, b, \varphi)\big) = \{R_1, \ldots, R_k\}$, and:

- $R_i = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}i}, \lambda'_{\mathcal{C}} \rangle$, where:
- $\mathcal{E}' = \mathcal{E} \cup \{c\}$;
- $\mathcal{C}' = \mathcal{C} \cup \{(a, c), (c, b)\}$;
- $\not\downarrow' = \not\downarrow \cup \{(a, c), (c, b)\}$;
- $\lambda'_{\mathcal{E}i} = \lambda_{\mathcal{E}} \cup \{c \to t_i\}$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \bot, (c, b) \to \top\}$.

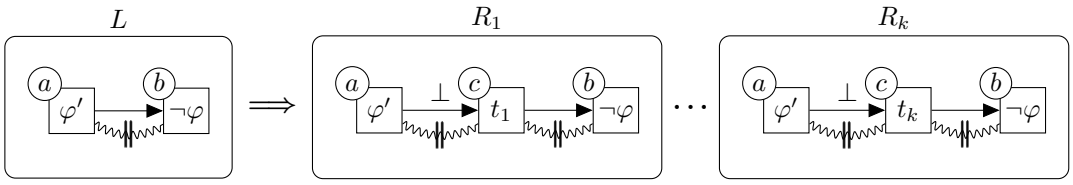


Figure 5.14: The *ForwardUnrolling* trace transformer

Proposition 5.17. The *ForwardUnrolling* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$, and $\sigma(b) = s_j$. We have that $i < j$, and the formulas $\varphi(s_{i+1}, s_{i+2})$, $\neg\varphi(s_j, s_{j+1})$ hold. It follows that $i{+}1 \neq j$ and, therefore. $i{+}1 < j$. Because π is a system run, it should be the case that there exists a system transition $t_i \in T$ such that $t_i(s_{i+1}, s_{i+2})$ holds. Thus, $sat(\varphi \wedge t_i)$ holds, and $\pi \in \mathcal{L}\big(\tau_i^{\mu}(F)\big)$. □

Backward Unrolling (Figure 5.15). In the same manner as *ForwardUnrolling*, the $BackwardUnrolling(a, b, \varphi)$ trace transformer explores all system transitions that can precede event b. Formally, given a transition system $S = \langle V, T, \Theta \rangle$, let $\{t_1, \ldots, t_k\} = \{\, t \in T \mid sat(t \wedge \neg\varphi') \,\}$ be the set of transitions that can precede b. Then we have:

$pre\big(BackwardUnrolling(a, b, \varphi)\big) = pre\big(ForwardUnrolling(a, b, \varphi)\big)$.

$post\big(BackwardUnrolling(a, b, \varphi)\big) = \{R_1, \ldots, R_k\}$, and:

- $R_i = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}i}, \lambda'_{\mathcal{C}} \rangle$, where:
- $\mathcal{E}' = \mathcal{E} \cup \{c\}$;
- $\mathcal{C}' = \mathcal{C} \cup \{(a, c), (c, b)\}$;
- $\not\downarrow' = \not\downarrow \cup \{(a, c), (c, b)\}$;
- $\lambda'_{\mathcal{E}i} = \lambda_{\mathcal{E}} \cup \{c \to t_i\}$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \top, (c, b) \to \bot\}$.

Proposition 5.18. The *BackwardUnrolling* trace transformer is sound.

Proof. Along the same lines as for the *ForwardUnrolling*. □

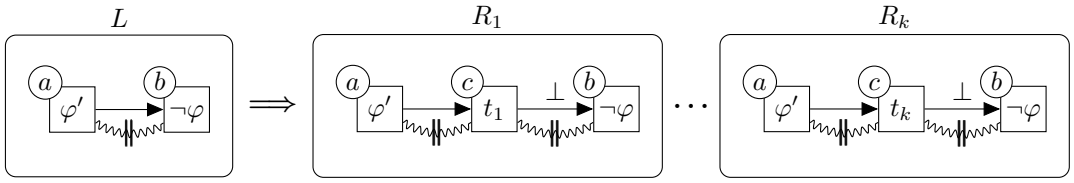


Figure 5.15: The *BackwardUnrolling* trace transformer

5.4 Trace Unwinding

Trace transformers, as described above, lay the foundation for the construction of proofs for concurrent programs. While they are the basic building blocks of such proofs, we are interested in the algorithms that construct proofs incrementally by composing the primitive trace transformers into larger structures. The most basic such structure, suitable for constructing composite trace transformers, is called *trace unwinding*.

Definition 5.19 (**Trace unwinding**)**.** For a transition system $S = \langle V, T, \Theta \rangle$, we define a *trace unwinding* as a tuple $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$, where:

- N is a set of unwinding *nodes*;
- $E \subset N \times N$ is a set of unwinding *edges*. We require that $\langle N, E \rangle$ is a directed forest, and partition the forest nodes into *internal nodes* and *leaves*: $N = N_I \uplus N_L$, where $N_I = \{n \in N \mid \exists\, (n, n') \in E\}$, $N_L = \{n \in N \mid \nexists\, (n, n') \in E\}$. Among nodes we also distinguish *roots*: $N_R = \{n \in N \mid \nexists\, (n', n) \in E\}$;
- $\gamma : N \to \mathcal{F}$ is a labeling of nodes with concurrent traces;
- $\delta : E \to \Pi$ is a labeling of edges with trace productions. We require that for all edges with the same source n, the labeling productions have the same left-hand side. Thus, we have an induced labeling of internal nodes $n \in N_I$ with trace transformers: $\delta(n) = \{\, \delta\big((n, n')\big) \mid (n, n') \in E \,\}$;
- μ is a labeling of internal nodes with trace morphisms: for all $n \in N_I$ we have $\mu(n) : \mathit{pre}\big(\delta(n)\big) \to \gamma(n)$.

A trace unwinding is a forest, which can be seen as an unrolling of the system causality relation from some set of initial nodes. If we take as initial a set of nodes labeled with the traces from $\mathit{Abstract}(S, \varphi)$, then the unwinding represents the computations of S that violate the property φ. By applying trace transformers we perform case splits; the label $\gamma(n)$ of a node n represents possible violating computations for one specific case.

Definition 5.20 (**Properties of trace unwinding**)**.** For a given transition system S, we call a trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ *correct* if for all internal nodes $n \in N_I$ the following conditions hold:

1. $\gamma(n) \subseteq_{\mu(n)} \mathit{pre}\big(\delta(n)\big)$: the trace transformer $\delta(n)$ can be applied, under the trace morphism $\mu(n)$, to the concurrent trace $\gamma(n)$ that labels node n;

2. for all $(n, n') \in E$ it holds that $\delta\big((n, n')\big)^{\mu(n)}\big(\gamma(n)\big) = \gamma(n')$, i.e. the trace production of each edge (n, n') transforms trace $\gamma(n)$ into trace $\gamma(n')$.

For a given transition system S, we call a trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu\rangle$ *sound* (*precise*, *exact*) if it is correct and, additionally, for all internal nodes $n \in N_I$ the trace transformer $\delta(n)$ is sound (precise, exact).

For a given transition system S and a safety property φ, we say that a trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu\rangle$ is *sound* (*precise*, *exact*) *for* φ, if it is sound (precise, exact), and for all traces $A \in \mathit{Abstract}(S, \varphi)$ there is a node $n \in N$ such that $A \subseteq \gamma(n)$ ($A \supseteq \gamma(n)$, $A = \gamma(n)$).

Finally, we say that a trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu\rangle$ is *complete* if for all its leaves $n \in N_L$ their labels $\gamma(n)$ are contradictory.

By applying either sound or precise trace transformers we can construct different algorithms that search for a proof or for a counterexample. When a trace unwinding is *sound*, the full exploration of the set of consequences is guaranteed, thus preserving the set of all system computations that possibly violate the property. Indeed, we have:

$$\mathcal{L}\big(\gamma(n)\big) \subseteq \bigcup_{(n,n')\in E} \mathcal{L}\Big(\delta\big((n, n')\big)^{\mu(n)}\big(\gamma(n)\big)\Big) = \bigcup_{(n,n')\in E} \mathcal{L}\big(\gamma(n')\big).$$

When we find out that each leaf of the unwinding is contradictory, in other words the unwinding is *complete*, we can be sure that no violating computations exist for the given property.

On the other hand, when an unwinding is *precise*, we have:

$$\mathcal{L}\big(\gamma(n)\big) \supseteq \bigcup_{(n,n')\in E} \mathcal{L}\Big(\delta\big((n, n')\big)^{\mu(n)}\big(\gamma(n)\big)\Big) = \bigcup_{(n,n')\in E} \mathcal{L}\big(\gamma(n')\big).$$

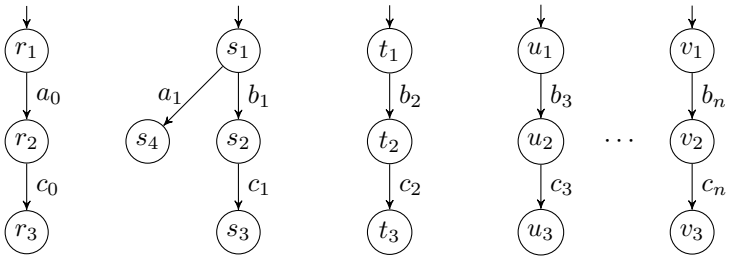
Thus, when we find a satisfiable trace at one of the unwinding leaves, we can be sure that the language of one of the unwinding roots is not empty, and a violating computation does exist.

Finally, an *exact* trace unwinding can be used both for constructing the proof and for the counterexample search.

Consider the example synchronized system shown in Figure 5.16: it was used by Esparza and Heljanko in [28] to illustrate the exponential succinctness of Petri net unfoldings. There are $n+1$ processes, and we want to check whether the global transition $\mathbf{c}$ is executable. Note that the state space of this system is exponential with respect to n: the system contains $3 \cdot 2^{n-1}$ reachable states. Thus, approaches based on state space exploration will suffer from the state space explosion problem. The authors of [28] show that the Petri net unfolding of the example system contains $2 \cdot n + 3$ places, i.e., a *linear size* unfolding can represent succinctly the exponential state space. We use the same example to demonstrate that the trace unwinding of the example system never exceeds $n+6$ nodes, but a *constant size* unwinding of just 7 nodes also suffice.

In Figure 5.17 we show the constant size trace unwinding for the example system. We first explain the unwinding construction informally, and then proceed to the formal definition of the algorithm.

Node 1, the root of the unwinding, captures all system traces where $\mathbf{c}$ is executed. One of it's preconditions is that the first process should be at location r_2;



$\mathbb{T} = \{\mathbf{a} = \{a_0, a_1\}, \mathbf{b_1} = \{b_1\}, \ldots, \mathbf{b_n} = \{b_n\}, \mathbf{c} = \{c_0, c_1, c_2, c_3, \ldots, c_n\}\}$
$\Theta \equiv r_1' \wedge s_1' \wedge t_1' \wedge u_1' \wedge \ldots \wedge v_1'$

Figure 5.16: Example system with explicit synchronization

but the initial condition says that the system is at location r_1: a contradiction. Thus, a transition that goes from r_1 to r_2 is *necessary*, and we insert the only such transition, $\mathbf{a}$, into the trace of node 2. Formally, this is done by applying the trace transformer *LastNecessaryEvent*, this is where the link label $r_2 \wedge r_2'$ comes from: it enforces to select the last transition that goes into location r_2. By a similar reasoning we conclude that transition $\mathbf{b_1}$ is also necessary, and include it into the trace of node 3. Notice that these events occur concurrently, i.e. no specific order between them is specified till now.

But in the next iteration, when we try to put them in some linear order, we find out that these transitions contradict each other: $\mathbf{a}$ goes to location s_4, $\mathbf{b_1}$ goes to location s_2, but both can start only at location s_1. Therefore, we perform a case split between two possible linearizations using the transformer *OrderSplit*, and obtain the traces of nodes 4 and 5. In node 4 the contradiction between now linearly ordered transitions $\mathbf{a}$ and $\mathbf{b_1}$ is enforced by the trace, and we again apply the transformer *LastNecessaryEvent*. It inserts an event between transitions $\mathbf{a}$ and $\mathbf{b_1}$ which needs to change the location from s_4 (the postcondition of $\mathbf{a}$) to s_1 (the precondition of $\mathbf{b_1}$). Formally, this requirement is captured by the Craig interpolant between between the post- and pre-conditions of $\mathbf{a}$ and $\mathbf{b_1}$, respectively, which happens to be $\neg s_1$. As there is no transition that goes from a location different from s_1 to s_1, this trace is declared as contradictory, and the left branch of the unwinding is closed. For the right branch of the unwinding, consisting of nodes 5 and 7, we proceed in the same way, and also close it as contradictory. Thus, the unwinding is complete, and we conclude that transition $\mathbf{c}$ is not executable.

The unwinding of Figure 5.17 has constant size, independent on the number of processes in the example system. In the worst case it could have linear size: for that to happen, the other necessary transitions $\mathbf{b_2}$ through $\mathbf{b_n}$ would have to be introduced into concurrent before transitions $\mathbf{a}$ and $\mathbf{b_1}$ are introduced. But note that these transitions do not form any contradictions both with each other as well as with $\mathbf{a}$ and $\mathbf{b_1}$; therefore, they would stay concurrent in any trace that involves them, and no ordering splits would be necessary. In fact, a

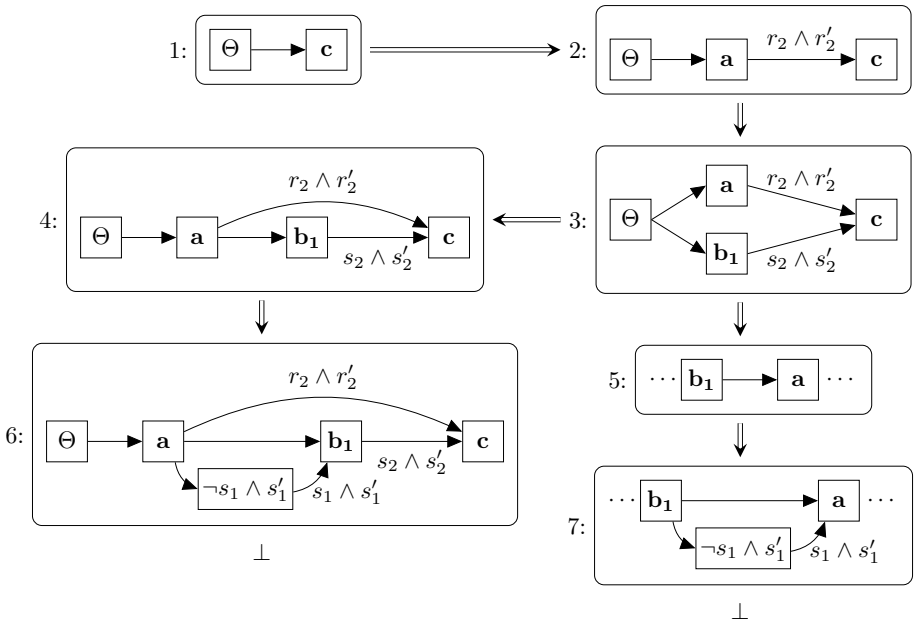


Figure 5.17: Trace unwinding for the example system of Figure 5.16

simple heuristic is able to select transitions **a** and $\mathbf{b_1}$ first, and it will always produce the trace unwinding of constant size.

Algorithm 3 in Figure 5.18 formalizes the above reasoning for an arbitrary transition system S and a safety property φ. The algorithm starts by constructing the set of roots labeled with concurrent traces from the set $\mathit{Abstract}(S, \varphi)$. It maintains a queue Q of unexplored unwinding leaves, initialized with the set of unwinding roots. At each iteration of the algorithm's main loop we select some node n from Q, and check whether the trace $\gamma(n)$ can be concretized. Function *Satisfiable* builds a compactization of $\gamma(n)$, and checks it for emptiness as described in Section 4.2. If the language of the compactization is not empty, then we report a counterexample.

The unsatisfiable subtrace is analyzed in function *SafetyRefinement*. The purpose of the function is to select the trace transformer that will be applied to the trace under consideration. This function can be instantiated in different ways yielding algorithms with different properties. Here we show one possible instantiation of *SafetyRefinement*, which is generic and applicable for both proof and counterexample search as it uses only exact trace transformers.

First, function *UnsatSubtrace* extracts from the trace an unsatisfiable subtrace, comprising only a subset of trace events, links, and conflicts. Then *SafetyRefinement* checks whether there are two events in the unsatisfiable subtrace which are not in conflict. If there are, it means that they were contracted in the compactization, but there is a possibility that they are actually in conflict. Thus, we should consider both cases, which is implemented by applying the *ConflictSplit* trace transformer. Second, it checks whether all events of the unsatisfiable subtrace are ordered. If not, then the *OrderSplit* trace transformer is applied, which considers both possible orderings between two unordered events.

Algorithm 3: Exploration of Trace Unwinding

Input : transition system $S = \langle V, T, \Theta \rangle$, safety property φ
Output: ***property holds*** / ***counterexample***
Data: trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$, queue $Q \subseteq N_L$ of unexplored nodes, trace transformer τ, trace morphism m

begin
  set $\Upsilon \longleftarrow \mathit{InitialAbstraction}(S, \varphi)$
  set $Q \longleftarrow N$
  while Q *not empty* **do**
    select some n from Q
    if $\mathit{Satisfiable}\big(\gamma(n)\big)$ **then**
      return ***counterexample*** $\gamma(n)$
    else
      set $\langle \tau, m \rangle \longleftarrow \mathit{SafetyRefinement}\big(\gamma(n)\big)$
      $\mathit{Apply}\big(\Upsilon, \tau, m, n\big)$
    set $Q \longleftarrow Q \cup \{\, n' \mid (n, n') \in E \,\} \setminus \{n\}$
  return ***property holds***

Function SafetyRefinement(F)

In : concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$
Out: transformer τ, morphism $m : \mathit{pre}(\tau) \to F$

begin
  set $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle \longleftarrow \mathit{UnsatSubtrace}\big(F\big)$
  if $\exists e_1, e_2 \in \mathcal{E}' \,.\, (e_1, e_2) \notin \not\downarrow$ **then return** $\mathit{ConflictSplit}(e_1, e_2)$
  else if $\exists e_1, e_2 \in \mathcal{E}' \,.\, (e_1, e_2) \notin \mathcal{C}^*$ **then return** $\mathit{OrderSplit}(e_1, e_2)$
  else switch $\mathcal{E}'$ **do**
    case $\{e_1\}$ **return** $\mathit{Contradiction}(e_1)$
    case $\{e_1, e_2\}$
      $\phi \longleftarrow \mathit{interpolate}\big(\lambda_{\mathcal{E}}(e_1)\,;\, \lambda_{\mathcal{E}}(e_2)'\big)$
      return $\mathit{Instantiate}(c) \circ \mathit{LastNecessaryEvent}(a \to e_1, b \to e_2, \phi_{[V'/V]})$
    otherwise $\{e_1, e_2, \ldots, e_k\}$
      $\varphi \longleftarrow \mathit{interpolate}\big(\lambda_{\mathcal{E}}(e_1) \wedge \lambda_{\mathcal{E}}(e_2)' \wedge \ldots \wedge \lambda_{\mathcal{E}}(e_{k-1})^{k-2}\,;\, \lambda_{\mathcal{E}}(e_k)^{k-1}\big)$
      return $\mathit{EventSplit}(a \to e_{k-1}, \varphi_{[V^{k-1}/V']})$

Function InitialAbstraction(S, φ)

In : system S, property φ
Out: unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$

begin
  set all of $\{N, E, \gamma, \delta, \mu\} \longleftarrow \emptyset$
  foreach $F \in \mathit{Abstract}(S, \varphi)$ **do**
    set $N \longleftarrow N \cup n$,
      where n is a new node
    set $\gamma(n) \longleftarrow F$

Procedure Apply(Υ, τ, m, n)

In: unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$,
  transformer τ, morphism m, node n

begin
  set $\mu(n) \longleftarrow m$
  foreach $\tau_i \in \tau$ **do**
    set $N \longleftarrow N \cup n'$,
      where n' is a new node
    set $E \longleftarrow E \cup (n, n')$
    set $\gamma(n') \longleftarrow \tau_i^m\big(\gamma(n)\big)$
    set $\delta\big((n, n')\big) \longleftarrow \tau_i$

Figure 5.18: Exploration of trace unwinding

If all events are ordered, we consider different cases with respect to the cardinality of the set of events in the subtrace.

If there is only one event, the trace is surely contradictory; the *Contradiction* transformer doesn't produce any new traces. If there are two events, we apply the *LastNecessaryEvent* transformer, which tries to repair the conflict by introducing a new event in the middle. For that purpose we compute the Craig interpolant between contradictory events. Finally, if there are more than two events in the subtrace, we shorten it by splitting the last but one event with the Craig interpolant between the last event and all the others; this is done by the *EventSplit* trace transformer. Note that we have the freedom to split any event in the unsatisfiable subtrace with the properly computed Craig interpolant, namely between the subtrace up to and including this event, and the rest; different choices can be appropriate depending on the context.

Theorem 5.21 (**Soundness of trace unwinding exploration**)**.** *Let a transition system S and a safety property φ be given. For any instantiation of the function SafetyRefinement the following holds. If Algorithm 3 terminates and returns* ***property holds****, and only sound trace transformers are applied in SafetyRefinement, then all system computations of S satisfy φ. If Algorithm 3 terminates and returns* ***counterexample****, and only precise trace transformers are applied in SafetyRefinement, then there exists a computation of S that violates φ.*

Proof. Assume that Algorithm 3 terminates and returns ***property holds***. In that case the queue Q is empty, and all leaves of Υ are contradictory. Any violating system computation is contained in the language of one of the roots. Due to the soundness criteria of definition 5.20, the language of the roots is contained in the union of the languages of all the leaves. But, as the latter one is empty, the former one is empty as well, and no violating system computation exists.

Assume that Algorithm 3 terminates and returns ***counterexample***. Then there is a node n that is labeled with a satisfiable concurrent trace $\gamma(n)$. Also, because the trace unwinding is precise, there is a chain of precise trace transformers from some root node in the unwinding forest to node n. Due to the inverse language inclusion along each trace transformer, the language of such a root node is not empty, and a violating system computation exists. □

Our main purpose in outlining the Algorithm 3 is its conceptual simplicity: the only thing it does is unwinding of the system causality relation by applying appropriate trace transformers. If the transition system is, actually, correct, the above algorithm will not terminate in most cases: it will continue constructing larger and larger traces. There are, however, two cases where this simple-minded approach can be adequate.

The first case is program falsification, or, in other words, bug finding. If the system does contain an error, i.e. a violating system computation exists, Algorithm 3 will eventually find it, provided that a suitable exploration strategy is used. One possible choice for a strategy is to explore the causality relation in the breadth-first manner by always selecting an unwinding leaf that is marked with the smallest concurrent trace among others. In that way, the shortest violating computation will be eventually found.

Another interesting case is *acyclic transition systems*. We call a transition system *acyclic* if no system transition can appear more than a predefined finite number of times in any system computation. Such systems comprise syntactically acyclic programs, but are not necessarily limited to them. Individual components of a synchronized concurrent program can still contain cycles; but if, whenever a cycle in a concurrent component is executed, the corresponding transition synchronizes with a different partner, the global transition system will still be acyclic. Acyclic transition systems arise naturally as models of input-driven applications, where the input sequence is finite.

Theorem 5.22 (**Completeness of trace unwinding exploration**). *Algorithm 3 is complete for* ***acyclic*** *transition systems, i.e. it always terminates for such systems.*

Proof. The trace unwinding constructed by Algorithm 3 has a finite branching degree: indeed, all transformers except *Instantiate* introduce a fixed number of branches, and the *Instantiate* transformer introduces the number of branches that is bounded by the number of system transitions. Suppose that Algorithm 3 doesn't terminate. Then, by König's lemma, the unwinding should have an infinite branch. Consider a sequence of trace transformers applied along this branch. We show that the combination of *Instantiate* and *LastNecessaryEvent* transformers should be applied infinitely often along this branch.

Suppose, this is not the case. Then, after some node n, having k events, only the other trace transformers are applied. Notice, that they do not introduce new events into the concurrent trace. *Contradiction* can be applied only once as it doesn't produce any children. We can apply neither *ConflictSplit* nor *OrderSplit* infinitely often: the number of their applications is limited by k^2, when all events are in conflict, resp. ordered. The only remaining case is to apply *EventSplit* infinitely often. This can't be the case either, due to the fact that φ is an interpolant between the first $k-1$ events and the last one. The first child, where event e_{k-1} is labeled with φ, will have an unsatisfiable subtrace consisting of events e_{k-1} and e_k only. The second child, where event e_{k-1} is labeled with $\neg\varphi$, will have an unsatisfiable subtrace comprising only the events $e_1, \ldots, e_{k-1}$. In both cases the size of the unsatisfiable subtrace decreases, and this process cannot take more than k steps.

Thus, *Instantiate* and *LastNecessaryEvent* transformers are applied infinitely often along the infinite branch. Each application of *Instantiate* introduces an event labeled with some system transition. By pigeonhole principle, there should be a system transition that occurs infinitely often in a system computation, but this cannot happen for acyclic transition systems. □

There are at least two complementary views on completeness. The above theorem belongs to the class that we would call *algorithmic completeness*: it specifies a class of programs for which some algorithm is guaranteed to terminate and give a conclusive answer. Another view is *logical completeness*: it concerns more the proof system than the actual algorithm to build a proof, and asks whether the proof system is powerful enough to represent a proof for any program which is known to be correct. The completeness proofs in the later category are usually *relative* to some assumptions; e.g., *relative to first-order reasoning* (for a nice example of such a completeness proof see, e.g., [53],

p.220). In the logical view, trace unwinding is already complete for arbitrary programs.

Theorem 5.23 (**Relative completeness of trace unwinding**). *If a transition system S satisfies a safety property φ, then there exists a sound and complete trace unwinding for S and φ.*

Proof. Let, according to Definition 5.5, the set of violating computations be represented by the set $\mathit{Abstract}(S, \varphi)$, and assume that S satisfies φ. We assume that the construction of [42] is applied, and safety checking is reduced to reachability. Then all traces in $\mathit{Abstract}(S, \varphi)$ consist of two events: the first is labeled with the initial condition Θ, and the second with one of the reachability goals.

Similar to [53], we use as given the assertion Acc that characterizes the set of reachable (accessible) system states. Now, consider an arbitrary node n; let its first event be n_1 (labeled with Θ), and its second event be n_2 (labeled with some goal g). We apply the *EventSplit* trace transformer to n_2, and split it with the predicate Acc. We get two traces, where in the first one n_2 is labeled with $(g \wedge \mathit{Acc})$, and in the second one with $(g \wedge \neg \mathit{Acc})$. The first one is closed as contradictory, because the formula $(g \wedge \mathit{Acc})$ is unsatisfiable (S is safe, therefore g implies $\neg \mathit{Acc}$). The second one is closed as contradictory, because we know that no state satisfying $\neg \mathit{Acc}$ can be reached. We apply this construction to every node: all branches are closed as contradictory, and we obtain a sound and complete trace unwinding for S. □

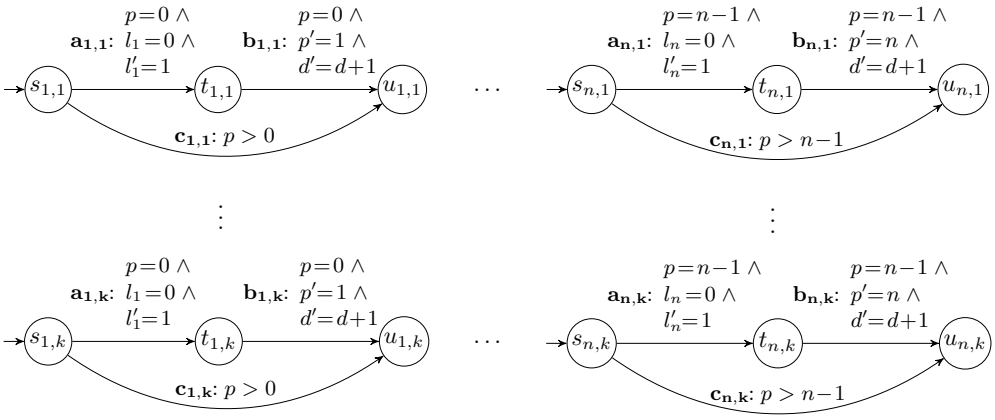
The above proof shows that for logical completeness having a single trace transformer *EventSplit* suffice. Surely, the assumption of having such a powerful assertion as Acc is completely unrealistic: for any non-trivial program this assertion would be overly complicated. Yet, such proofs do allow to highlight what are the most important constituents of a proof system.

We illustrate the refinement procedure of algorithm 3 with another example; we use it also to motivate the efficiency improving modifications of the algorithm, which are outlined in the next section. Consider the synchronized system with interleaving semantics presented in Figure 5.19. This is an acyclic transition system: it represents an abstraction of the initialization protocol, which proceeds in n phases, with k processes participating in every phase. In each phase one device needs to be started; this event is represented by the increment of variable d. The participating processes coordinate in such a way that the device is started *exactly* once, and all processes visit their respective final location $u_{i,j}$ only when the device is started. The coordination is attained by checking and assigning values to variable p (at the end of the i-th phase p has value i), as well as individual lock variable l_i for each phase.

The natural safety requirement can be formulated as follows: at the end of each phase i, exactly i devices have been started: $\bigwedge_{i\in[1,n]} \Box(p = i \implies d = i)$. The negation of this requirements can be written as

$$\bigvee_{i\in[1,n]} \Diamond(p = i \wedge d < i) \vee \bigvee_{i\in[1,n]} \Diamond(p = i \wedge d > i).$$

Let us restrict our attention only to the first phase for now, and consider only the first disjunct. It represents the error condition $\mathbf{e} \equiv (p = 1 \wedge d < 1)$: if



$\mathbb{S} = \langle S_{1,1}, \ldots, S_{n,k}, \mathbb{T}, \Delta \rangle,\ S_{i,j} = \langle V_{i,j}, T_{i,j}, \Theta_{i,j} \rangle$

$\mathbb{T}$ and Δ are defined in the interleaving semantics

$V_{i,j} = \{pc_{i,j}, p, d, l_i\}$

$T_{i,j} = \{\mathbf{a_{i,j}} \equiv (s_{i,j} \wedge t'_{i,j} \wedge p{=}i{-}1 \wedge l_i{=}0 \wedge l'_i{=}1 \wedge p'{=}p \wedge d'{=}d)\,,$

$\mathbf{b_{i,j}} \equiv (t_{i,j} \wedge u'_{i,j} \wedge p{=}i{-}1 \wedge p'{=}i \wedge d'{=}d{+}1 \wedge l'_i{=}l_i)$

$\mathbf{c_{i,j}} \equiv (s_{i,j} \wedge u'_{i,j} \wedge p{>}i{-}1 \wedge p'{=}p \wedge d'{=}d \wedge l'_i{=}l_i)\}$

$\Theta_{i,j} \equiv (s'_{i,j} \wedge p' = 0 \wedge d' = 0 \wedge l'_i = 0)$

Figure 5.19: Initialization protocol with n phases, and k processes per phase

a state, satisfying it, is ever reached, then the system is erroneous, because the first device was not started at all. The trace unwinding which analyzes this error condition is shown in Figure 5.20. Node 1 is labeled with the trace where the error is required to happen. This trace is unsatisfiable: initial condition Θ specifies that $p = 0$, while $\mathbf{e}$ requires that $p = 1$. This unsatisfiable trace consists only of two events; therefore, our refinement procedure applies the *LastNecessaryEvent* transformer with the Craig interpolant $p \neq 1$, the consequence of Θ. The newly introduced event, labeled with $p \neq 1 \wedge p' = 1$ is instantiated with all possible transitions that satisfy its label, namely $\mathbf{b_{1,1}}$ through $\mathbf{b_{1,k}}$. We show here only the analysis for $\mathbf{b_{1,1}}$, which occurs in the label of node 2.

The trace of node 2 is again unsatisfiable, because its events are labeled with the formulas $d = 0$, $d' = d + 1$, and $d < 1$. This time the unsatisfiable trace consists of 3 events; thus, *SafetyRefinement* executes the last branch. It computes the Craig interpolant between the first two events and the last one; this interpolant is $d' \geq 1$. The procedure splits the middle event with this formula, and obtains nodes 3 and 4. Now, consider node 3: its trace cannot be concretized, but now the unsatisfiable subtrace is shorter: it consists of only the last two events. The refinement procedure applies *LastNecessaryEvent* (the new edge label is not shown for readability). The Craig interpolant is $d \geq 1$, and the newly introduced event is labeled with the formula $d \geq 1 \wedge d' < 1$. This time,

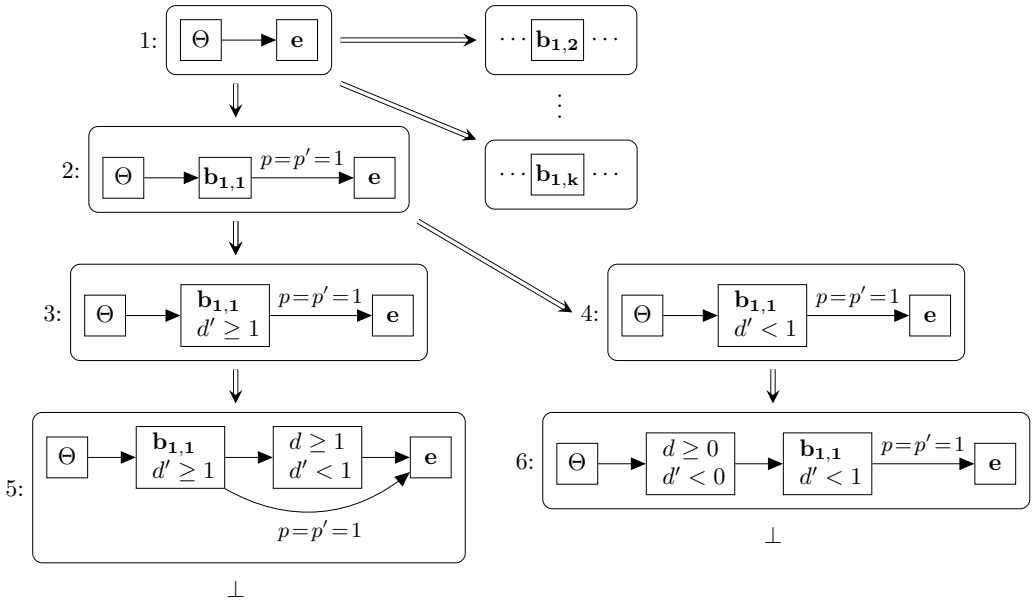


Figure 5.20: Trace unwinding for the system of Figure 5.19 and error condition $\mathbf{e} \equiv (p{=}1 \wedge d{<}1)$

however, the event cannot be instantiated: there is no transition in the system which decreases d. Therefore, node 5 is closed as contradictory.

Similarly for node 4: the unsatisfiable subtrace now consists of the first two events, and the interpolant is is $d \geq 0$. Again, there is no transition in the system satisfying $d \geq 0 \wedge d' < 1$, and the branch is closed. The same process, as outlined, will close the branches with the transitions $\mathbf{b_{1,1}}$ through $\mathbf{b_{1,k}}$, and we conclude that the error $(p = 1 \wedge d < 1)$ is unreachable. Note that the number of nodes in the tableau is linear, despite the exponential number of reachable states.

5.5 Trace Tableau

As we have demonstrated above, trace unwinding is adequate for program falsification. As some studies of real world concurrency bugs show [48], *"almost all (92%) of the examined concurrency bugs are guaranteed to manifest if certain partial order among no more than 4 memory accesses is enforced."* From that observation we may conclude that no algorithmic improvements to the trace unwinding exploration are needed for program falsification. Indeed, most errors will be caught by choosing an appropriate heuristic, which selects the unwinding branch that contains the erroneous computation (this leaves, though, a wide field for the development of heuristics for particular application domains; the area that we leave for future research).

But what do we do, when we have caught most errors? Is 92% assurance enough? For some applications it may be true, but for the vast majority this is definitely not enough: no one will fly an airplane having only 92% chance to

reach its destination. Therefore, for most applications we need a formal proof of their correctness. Thus, in this and subsequent sections we focus solely on *proof construction*.

As we have seen, trace unwinding can also be applied for constructing proofs for acyclic programs. Still, even for this simple class of programs its efficiency can be improved. Consider again the initialization example from Figure 5.19, and suppose that we want now to verify the second part of the negated safety requirement, namely $\bigvee_{i\in[1,n]} \Diamond(p = i \wedge d > i)$. This error condition specifies that after the end of phase i more than i devices have been started. With pen and paper we can disprove this requirement by induction: show that the base case, for $i = 1$, holds; then show that the requirement holds for the phase $i+1$, *assuming* it holds for the phase i. In this section we propose an extension of trace unwinding, which we call *trace tableau*, that automates such inductive reasoning.

Trace tableau extends trace unwinding with the possibility to refer to other parts of the proof, if they are applicable at the current point, without duplicating them. We first explain the concept by way of example, and then provide the formal definition. Consider the negated safety requirement for the initialization protocol from the above paragraph; let us restrict our attention to the first two disjuncts only: $\mathbf{e_1} \equiv (p = 1 \wedge d > 1)$ and $\mathbf{e_2} \equiv (p = 2 \wedge d > 2)$. Figure 5.21 represents the trace tableau for these two error conditions. Everything in the tableau is as defined for the unwinding, with the exception of the double dashed arrow: we will return to it later. In the figure, we omit or replace with "..." many irrelevant details.

The analysis starts with two root nodes 1 and 2, labeled with the traces where $\mathbf{e_1}$ and $\mathbf{e_2}$ are reached, respectively. Suppose, we have carried out the analysis of node 1 already, and fugired out that the subtree underneath node 1 is closed. This will be our *base case*. The subtree of node 1 is equivalent, modulo constants, to that of node 2, so we do not show it here.

Consider now node 2: it's trace is unsatisfiable because on the conditions on variable p, and we consider one possible extension of it with transition $\mathbf{b_{2,1}}$. The trace of node 3 is again unsatisfiable: transition $\mathbf{a_{2,1}}$ needs to be executed before $\mathbf{b_{2,1}}$; we introduce it, and get the trace of node 4. This trace is, in turn, unsatisfiable because of the following conditions on variable d: $d = 0$, $d' = d$, $d' = d + 1$, $d > 2$. We apply *EventSplit* twice, and split events labeled with $\mathbf{a_{2,1}}$ and $\mathbf{b_{2,1}}$ with Craig interpolants $d' \leq 1$ and $d' \leq 2$ respectively. As a result, we get the nodes 5, 6, and 7.

One important point to observe now is that the trace of node 5 contains the trace of node 1 as a subtrace; namely, the two initial conditions match, and we can match event $\mathbf{e_1}$ to event $\mathbf{a_{2,1}}$. We have that the label of $\mathbf{a_{2,1}}$, which is $(s_{2,1} \wedge t'_{2,1} \wedge p{=}1 \wedge l_2{=}0 \wedge l'_2{=}1 \wedge p'{=}p \wedge d'{=}d \wedge d'{>}1)$, implies the label of $\mathbf{e_1}$, which is $(p{=}1 \wedge d{>}1)$. Thus, the trace inclusion holds between the traces of nodes 5 and 1, which implies their language inclusion. We do know already that the language of node 1 is empty; thus, we conclude that the language of node 5 is empty as well. In the tableau construction we *cover* node 5 with node 1, which tells us that we can reuse the proof of node 1, and apply it to the subtree of node 5.

The remaining parts of the tableau are analyzed in the same way as for the unwinding. Consider node 6: here the conditions on variable d require the introduction of some event that increases it. It can be shown that only the transition

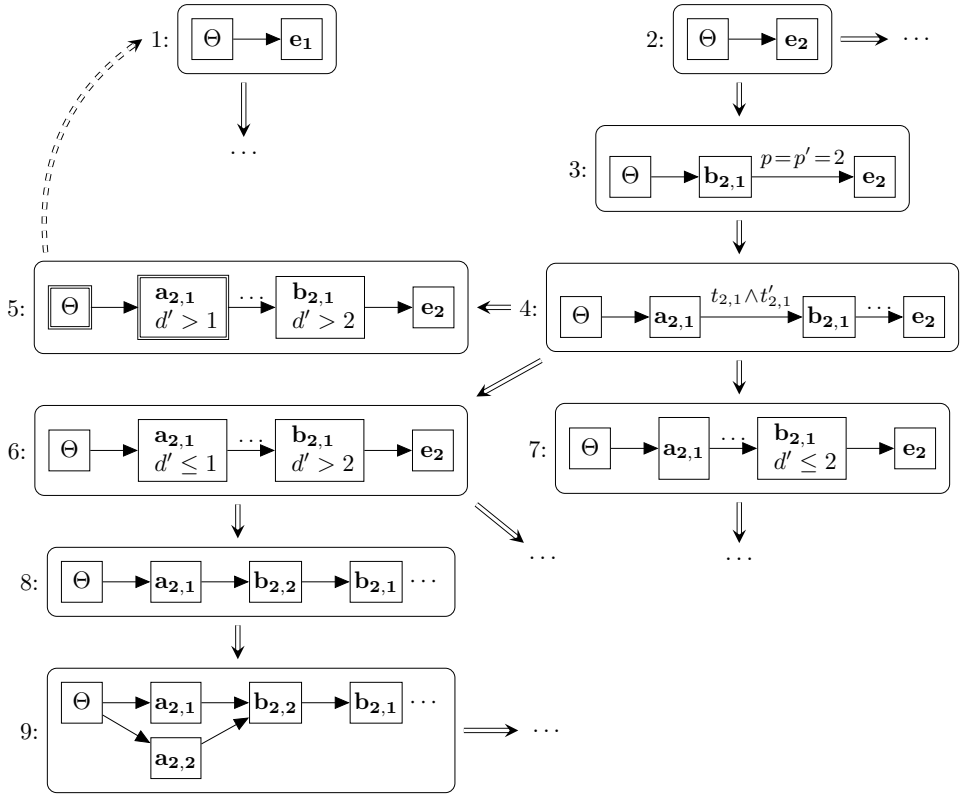


Figure 5.21: Trace tableau for the system of Figure 5.19 and error conditions $\mathbf{e_1} \equiv (p{=}1 \wedge d{>}1)$ and $\mathbf{e_2} \equiv (p{=}2 \wedge d{>}2)$

from the process of the same phase is applicable; suppose, it's transition $\mathbf{b_{2,2}}$. After we introduce it in node 8, execution of transition $\mathbf{a_{2,2}}$ becomes necessary, and we introduce it in node 9. Now, the trace of node 9 represents the mutual exclusion between events $\mathbf{a_{2,1}}$ and $\mathbf{a_{2,2}}$ over the same lock variable l_2. We will consider this example in more details in the next section; now it is sufficient to say that node 9 can be closed after a few more steps.

Note that covering in the tableau allowed us to separate the reasoning for the initialization example into layers, each layer corresponding to one phase of the protocol execution. In Figure 5.21 the reasoning at layer 2 was split into proving that the processes of phase 2 do not start the device twice (node 6 and beyond), and that the processes of the previous phase 1 do not start the device twice (node 5). The last part is delegated to layer 1. It is also worth noting that *without* covering, which represents proof reuse, the tableau would be exponential in n, because the reused parts would have to be repeated for each layer. On the contrary, *with* covering, the tableau is quadratic in n and k.

Definition 5.24 (**Trace tableau**)**.** For a transition system $S = \langle V, T, \Theta \rangle$, we define a *trace tableau* as a tuple $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$, where:

- $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is a trace unwinding;
- $\rightsquigarrow \subset N_L \times N_I$ is a *covering relation*; for $(n, n') \in \rightsquigarrow$ we call n a *covered* node, and n' a *covering* node;

A trace tableau is a trace unwinding extended with the covering relation $\rightsquigarrow$, which specifies sharing between proof parts. We cover a node n by another node n' when the trace of node n is more specific that the trace of node n', i.e., when language inclusion holds between these traces. This is expressed formally by the following definition.

Definition 5.25 (**Properties of trace tableau**)**.** For a given transition system S, we call a trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$ *correct* if:

1. $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is a correct trace unwinding;
2. for all $(n, n') \in \rightsquigarrow$ we have that $\mathcal{L}\big(\gamma(n)\big) \subseteq \mathcal{L}\big(\gamma(n')\big)$.

We call a trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$ *sound* , when the corresponding trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is sound. We call Γ *acyclic* if the relation $(E \cup \rightsquigarrow)$ is acyclic. We say that a trace tableau Γ is *complete* if for all its leaves $n \in N_L$ which are *uncovered*, i.e. there is no $(n, n') \in \rightsquigarrow$, their labels $\gamma(n)$ are contradictory.

A trace tableau, as defined above, represents in a compact way a trace unwinding, where for each covering $(n, n') \in \rightsquigarrow$ we can reuse for node n the proof constructed for node n'. The language inclusion can be tested by the algorithm of Section 4.3. One very simple way to cover n by n' is when the trace of n contains the trace of n' as a subtrace. In that case we have trace inclusion, which, as Proposition 4.23 shows, implies language inclusion. A nice feature of using trace inclusion instead of more general language inclusion is its visual appeal: if trace inclusion holds, then we can "plug in" the subtree of node n' at the position of node n, and reproduce all proof steps.

$S = \langle V, T, \Theta \rangle$
$V = \{x, y\}$
$T = \{\mathbf{a} \equiv (x'=1 \wedge y'=y),\ \mathbf{b} \equiv (y'=1 \wedge x=1 \wedge x'=0),\ \mathbf{c} \equiv (y'=0 \wedge x'=x)\}$
$\Theta \equiv (x'=0 \wedge y'=0)$
$\mathbf{e} \equiv (x=0 \wedge y=0)$

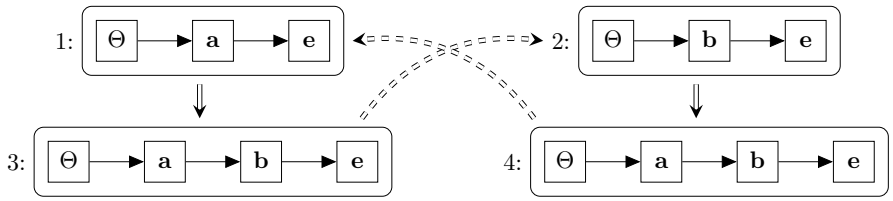


Figure 5.22: Unsoundness of trace tableau with loops

It is important that the covering relation does not create loops in the tableau: in the presence of loops a complete tableau does not represent a valid proof. This can be demonstrated by the following counterexample.

Example 5.26 (**Unsoundness of trace tableau with loops**)**.** Figure 5.22 shows a transition system S with three transitions **a**, **b**, **c** over the variables x, y. The lower part of the figure shows the trace tableau with two root nodes 1 and 2. Their traces specify the following two error conditions: transitions **a** or **b** happen, which assign 1 to variables x and y respectively, but after that the system returns to its initial state ($x = 0 \wedge y = 0$), represented by **e**. Applying *NecessaryEvent* to node 1 produces node 3: transition **b** is necessary between **a** and **e**. Similarly for node 4: transition **a** is necessary between **b** and Θ. At this point trace of node 3 includes trace of node 2 as a subtrace, while trace of node 4 includes trace of node 1 as a subtrace; consequently, we cover node 3 by node 2, and node 4 by node 1 (represented as dashed double lines in the figure). The resulting trace tableau is complete, and we wrongly conclude that no violating system computation exists. In fact, extending traces of nodes 3 and 4 with transition **c** between **b** and **e** would produce a counterexample trace.

Algorithm 4 constructs a proof for a given transition system and a safety property. It is a slight modification of Algorithm 3: before applying refinement to the concurrent trace $\gamma(n)$, it checks whether node n can be covered by another node n'. Functions *InitialAbstraction*, *Apply*, and *SafetyRefinement* are inherited from Algorithm 3 without changes.

Theorem 5.27 (**Soundness of trace tableau exploration**)**.** *Let a transition system S and a safety property φ be given. If Algorithm 4 terminates and returns* ***property holds****, only sound trace transformers are applied in SafetyRefinement, and the resulting tableau is acyclic, then all system computations of S satisfy φ.*

Proof. Assume that algorithm 4 terminates with the final trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$, and returns ***property holds***. In that case the queue Q is empty, and all leaves of Γ are either contradictory, or covered by other nodes in the tableau.

Algorithm 4: Exploration of Trace Tableau

Input : transition system $S = \langle V, T, \Theta \rangle$, safety property φ
Output: **property holds/counterexample**
Data: trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$, queue $Q \subseteq N_L$ of unexplored nodes, trace transformer τ, trace morphism m

begin
- set $\langle N, E, \gamma, \delta, \mu \rangle \longleftarrow \mathit{InitialAbstraction}(S, \varphi)$
- set $\rightsquigarrow \longleftarrow \emptyset$
- set $Q \longleftarrow N$
- **while** Q *not empty* **do**
  - select some n from Q
  - **if** $\mathit{Satisfiable}\big(\gamma(n)\big)$ **then**
    - **return *counterexample*** $\gamma(n)$
  - **else if** *exists* $n' \in N$ *such that* $\mathcal{L}\big(\gamma(n)\big) \subseteq \mathcal{L}\big(\gamma(n')\big)$ *and* $\big(E \cup \rightsquigarrow \cup \{(n, n')\}\big)$ *is acyclic* **then**
    - set $\rightsquigarrow \longleftarrow \rightsquigarrow \cup \{(n, n')\}$
  - **else**
    - set $\langle \tau, m \rangle \longleftarrow \mathit{SafetyRefinement}\big(\gamma(n)\big)$
    - $\mathit{Apply}\big(\Gamma, \tau, m, n\big)$
  - set $Q \longleftarrow Q \cup \{\, n' \mid (n, n') \in E \,\} \setminus \{n\}$
- **return *property holds***

Figure 5.23: Exploration of trace tableau

Assume that only trace inclusion is used for the language inclusion test. As the resulting tableau is acyclic, we can construct from it a sound and complete trace unwinding. For that, repeat the following process till saturation:

1. Take any pair $(n, n') \in \rightsquigarrow$, such that no other such pair is reachable from n' in the relation $E \cup \rightsquigarrow$ (it exists do to the acyclicity of $E \cup \rightsquigarrow$).

2. Add a fresh node n'', and duplicate the subtree of n' to n''. This subtree doesn't contain covering edges by the condition above.

3. Remove the covering edge (n, n') from $\rightsquigarrow$, and add a standard edge (n, n'') to E. This edge can be labeled with the sound trace transformer because of the soundness condition of trace trableau.

The process is guaranteed to terminate, and at the end we obtain a sound and complete trace unwinding. Therefore, by Theorem 5.21, no violating system computation exists.

Alternatively, if the more general language inclusion test is used, then we can traverse the tableau bottom-up, like it is done above, and, because of its acyclicity, show that the language of any covering node is empty. Therefore, we can replace covering edges with closing the covered nodes as contradictory. □

Theorem 5.28 (**Completeness of trace tableau exploration**). *Algorithm 4 is complete for* ***acyclic*** *transition systems, i.e. it always terminates for such systems. Moreover, there are acyclic transition systems and properties, for which trace tableaux are exponentially more succinct than trace unwindings.*

Proof. The first part is the same as in Theorem 5.22. The initialization protocol example proves the second part: any trace unwinding for it has exponential size in n, while its trace tableau is of polynomial size, as we have shown before. □

5.6 Causal Loops and Looping Trace Tableau

We have shown that trace tableaux can represent compactly proofs for acyclic concurrent programs. But most programs do contain cycles; how do we extend trace tableau to cyclic state spaces? At the beginning of the chapter we talked informally about *causal loops*: this is the notion we use to mirror state space cycles in tableau proofs. Now we are in a position to formalize this notion.

Definition 5.29 (**Causal path**)**.** We define a *causal path* as an ordered sequence $\tau_1, \ldots, \tau_k$ of trace productions $\tau_i : (L_i \xrightarrow{r_i} R_i)$ such that for all $1 \leq i < k$ we have $R_i \subseteq L_{i+1}$.

For any trace $F \subseteq L_1$ we define the *application* of the causal path to the trace as a sequence $F_0 = F$, $F_i = \tau_i(F_{i-1})$, for $1 \leq i \leq k$.

If the the starting trace production τ_1 can be applied again at the end of a causal path, we get a causal loop.

Definition 5.30 (**Causal loop**)**.** We define a *causal loop* as a causal path $\tau_1, \ldots, \tau_k$, where $\tau_i : (L_i \xrightarrow{r_i} R_i)$, such that $R_k \subseteq L_1$.

For any trace $F \subseteq L_1$ we define the *application* of the causal loop to the trace as a sequence $F_0^0 = F$, $F_i^j = \tau_i(F_{i-1}^j)$, for $1 \leq i \leq k$, and $F_0^{j+1} = F_k^j$.

Thus, a causal loop is simply a cyclic sequence of trace transformations. But, as Example 5.26 shows, allowing such unrestricted loops in a trace tableau leads to unsoundness. What we need is to restrict ourselves to causal loops which, when applied to any trace, would pump it infinitely with new events.

Definition 5.31 (**Soundness of causal loops**)**.** We say that a causal loop $\tau_1, \ldots, \tau_k$, where $\tau_i : (L_i \xrightarrow{r_i} R_i)$, is *sound*, if for any concurrent trace $F \subseteq L_1$ its size increases beyond any bound under the application of the causal loop:

$$\exists k \geq 0 \,.\, \forall i \geq k \,.\, |F_0^{i+1}| > |F_0^i|.$$

If a causal loop is sound, we can allow it to be present in a trace tableau without affecting its soundness. We define *looping trace tableaux* as an extension of standard trace tableaux with causal loops.

Definition 5.32 (**Looping trace tableau**)**.** We define a *looping trace tableau* $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$ in the same way as the standard trace tableau; we extend labeling μ to covered nodes: for all $(n, n') \in \rightsquigarrow$ we have $\mu : \gamma(n') \to \gamma(n)$. For a given transition system S, we call a looping trace tableau Γ *correct* if:

1. $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is a correct trace unwinding;
2. for all $(n, n') \in \rightsquigarrow$ we have that $\gamma(n) \subseteq_{\mu(n)} \gamma(n')$.

Given any finite or infinite path in a tableau as an ordered sequence of edges $\Lambda = (n_1, n_2), (n_2, n_3), \ldots$ where $(n_i, n_{i+1}) \in E \cup \rightsquigarrow$, we define its corresponding causal path as a sequence of trace productions labeling the edges from E:

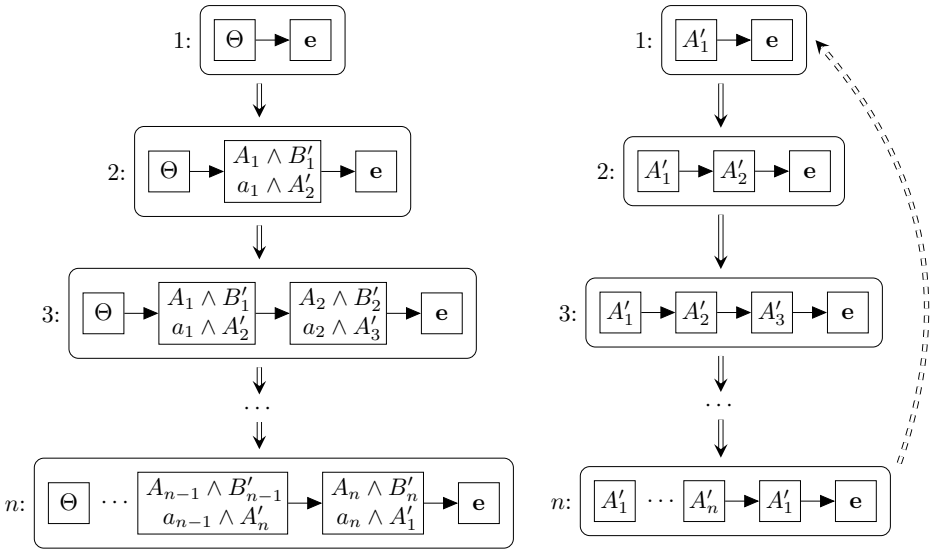


Figure 5.24: Trace unwinding (*left*) and looping trace tableau (*right*) for the example of Figure 5.1. $\Theta \equiv (A'_1 \wedge \ldots \wedge A'_n)$, and $\mathbf{e} \equiv (B_1 \wedge \ldots \wedge B_n)$

$\delta\big((n_{i_1}, n_{i_2})\big), \delta\big((n_{i_2}, n_3)\big), \ldots$, where $(n_{i_j}, n_{i_{j+1}}) \in \Lambda \cap E$. We have a causal loop in a tableau if for some k we have $n_k = n_1$.

We call a looping trace tableau *sound*, when the corresponding trace unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is sound, and, additionally, all causal loops of Γ are sound. We say that a looping trace tableau Γ is *complete* if for all its leaves $n \in N_L$ which are *uncovered*, i.e. there is no $(n, n') \in \rightsquigarrow$, their labels $\gamma(n)$ are contradictory.

Now we can return to the example from Figure 5.1, consisting of a chain of n automata with binary synchronization. Consider the left part of Figure 5.24: it shows the trace unwinding for the example; for simplicity, we do not outline any labels on causal links. Node 1 is labeled with the trace, representing the error scenario: all automata start in their top locations A_i, and finish in their bottom locations B_i. The trace is unsatisfiable, and in node 2 we extend it with the necessary event that synchronizes over a_1, and brings the first automaton to B_1, and the second automaton to A_2.

The trace of node 2 is again unsatisfiable, but this time only because of the last two events: the post-condition of the first event requires that the second automaton is in the state A_2, while event $\mathbf{e}$ requires that it is in location B_2. We extend, the trace with the next necessary event, and obtain the trace of node 3. The process continues, till we insert all n necessary events, and get the trace of node n.

Semantically, the sequence of trace productions, labeling the edges between the tableau nodes, represents a causal loop. Indeed, starting from node n we could repeat the outlined trace transformations infinitely, inserting the same events over and over again. There is a small syntactic problem though: the trace of node n *does not* contain the trace of node 1 as a subtrace. The only obstruction here is that the initial condition Θ specifies more information than

needed: it gives the initial locations for all of n automata, while it is enough to specify only that the first automaton is in location A_1.

This is where the second ingredient of incorporating causal loops in the tableau construction comes in. We *abstract* the traces of the node labels in such a way that they contain only the information necessary to repeat all transformations in the subtree of a particular node. The right part of Figure 5.24 shows the (abstract) looping trace tableau for the example. The only information that we need to preserve for the event introduced in the trace of node i is the post-condition A'_i. Now, we see that the trace of node n does contain the trace of node 1 as a subtrace, and we can cover node n with node 1, thus completing the causal loop and the proof of error unreachability. Note that the single causal loop is sound, because after each iteration of it the length of any trace in its language is increased by n.

Looping trace tableaux are our ultimate goal: they are enough to prove safety of any transition system for which a finite proof exists, as the following two theorems show.

Theorem 5.33 (**Soundness of looping trace tableau**). *Let a transition system S and a safety property φ be given. If there exists a correct, sound and complete looping trace tableau Γ such that every trace from $Abstract(S, \varphi)$ is subsumed by the label of some node of Γ, then all system computations of S satisfy φ.*

Proof. We prove by contradiction. Suppose some system computation $\pi \in \mathcal{L}(S)$ violates φ; note that this computation should be finite, because φ is a safety property. Then, there is a trace $F \in Abstract(S, \varphi)$ such that $\pi \in \mathcal{L}(F)$. By assumption, there is a correct, sound and complete looping trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$, and a node $n_1 \in N$ such that $\mathcal{L}(F) \subseteq \mathcal{L}\big(\gamma(n)\big)$. The tableau is correct and sound; thus, we can build a tableau path $\Lambda = (n_1, n_2), (n_2, n_3), \ldots$, where $(n_i, n_{i+1}) \in E \cup \rightsquigarrow$, such that $\pi \in \mathcal{L}\big(\gamma(n_i)\big)$ for all i. There are two possibilities:

1. Λ is finite. Then it ends in a contradictory node n_f, and $\pi \in \mathcal{L}\big(\gamma(n_f)\big) = \emptyset$.

2. Λ is infinite. Then, there should be a node n_∞ which appears infinitely often in Λ. Let $F_1, F_2, F_3, \ldots$ be the sequence of traces from the application of the causal path of Λ to F, which we get when Λ passes through n_∞. Each sequence of edges between a pair of neighboring occurrences of n_∞ forms a causal loop in Γ, and by assumption, all these causal loops are sound. Therefore, we have that $|F_1| < |F_2| < |F_3| < \ldots$, and $\pi \in \mathcal{L}(F_i)$. Thus, π cannot have finite length; a contradiction.

□

In [36], Henzinger, Majumdar and Raskin establish a classification of infinite-state transition systems into five increasingly comprehensive classes STS1 to STS5 with respect to the decidable properties; the authors describe also symbolic algorithms for each system class. These symbolic algorithms work provided some *region algebra* exists, which describes such operations on symbolic regions as intersection *And*, difference *Diff*, emptiness check *Empty*, and predecessor computation *Pre* (we refer the reader to [36] for further details). The

most comprehensive class the authors describe, STS5, has decidable reachability properties, and can be analyzed by the symbolic semi-algorithm Reach, which aggregates predecessors starting from some set of goal regions. As it is described in [42] by Kupferman and Vardi (see also Section 2.1 for more details), such reachability checking algorithm can decide general safety properties using the monitor automaton construction. In the next theorem we show that looping trace tableau can simulate algorithm Reach, and is thus complete for the most comprehensive class STS5 of infinite-state systems with decidable safety properties.

Theorem 5.34 (**Completeness of looping trace tableau**). *Suppose that a transition system $S = \langle V, T, \Theta \rangle$ has a finite bounded reachability quotient (i.e., it belongs to the class STS5). If S satisfies a safety property φ, then there exists a correct, sound and complete looping trace tableau Γ for S and φ.*

Proof. We assume that the construction of [42] is applied, and safety checking is reduced to reachability. By assumption, there is a region algebra with all necessary operations, and Reach terminates for S and the reachability property obtained from φ. We prove the theorem by providing a simple algorithm that can simulate Reach. Our algorithm constructs a looping trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$, where all nodes are labeled with linear traces consisting of two events. For any tableau node n, let n_1 and n_2 denote the first and the second event of its label, respectively. The algorithm works as follows:

1. Create a tableau, where nodes are labeled by the traces from the set $\mathit{Abstract}(S, \varphi)$: these traces all consist of two events, where the first one is the system initial condition Θ, and the second describes one of the goal regions.

2. Apply *Contradiction* to all leaf nodes with contradictory labels (this removes them from the set of leaves). If there is a leaf node $n' \in N_L$ such that its label is satisfiable, return *unsafe*; if there are no more leaf nodes, return *safe*; otherwise proceed to step 3. We use the region algebra operation *Empty* for the contradiction and satisfiability checks.

3. Take the set of leaf nodes N_L. For every leaf node $n \in N_L$, apply *BackwardUnrolling* to n and events n_1, n_2 of its label (we use the region algebra operation *Pre*), and abstract the result by keeping only the first two trace events. We obtain a new leaf node n' for each former leaf node n.

4. For each leaf node $n' \in N_L$, and each internal node $n \in N_I$, do

   a) Apply *EventSplit* to node n', its event n'_2, and predicate $\lambda_{\mathcal{E}}(n_2)$, and obtain two new nodes n'_+, n'_- (we use the operations *And* and *Diff* from region algebra). Event n'_{+2} is labeled with $\lambda_{\mathcal{E}}(n'_2) \wedge \lambda_{\mathcal{E}}(n_2)$, while event n'_{-2} is labeled with $\lambda_{\mathcal{E}}(n'_2) \wedge \neg\lambda_{\mathcal{E}}(n_2)$. Add a covering edge from n'_+ to n; this is possible, because we can map event n_1 to n'_{+1}, and event n_2 to n'_{+2}, and the label of n'_{+2} implies the label of n_2.

   b) Assign $n' \longleftarrow n'_-$

5. Go to step 2.

The algorithm above, at each execution of its main cycle, unrolls the transition relation backwards one step more, starting from the set of goal regions. After each unrolling we slice away those regions, that we already encountered before, and cover these sliced regions by other nodes. The set of labels of the second events of leaf nodes at the end of each iteration of the main loop represents the *onion slice*: the set of regions, obtained by applying the transition relation to the previous onion slice, and never encountered before: $\bigvee_{n' \in N'_L} \big(\lambda_{\mathcal{E}}(n'_2)\big) = pre_T\big(\bigvee_{n \in N_L} \lambda_{\mathcal{E}}(n_2)\big) \wedge \bigwedge_{n \in N_I} \big(\neg \lambda_{\mathcal{E}}(n_2)\big)$. Due to the existence of a finite-state quotient, the algorithm terminates.

Moreover, all causal loops in the tableau are sound. To observe this, take into account, that the set of leaves after the i-th iteration of the algorithm main loop, represents the set of backward system computations of size i from one of the goal regions (due to the application of *BackwardUnrolling*). We always cover the current leaf nodes with the nodes from one of the previous iterations. Thus, the size of any trace, when any causal loop from the tableau is applied to it, is increased by at least 1 with each iteration of the loop. □

We have shown that a looping trace tableau is able to succinctly represent a safety proof, whenever such proof exists. Another desirable property for a proof is its *efficient certification*: given a proof, can we check in polynomial time with respect to its size, whether the proof is indeed correct? Unfortunately, the soundness condition for causal loops doesn't satisfy this criterion: even a small tableau may contain an exponential number of loops (consider, e.g., the classical chain-of-diamonds construction, as in [69]). The problem is that soundness of causal loops is a global condition on the whole tableau. What we need, is a local condition, which can be checked efficiently at the time when we cover one node by another, and which would guarantee that all newly created causal loops are sound. There are many possible localizations of the soundness condition; here we outline the one which is particularly simple.

Definition 5.35 (**Forgetful trace inclusion**)**.** Let the trace inclusion $F' \subseteq_\mu F$ hold for a trace morphism $\mu = \langle \mu_{\mathcal{E}} : \mathcal{E} \to \mathcal{E}', \mu_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}' \rangle$. Let $\mu^{\mathcal{E}}$ be the image of $\mu_{\mathcal{E}}$: $\mu^{\mathcal{E}} = \{\, e' \in \mathcal{E}' \mid (e, e') \in \mu_{\mathcal{E}} \,\}$. We say that the above trace inclusion is *left-forgetful* (resp. *right-forgetful*), if for all $e' \in \mu^{\mathcal{E}}$ there exists $e'_\times \in \mathcal{E}' \setminus \mu^{\mathcal{E}}$ such that $(e'_\times, e') \in \mathcal{C} \cap \not\downarrow$ (resp. $(e', e'_\times) \in \mathcal{C} \cap \not\downarrow$). We call the trace inclusion *forgetful* if it is either left- or right-forgetful.

Intuitively, $F' \subseteq_\mu F$ is a forgetful trace inclusion, if we "forget" some event on the left or on the right when moving from F' to F, and this event is in conflict with all events that remain: this requirement ensures that $|F'| > |F|$. Recover that all trace productions in a tableau are context-bounded. The definition above, together with context-boundedness, allows us to simplify the check for soundness of causal loops in a tableau.

Proposition 5.36. Let a tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$ be given. If all trace productions in Γ are context-bounded, and for all coverings $(n, n') \in \rightsquigarrow$ in Γ we have that $\gamma(n) \subseteq_{\mu(n)} \gamma(n')$ is a forgetful trace inclusion, then all causal loops in Γ are sound.

Proof. Suppose, that all coverings are forgetful. Take any causal loop $\Lambda = (n_1, n_2), (n_2, n_3), \ldots, (n_k, n_1)$, where $(n_i, n_j) \in (E \cup \rightsquigarrow)$, . The loop should contain at least one covering edge; w.l.o.g., let $(n_k, n_1) \in \rightsquigarrow$. For some trace F,

let the sequence F_i^j be the application of Λ to F: $F_0^0 = F$, $F_i^j = \tau_i(F_{i-1}^j)$, for $1 \leq i \leq k$, and $F_0^{j+1} = F_k^j$.

We have that $F_0^j \subseteq \gamma(n_1)$, $F_0^{j+1} = F_k^j \subseteq \gamma(n_k)$, and $\gamma(n_k) \subseteq_{\mu(n)} \gamma(n_1)$ is a forgetful trace inclusion. All trace productions in Γ are context-bounded; therefore, we can partition events of the traces as follows (for some event $e_\times$, and sets $\mathcal{E}_0$, $\mathcal{E}_1$): $F_0^j = \mathcal{E}_0 \uplus \mathcal{E}\big(\gamma(n_1)\big)$, $F_0^{j+1} = \mathcal{E}_0 \uplus \mathcal{E}\big(\gamma(n_k)\big)$, and $\mathcal{E}\big(\gamma(n_k)\big) = \mathcal{E}\big(\gamma(n_1)\big) \uplus \{e_\times\} \uplus \mathcal{E}_1$, where event $e_\times$ is in conflict with all events from $\mathcal{E}\big(\gamma(n_1)\big)$. Thus, after j iterationd of the causal loop we have at least j events which are linearly ordered and in conflict with each other (one event $e_\times$ from each iteration). Therefore, $|F_0^j| \geq j$, and the causal loop is sound. □

5.7 Abstract Trace Tableau

The analysis shown in Figure 5.24 suggests the way to construct a looping trace tableau: keep side by side a trace unwinding and a looping trace tableau, and mirror in the looping tableau the proof steps taken in the unwinding. At the same time, in each node of the looping tableau keep track of the premises which are sufficient to repeat the proof steps in the subtree of that node. We formalize this idea as an *abstract trace tableau* below.

Definition 5.37 (Abstract trace tableau). We define *abstract trace tableau* as a tuple $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, where:

- $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is a (*concrete*) trace unwinding;
- $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$ is an (*abstract*) looping trace tableau;
- $\sigma : \hat{\gamma}(n) \to \gamma(n)$, for all $n \in N$, is a *concretization trace morphism*.

An abstract trace tableau is a trace tableau that carries for each node two labels: a concrete and an abstract one. Concrete label $\gamma(n)$ is obtained as a result of a chain of applications of trace transformers on the path from some root to node n. Abstract label $\hat{\gamma}(n)$, on the other hand, represents conditions, sufficient to apply all trace transformers in the subtree which originates at node n. According to the concretization morphism $\sigma : \hat{\gamma}(n) \to \gamma(n)$, abstract labels are always subtraces of concrete labels. Thus, by Proposition 4.23, the language of the concrete label is contained in the language of the abstract label. Finally, $\hat{\delta} : E \to \Pi$ and $\hat{\mu} : pre\big(\hat{\delta}(n)\big) \to \hat{\gamma}(n)$ represent labeling of edges with trace productions, and of internal nodes with trace morphisms, respectively, relativized to abstract node labels.

Definition 5.38 (Properties of abstract trace tableau). We call an abstract trace tableau $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$ *correct* if $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ is a correct trace unwinding, $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$ is a correct looping trace tableau, and for all $n \in N$ we have $\gamma(n) \subseteq_{\sigma(n)} \hat{\gamma}(n)$. We call Δ *sound*, when both Υ and $\hat{\Gamma}$ are sound. We call Δ *complete*, when $\hat{\Gamma}$ is complete.

Algorithm 5 extends Algorithm 4 with the tracking of abstract labels. For that, we do two modifications to the previous algorithm.

First, given two nodes n and n', the covering check in function *TryCover* is done now not by checking the language inclusion $\mathcal{L}\big(\gamma(n)\big) \subseteq \mathcal{L}\big(\gamma(n')\big)$ between

Algorithm 5: Exploration of Abstract Trace Tableau

Input : transition system $S = \langle V, T, \Theta \rangle$, safety property φ
Output: **property holds/counterexample**
Data: abstract trace tableau $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, comprising unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$, and looping tableau $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$, queue $Q \subseteq N_L$, trace transformer τ, trace morphism m

begin
- set $\Delta \longleftarrow$ *InitialAbstractTableau*(S, φ), $Q \longleftarrow N$
- **while** Q *not empty* **do**
  - select some n from Q
  - **if** *Satisfiable*$\big(\gamma(n)\big)$ **then return** ***counterexample*** $\gamma(n)$
  - **else** $\langle pre(\tau), m \rangle \longleftarrow$ *TryCover*(Δ, n)
  - **else**
    - set $\langle \tau, m \rangle \longleftarrow$ *SafetyRefinement*$\big(\gamma(n)\big)$
    - *Apply*$\big(\Upsilon, \tau, m, n\big)$
  - set $Q \longleftarrow Q \cup \{\, n' \mid (n, n') \in E \,\} \setminus \{n\}$
  - *PropagateUp*$(\hat{\Gamma}, pre(\tau), m, n)$
- **return** ***property holds***

Function InitialAbstractTableau(S, φ)

In : system S, property φ
Out: abstract trace tableau $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$
begin
- set $\Upsilon \longleftarrow$ *InitialAbstraction*(S, φ), all of $\{\rightsquigarrow, \hat{\delta}, \hat{\mu}\} \longleftarrow \emptyset$
- **foreach** $n \in N$ **do** set $\sigma(n) \longleftarrow \emptyset$, $\hat{\gamma}(n) \longleftarrow$ empty trace

Function TryCover(Δ, n)

In : abstract trace tableau $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, node n
Out: $\langle$node n', morphism $m\rangle$ / $\perp$
begin
- **if** *exists* $n' \in N$, $m : \hat{\gamma}(n') \to \gamma(n)$ *s.t.* $\gamma(n) \subseteq_m \hat{\gamma}(n')$ *is forgetful* **then**
  - set $\hat{\gamma}(n) \longleftarrow \hat{\gamma}(n')$, $\sigma(n) \longleftarrow m$
  - put (n, n') into $\rightsquigarrow$
  - **return** $\langle \hat{\gamma}(n'), m \rangle$
- **else return** $\perp$

Procedure PropagateUp$(\hat{\Gamma}, pre(\tau), m, n)$

In : looping tableau $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$, premise $pre(\tau)$, morphism m, node n
begin
- **if** $\nexists\, \hat{m} = \langle \hat{m}_{\mathcal{E}}, \hat{m}_{\mathcal{C}} \rangle : pre(\tau) \to \hat{\gamma}(n)$ *such that* $m = \sigma \circ \hat{m}$ **then**
  - **foreach** $o \in \gamma(n)\,.\,\big(\exists\, o' \in pre(\tau)\,.\, o = \xi(o')\big) \wedge \big(\nexists\, o'' \in \hat{\gamma}(n)\,.\, o = \sigma(o'')\big)$ **do**
    - add o' to $\hat{\gamma}(n)$, and (o', o) to $\sigma(n)$
- let $\hat{m} = \langle \hat{m}_{\mathcal{E}}, \hat{m}_{\mathcal{C}} \rangle : pre(\tau) \to \hat{\gamma}(n)$ such that $m = \sigma \circ \hat{m}$
- **if** $\hat{\gamma}(n) \not\subseteq_{\hat{m}} pre(\tau)$ **then**
  - **foreach** $e \in \mathcal{E}\big(pre(\tau)\big)\,.\,\big(\lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \not\Longrightarrow \lambda_{\mathcal{E}}(e)\big)$ **do**
    - set $\lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \longleftarrow \lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \wedge \lambda_{\mathcal{E}}(e)$
  - **foreach** $c \in \mathcal{C}\big(pre(\tau)\big)\,.\,\big(\lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \not\Longrightarrow \lambda_{\mathcal{C}}(c)\big)$ **do**
    - set $\lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \longleftarrow \lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \wedge \lambda_{\mathcal{C}}(c)$
- **foreach** $(n', n) \in \rightsquigarrow$ **do**
  - **if** $\hat{\gamma}(n') \subseteq_{\mu(n')} \hat{\gamma}(n)$ *not forgetful* **then**
    - remove (n', n) from $\rightsquigarrow$
    - put n' into Q
- **if** $\exists$ *parent* $n'\,.\,(n', n) \in E$ **then**
  - set $\langle pre(\tau)', m' \rangle \longleftarrow$ *Pullback*$\Big(\delta\big((n', n)\big), m\Big)$
  - *PropagateUp*$(\hat{\Gamma}, pre(\tau)', m', n')$

Figure 5.25: Exploration of abstract trace tableau

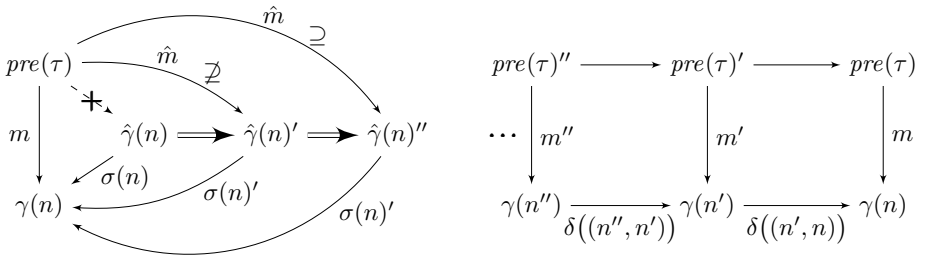


Figure 5.26: Upward propagation of premises in trace tableau. *Left*: calculation of abstract label $\hat{\gamma}(n)$ in procedure *PropagateUp*. *Right*: pullback construction, propagation of premise $pre(\tau)$ to parent nodes

their concrete labels, but by checking the trace inclusion $\gamma(n) \subseteq_m \hat{\gamma}(n')$ between the concrete label of n and the abstract label of n'. If this check succeeds, then the language inclusion holds as well. Additionally, we require the trace inclusion to be forgetful, to guarantee that all loops in the tableau are sound.

Second, we track the premises of the applied proof steps in procedure *PropagateUp*; this is explained graphically in Figure 5.26. Given as input the premise $pre(\tau)$ of some trace transformer τ applied at node n, and the mapping m of the premise to the concrete label $\gamma(n)$, the procedure adds missing components to the abstract label $\hat{\gamma}(n)$. This is done in two stages. In the first stage, the objects (events or links), which are present in $pre(\tau)$, but missing in $\hat{\gamma}(n)$, are inserted into $\hat{\gamma}(n)$, producing such $\hat{\gamma}(n)'$ that there is a mapping $\hat{m} : pre(\tau) \to \hat{\gamma}(n)'$. In the second stage, the labels in $\hat{\gamma}(n)'$ are adjusted in such a way (giving the trace $\hat{\gamma}(n)''$) that $\gamma(n) \subseteq_{\hat{m}} \hat{\gamma}(n)''$ holds. Because the abstract label $\hat{\gamma}(n)$ becomes more concrete, previous covering by that node may stop to hold; they are checked and uncovered as needed. Finally, the premise is propagated up in the tableau to a parent node n', if present, by constructing the premise $pre(\tau)'$ and the mapping $m' : pre(\tau)' \to \gamma(n')$ as a pullback object and arrow of two arrows m and $\delta\big((n', n)\big)$, where the latter is a trace transformation of $\gamma(n')$ into $\gamma(n)$. Then *PropagateUp* is called recursively with this new premise and mapping; the propagation process terminates either when the root node is reached, or when the abstract label $\hat{\gamma}(n)$ already contains all objects used in the premise, i.e. when the inclusion $\hat{\gamma}(n) \subseteq pre(\tau)$ holds.

Theorem 5.39 (**Soundness of abstract trace tableau exploration**). *Let a transition system S and a safety property φ be given. If Algorithm 5 terminates and returns* ***property holds****, and only sound trace transformers are applied in SafetyRefinement, then all system computations of S satisfy φ.*

Proof. The only way for Algorithm 5 to terminate with **property holds** is when queue Q is empty. By construction, the final abstract looping trace tableau $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$ is correct, sound, and complete. Moreover, for all traces $F \in Abstract(S, \varphi)$, there is a node $n \in N$ such that $F = \gamma(n) \subseteq \hat{\gamma}(n)$. Thus, the looping trace tableau $\hat{\Gamma}$ is a proof of correctness of S with respect to φ. □

Syntax	Semantics
acq l_i	$l_i = 0 \wedge l_i' = 1 \wedge pc' = pc + 1$
rel l_i	$l_i' = 0 \wedge pc' = pc + 1$
skip	$pc' = pc + 1$
goto j	$pc' = j$

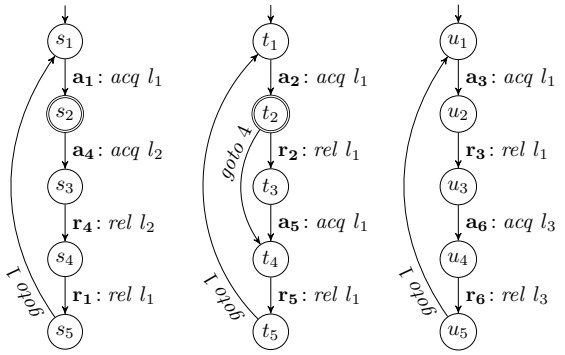


Figure 5.27: Class of multithreaded programs with binary semaphores. *Left:* Syntax and semantics. *Right:* Example system consisting of 3 threads with critical sections over 3 semaphore variables. Initial condition $\Theta \equiv (s_1' \wedge t_1' \wedge u_1' \wedge l_1' = 0 \wedge l_2' = 0 \wedge l_3' = 0)$, and error condition $\mathbf{e} \equiv (s_2 \wedge t_2)$

5.8 Polynomial Verification of Semaphore Programs

We finish this chapter with a more elaborated example, which illustrates the application of abstract trace tableau and Algorithm 5.

Example 5.40 (**Binary semaphore programs**)**.** We consider the class of multithreaded programs in the interleaved semantics, with critical sections protected by shared binary semaphores (or *locks*), which is described in [49]. A program in this class consists of n threads, executing in the interleaved fashion, and m shared boolean lock variables. Each thread contains some finite number of critical sections, protected by the "*acquire* lck_i" and "*release* lck_i" statements for one of the lock variables. Critical sections may be arbitrarily nested or intersected, and a thread may have an arbitrary control structure via the use of "*goto*" statements. The only restriction for a correct program is that the control flow may enter one of the critical sections for the lck_i variable only via the "*acquire* lck_i" statement, and may exit it either by jumping to another critical section for the same lock variable, or by executing the "*release* lck_i" statement. The syntax and semantics of such programs are shown in the left part of Figure 5.27; they can be used to analyze systems with built-in "test-and-set" primitive. In the following we consider the example program depicted on the right of Figure 5.27; we want to verify that threads 1 and 2 cannot be simultaneously at their critical sections 2, protected by lock l_1.

Standard model checking approaches require exponential, with respect to the number of threads, time and space to prove safety of such programs. In [49], Alexander Malkis developed a counterexample-guided refinement algorithm based on cartesian abstraction with exception sets, which is capable to solve in polynomial time the safety problem for the *restricted* class of programs with locks. The restricted class allows only one lock variable, prohibits nesting/intersection of critical sections, and disallows control flow transfers. The following open problem was posed:

> **Open Problem ([49], p.65).** Is the most general class of programs with locks polynomially verifiable for a fixed number of locks?

Here we settle this question affirmatively; moreover, Algorithm 5 finds safety proofs for the most general class of programs with locks using only polynomial time and space with respect both to the number of threads and to the number of locks.

Consider Figure 5.28: it depicts a part of the abstract trace tableau for the example program from the right part of Figure 5.27. The concrete unwinding is shown on the left, and the abstract tableau is shown on the right; all events are in conflict, so we do not show conflict links. Let us first concentrate on the left part. The trace of node 1 captures the error: after the initial state, the two first threads are simultaneously in their critical sections. This trace is unsatisfiable, and traces on nodes 2 and 3 extend it with two necessary events, which are labeled with transitions $\mathbf{a_1}$ and $\mathbf{a_2}$, respectively.

The trace of node 3 is unsatisfiable because of the conditions that transition $\mathbf{a_1}$ and $\mathbf{a_2}$ impose on the lock variable l_1. The events with these transitions are unordered, and an *OrderSplit* is performed. We consider only one possible ordering, shown in node 4, the other one is analyzed analogously.

After choosing one particular ordering, the conflict between the post-condition of $\mathbf{a_1}$ ($l_1' = 1$) and the post-condition of $\mathbf{a_2}$ ($l_1 = 0$) is present for all computations in the trace language. We apply the *LastNecessaryEvent* transformer, and instantiate the newly introduced event with all transitions which could possibly set l_1 to 0. There are four such transitions, namely $\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$, and $\mathbf{r_5}$. Wwe show the traces for the first three in nodes 5, 6, and 7, respectively; we omit the last one due to the lack of space.

Consider node 5: the link label between $\mathbf{a_1}$ and $\mathbf{a_2}$ requires that the first thread stays at location s_2, while transition $\mathbf{r_1}$, which is in the scope of the edge, can be executed only at location s_4: the node is closed as contradictory. Node 6 contains as a subtrace the trace of node 1: indeed, due to the combination of constraints of the link label between $\mathbf{a_1}$ and $\mathbf{a_2}$ and of event $\mathbf{r_2}$, the latter is labeled with the predicate $s_2 \wedge t_2$. Thus, we can cover node 5 with node 1 in the abstract tableau; we will return to this later. Finally, in node 7 we have the conflict between the initial condition Θ and the event labeled with $\mathbf{r_3}$: a necessary transition $\mathbf{a_3}$ is needed in between, which is inserted in the trace of node 8. From node 8 the analysis continues in a similar way.

Now, we turn our attention to the right part of Figure 5.28, depicting the abstract trace tableau for the example. Remember, that the abstract trace tableau represents for each node only those parts of its trace which are necessary to reproduce the proof steps in the subtree below this node. The abstract node labels are, therefore, much more concise than the concrete ones.

The abstract label of node 1 is the same as its concrete one; but already from node 2 they start to differ. In particular, in the abstract label of node 2, the full error condition reduces to only one conjunct, t_2, because the other one has already "fired". In node 3 the error condition is further reduced to $\top$, the most abstract predicate, because all of its conjuncts have fulfilled their purpose, and are not needed for further steps.

For the other nodes the abstract labels are in even larger contrast with the concrete ones; here we highlight the most striking differences. Consider node 5: the concrete label consists of 5 events with the full transition predicates; while

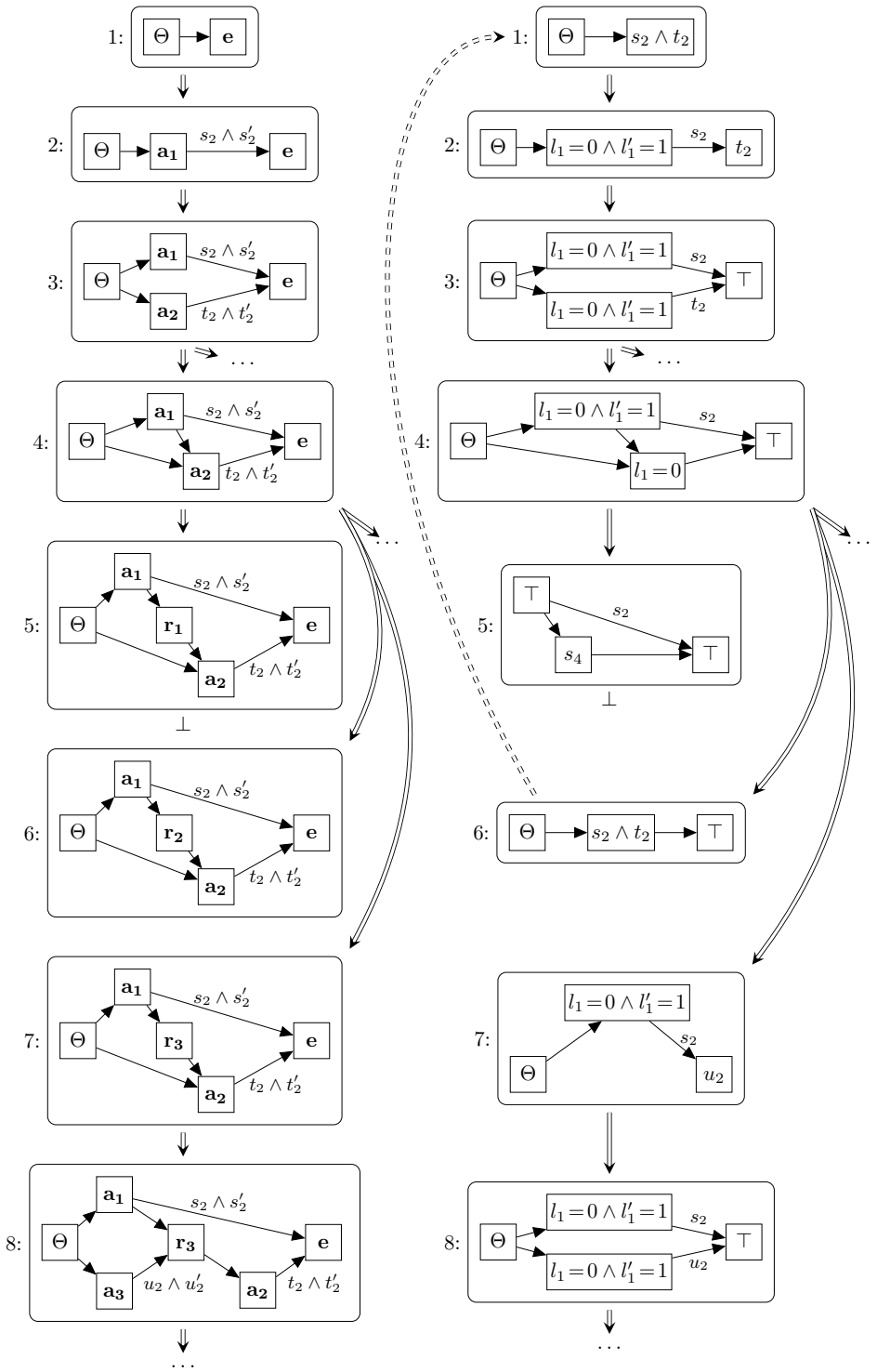


Figure 5.28: abstract trace tableau for the example of Figure 5.27.
$\Theta \equiv (s_1' \wedge t_1' \wedge u_1' \wedge l_1'=0 \wedge l_2'=0 \wedge l_3'=0)$, and $\mathbf{e} \equiv (s_2 \wedge t_2)$

the abstract one consists only of 3 events, and concisely represents the reason why the trace is contradictory. It is the case, because there are two events such that the link between them is labeled with s_2, and there is the third event in the scope of the link, which is labeled with s_4. Similarly for node 6: it's abstract label consists only of two events, which match exactly those in the trace of node 1; this justifies the covering of node 6 with node 1. Finally, compare abstract labels of nodes 3 and 8: despite the concrete labels are very different, the abstract ones are almost the same, modulo one difference. These abstract labels represent the core of the mutual exclusion proof: there are two concurrent events, each entering and staying in their respective critical section, and these events bear incompatible constraints on the lock variable, irrespective if their ordering.

It is easy to check, that the number of nodes in the tableau is proportional to the cubic power of the number of critical sections, while the size of the concurrent traces, labeling the vertices, is independent of the number of threads, critical sections, and locks. The execution of our algorithm takes at most quadratic time with respect to the number of nodes; thus, we have the following:

Theorem 5.41. *Algorithm 5 proves the safety of the most general class of multi-threaded programs with binary semaphors in deterministic polynomial time with respect to the number of threads and locks.*

Proof. For k critical sections, we have $O(k^2)$ top-level concurrent scenarios as in Figure 5.28, where there are two two transitions from different threads trying to get access to a critical section protected by the same lock variable. Analysis of each of them introduces a subtree with $O(k)$ branches. The length of each branch is limited by a constant, independent of k, namely by the length of the longest critical section; thus we have $O(k^3)$ nodes in the tableau. Application of trace transformers takes time which is independent from k. Only the search for coverings, in the worst case, can examine each existing node for a potential covering of a new vertex, thus giving the worst-case running time of $O(k^6)$. □

Chapter 6

Causality-based Verification: Liveness

In this chapter we specialize causality-based verification to *liveness properties*; see Section 2.2 for the problem statement and the discussion of the related work. As we describe there, *termination* can be considered a canonical liveness property. We defer the formal presentation of the causality-based approach for liveness in favor of a small example, to illustrate on the intuitive level how the approach works.

Consider the Producer-Consumer example presented in Figure 6.1, which is a simplified model of the *Map-Reduce* architecture from distributed processing: producers model the mapping step for separate data sources, consumers model the reducing step for different types of input data. The natural requirement for such an architecture is that the distributed processing terminates for any finite amount of input data.

While very successful for sequential software, the CEGAR-based termination provers Terminator [19] and *T2* [10, 20], and, likewise, termination provers based on classic techniques for term rewrite systems, such as *AProVE* [11, 31], can handle no more than two threads of the Producer-Consumer benchmark. On the contrary, the prototype implementation of the causality-based approach for termination, the termination prover Arctor [43] (for *A*bstraction *R*efinement of *C*oncurrent *T*emporal *Or*derings) scales to a large number of concurrent threads: for the Producer-Consumer benchmark Arctor proves termination for 100 threads in less than three minutes (the detailed experimental evaluation is presented in Section 7).

The CEGAR-based termination provers Terminator and T2 build on the *Ramsey*-based approach, which searches for a termination argument in the form of a *disjunction* of wellfounded relations. If the transitive closure of the transition relation is contained in the union of these relations, we call the disjunction a *transition invariant*; Ramsey's theorem then implies that the transition relation is wellfounded as well. The approach is attractive, because it is quite easy to find individual relations: one can look at the available program statements and take any decreasing transitions as hints for new relations. In the Producer-Consumer example, the termination can be proved with the disjunction of the following

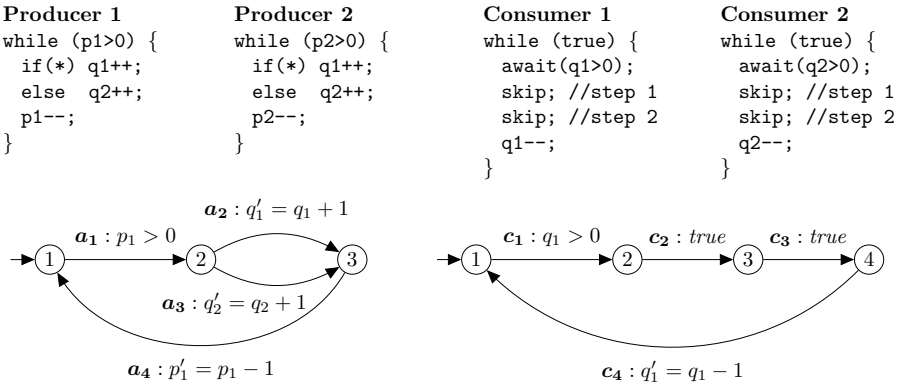


Figure 6.1: The *Producer-Consumer* benchmark, shown here for 2 producers and 2 consumers (*Top*: pseudocode; *Bottom*: control flow graphs with labeled transitions for Producer 1 and Consumer 1). The producer threads draw tasks from individual pools and distribute them to nondeterministically chosen queues, each served by a dedicated consumer thread; two steps are needed to process a task. The integer variables p_1 and p_2 model the number of tasks left in the pools of Producers 1 and 2, the integer variables q_1 and q_2 model the number of tasks in the queues of Consumers 1 and 2.

relations[1]: $p_1' < p_1$, $p_2' < p_2$, $q_1' < q_1$, and $q_2' < q_2$. The bottleneck of this approach is the containment check: with an increasing number of relations it becomes very expensive to check the inclusion of the transitive closure of the program transition relation in the transition invariant. Indeed, computation of the transitive closure is at least as hard as the computation of the reachable state space. As this part is already exponential for concurrent programs, it comes without surprise that the Ramsey-based termination proving technique, which uses reachability computation as a building block, also scales poorly with the number of concurrent threads.

Similar to the Ramsey-based approach, causality-based termination analysis works with multiple wellfounded relations that are individually quite simple and therefore easy to discover. The key difference is that we avoid disjunctive combinations, which would require us to analyze the transitive closure of the transition relation, and instead combine the relations only either *conjunctively* or based on a *case-split* analysis. Intuitively, our proof in the Producer-Consumer example makes a case distinction based on which thread might run forever. The case that Producer 1 runs forever is ruled out by the ranking function p_1. Analogously, Producer 2 cannot run forever because of the ranking function p_2. To rule out that one of the consumers, say Consumer 1, runs forever, we introduce the ranking function q_1, which allows an infinite execution of the `while` loop in Consumer 1 *only if* the `while` loop of Producer 1 or the `while` loop of Producer 2 also run forever, which we have already ruled out with the ranking functions p_1 and p_2. We discuss this example in more detail in Section 6.4; the informal reasoning should already make clear, however, that the case split has significantly simplified the proof: not only is the termination argument for the

[1] This ranking is slightly idealized: we omit components needed to account for the progress in the program counter, for example during the processing steps of the consumer threads.

individual cases simpler than a direct argument for the full program, the cases also support each other in the sense that the termination argument from one case can be used to discharge the other cases.

For the case of safety properties, the proof tableau was constructed stepwise, starting from the root nodes labeled with the set of finite concurrent traces capturing all possible property violations. For liveness properties in general, and for termination in particular, we construct a similar proof by contradiction that is also guided by the search for an erroneous computation. The difference to the safety case is that, instead of assuming the existence of a *finite* computation that leads to an error configuration, we start by assuming the existence of an *infinite* non-terminating computation, and then pursue the causal consequences that follow from this assumption. In this way, we build a tableau of potentially non-terminating traces. The discovery of a ranking function for the currently considered trace may either close the branch, if the rank decreases along all transitions, or result in one or more new traces, if the rank remains equal or increases along some transitions: in this case, we conclude that the existence of an execution for the current trace implies the existence of an execution for some other trace, in which at least one of these transitions occurs infinitely often.

We start in Section 6.1 with the description of a generalization from finite to infinite concurrent traces. In Section 6.2 we outline how infinite concurrent traces can represent violations of typical liveness properties, while in Section 6.3 we describe causality-based proof rules, which are specific to infinite computations. We finish this chapter with Section 6.4, where we outline the slight modifications to the verification algorithms from the previous chapter, needed to account for infinite computations, and provide a formal proof for the Producer-Consumer example described above.

6.1 Infinite Concurrent Traces

In order to reason about infinite computations and liveness properties, we generalize finite concurrent traces to infinite ones. We describe only the generalizations necessary for extending the Algorithm 5 (abstract tableau exploration) to the case of infinite computations.

Definition 6.1 (**Infinite concurrent trace**)**.** An *infinite concurrent trace* is a tuple $I = \langle F, F_\omega \rangle$, where:

- $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ is a finite concurrent trace called the *stem*;
- $F_\omega = \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega} \rangle$ is a finite concurrent trace called the *cycle*.

We denote the set of infinite concurrent traces by $\mathcal{I}$. For a given infinite concurrent trace $I = \langle F, F_\omega \rangle \in \mathcal{I}$ we denote its first component by $stem(I) = F$, and its second component by $cycle(I) = F_\omega$.

An infinite concurrent trace defines the set of infinite computations in the following way: the stem should occur once in the beginning of the computation, while the cycle should occur infinitely often after the stem.

Definition 6.2 (**Language of infinite concurrent trace**)**.** For a transition system $S = \langle V, T, \Theta \rangle$, the *language* of an infinite concurrent trace $I = \langle F, F_\omega \rangle$

is defined as a set $\mathcal{L}(I)$ of system computations such that for each computation $\pi = s_0, s_1, s_2, \ldots \in \mathcal{L}(I)$ there exists an infinite, strictly increasing sequence of indices $i_1, i_2, i_3, \ldots$ such that:

1. $s_0, s_1, \ldots s_{i_1} \in \mathcal{L}(F)$;
2. for all $k \geq 1$ it holds that $s_{i_k}, s_{i_k+1}, \ldots s_{i_{k+1}} \in \mathcal{L}(F_\omega)$.

From the above conditions we have that there exists an infinite sequence of mappings $\sigma_0, \sigma_1, \ldots$, called a *run* of I on π, where $\sigma_0 : \mathcal{E} \to \{s_0, s_1, \ldots s_{i_1}\}$, and, for all $k \geq 1$, $\sigma_k : \mathcal{E}_\omega \to \{s_{i_k}, s_{i_k+1}, \ldots s_{i_{k+1}}\}$.

Graphical Notation. The cycle part of the trace is depicted in round brackets, superscripted with ω. The opening round bracket corresponds to the cycle entry event $e_{\triangleleft\omega}$, while the closing round bracket corresponds to the cycle exit event $e_{\triangleright\omega}$. The stem part of a trace is depicted to the left of the opening bracket.

Inclusion of the infinitely repeating cycle part into concurrent traces makes the decision problems much harder than for finite concurrent traces. In particular, the transitivity restriction cannot recover the decidability of the emptiness checking, as the next theorem shows.

Theorem 6.3. *The emptiness problem for transitive infinite concurrent traces over any logic that includes atoms of the form $x - y = c$, for variables x, y ranging over $\mathbb{N}$, and constants $c \in [0, 1]$, is undecidable.*

Proof. We recycle here most of the proof of Theorem 4.19. The idea is that the existence of a non-halting computation of a Minsky machine can be easily represented by a simple infinite concurrent trace, where the cycle part offers the choice between different machine instructions, and checks the halting condition.

Let $S = \langle V, T, \Theta \rangle$ be the transition system as defined in the proof of Theorem 4.19. We define the modified transition system $S' = \langle V', T', \Theta' \rangle$. The set of variables is extended with variable *done* ranging over $\mathbb{N}$: $V' = V \cup \{done\}$; this variable tracks whether some instruction was executed within one cycle iteration. The initial condition is: $\Theta' = \Theta \wedge (done = 0)$.

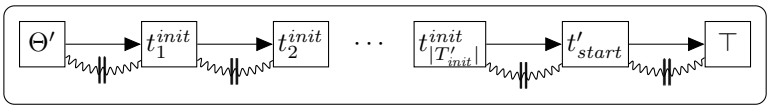
The transitions are $T' = T'_{init} \uplus \{t'_{start}\} \uplus T'_{main} \uplus \{t_{check}\}$, where:

- $T'_{init} = \{\, t_i \wedge pres(\{done\}) \mid t_i \in T_{init} \,\}$;
- $t'_{start} = t_{start} \wedge (done' = 1)$;
- $T'_{main} = \{\, t'_i \equiv t_i \wedge (done = 0) \wedge (done' = 1) \mid t_i \in T_{main} \,\}$;
- $t_{check} = (halt = 0) \wedge (done = 1) \wedge (done' = 0) \wedge pres(V \setminus \{done\})$.

The infinite concurrent trace $I = \langle F, F_\omega \rangle$ encodes the existence of a non-halting computation of the Minsky machine. The stem part requires that the transitions from T_{init} are executed in order one by one, followed by the execution of t'_{start}. Formally, $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, where:

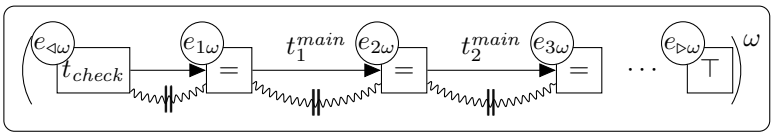
- $\mathcal{E} = \{e_\triangleleft, e_1, \ldots, e_{|T_{init}|}, e_{start}, e_\triangleright\}$;
- $\mathcal{C} = \{(e_\triangleleft, e_1), (e_1, e_2), \ldots, (e_{|T'_{init}|}, e_{start}), (e_{start}, e_\triangleright)\}$;

- $\lightning = \{(e_\triangleleft, e_1), (e_1, e_2), \ldots, (e_{|T'_{init}|}, e_{start}), (e_{start}, e_\triangleright)\}$;
- $\lambda_\mathcal{E} = \{e_\triangleleft \to \Theta', e_{start} \to t'_{start}, e_\triangleright \to \top\} \cup \{\, e_i \to t_i^{init} \mid t_i^{init} \in T'_{init} \,\}$;
- $\lambda_\mathcal{C} = \{(e_\triangleleft, e_1) \to \bot, \ldots, (e_{start}, e_\triangleright) \to \bot\}$.



The cycle part requires that transition t_{check} executes infinitely often, and some transition from the T'_{main} executes between each execution of t_{check}. Formally, the cycle part is $F_\omega = \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega} \rangle$, where:

- $\mathcal{E}_\omega = \{e_{\triangleleft\omega}, e_{1\omega}, \ldots, e_{|P|\omega}, e_{|P|+1\omega} = e_{\triangleright\omega}\}$;
- $\mathcal{C}_\omega = \{(e_{\triangleleft\omega}, e_{1\omega}), (e_{1\omega}, e_{2\omega}), \ldots, (e_{|P|\omega}, e_{\triangleright\omega})\}$;
- $\lightning_\omega = \{\, (e, e_{\triangleright\omega}) \mid e \in \mathcal{E}_\omega \setminus \{e_{\triangleright\omega}\} \,\}$;
- $\lambda_{\mathcal{E}\omega} = \{e_{\triangleleft\omega} \to t_{check}\} \cup \{\, e_i \to pres(V) \mid i \in 1 \ldots |P| \,\} \cup \{e_{\triangleright\omega} \to \top\}$;
- $\lambda_{\mathcal{C}\omega} = \{(e_{\triangleleft\omega}, e_{1\omega}) \to \bot\} \cup \{\, (e_{i\omega}, e_{i+1\omega}) \to t_i^{main} \mid t_i^{main} \in T'_{main} \,\}$.



Due to the conditions on variable *done*, there is a strict alternation of t_{check} and the transitions from T'_{main}. Indeed, some transition from T'_{main} should be executed in order to change the value of *done* from 0 to 1, and no more than one transition from T'_{main} can be executed, because each transition has as a precondition ($done = 0$). Because t_{check} also checks the condition ($halt = 0$), we have that the language of I is not empty if and only if the Minsky machine does not halt. □

As can be observed from the proof above, the infinite concurrent trace used is syntactically very primitive: every link label is transitive, because there can be at most one state change in its scope. Moreover, both the event and link labels are over a very simple logic, and can be easily made purely conjunctive. Therefore, there is no simple syntactic restriction that can recover decidability as it was the case for finite concurrent traces. Because already emptiness is undecidable, the complementation and language inclusion are undecidable as well.

Fortunately, the last algorithm we outlined in the previous chapter, does not depend on the precise language inclusion check. The underapproximating check based on trace inclusion is sufficient. Here we lift the trace inclusion relation to infinite concurrent traces.

Definition 6.4 (**Trace inclusion for infinite traces**)**.** For two infinite concurrent traces $I = \langle F, F_\omega \rangle$ and $I' = \langle F', F'_\omega \rangle$ we define the *trace inclusion relation* $\subseteq$ as follows: $I' \subseteq I$ iff there exists a pair of trace morphisms $\nu = \langle \mu, \mu_\omega \rangle$, called an *infinite trace morphism*, such that $F' \subseteq_\mu F$ and $F'_\omega \subseteq_{\mu_\omega} F_\omega$. We write $I' \subseteq_\nu I$ if trace inclusion holds for a particular infinite trace morphism ν.

Proposition 6.5. Structural inclusion between infinite concurrent traces implies also their language inclusion: if $I' \subseteq I$ then $\mathcal{L}(I') \subseteq \mathcal{L}(I)$.

Proof. Let I, I' – two infinite concurrent traces. Suppose that $I' \subseteq_\nu I$ for some infinite trace morphism ν, and there is a computation $s_0, s_1, s_2, \ldots \in \mathcal{L}(I')$. Then, by Definition 6.2 we have the sequence of indices $i_1, i_2, i_3, \ldots$ such that $s_0, s_1, \ldots s_{i_1} \in \mathcal{L}(F') \subseteq \mathcal{L}(F)$, and for all $k \geq 1$ it holds that $s_{i_k}, s_{i_k+1}, \ldots s_{i_{k+1}} \in \mathcal{L}(F'_\omega) \subseteq \mathcal{L}(F_\omega)$. Therefore, we have that $s_0, s_1, s_2, \ldots \in \mathcal{L}(I)$ for the same sequence of indices. □

As we have established previously for the looping trace tableau (Section 5.6), in order for the trace inclusion to be used in a sound way in a tableau proof, the inclusion should be forgetful (Definition 5.35). We generalize this notion to infinite concurrent traces.

Definition 6.6 (**Forgetful trace inclusion for infinite traces**)**.** Let the trace inclusion $I' \subseteq_\nu I$ hold for an infinite trace morphism $\nu = \langle \mu, \mu_\omega \rangle$. We say that it is *forgetful* if either $F' \subseteq_\mu F$ or $F'_\omega \subseteq_{\mu_\omega} F_\omega$ is a forgetful trace inclusion for finite traces.

Thus, a forgetful trace inclusion for infinite traces "forgets" some events either in the stem or in the cycle part of an infinite trace.

6.2 Representation of Liveness Properties

Similar to the case of safety properties, we assume the existence of the function *Abstract* that, given a transition system and a liveness property, described, e.g., in LTL, provides a set of *infinite* concurrent traces that encode all possible system runs that violate the property.

Definition 6.7 (**Abstract**)**.** For a transition system $S = \langle V, T, \Theta \rangle$ and a liveness property φ, the function $Abstract(S, \varphi) \in \mathcal{P}(\mathcal{I})$ gives a set of *infinite* concurrent traces such that:

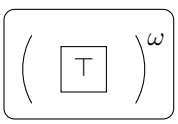
$$\mathcal{L}(S) \cap \mathcal{L}(\neg\varphi) \;=\; \bigcup_{I \in Abstract(S,\varphi)} \mathcal{L}(I)$$

Below we describe the encoding of some typical cases of liveness LTL formulas; the example LTL formulas are taken from [52].

Unconditional Termination. For a program P that performs some computational task, we expect that it finally terminates and delivers the computation result. The typical liveness property of *termination* can be expressed by the temporal formula

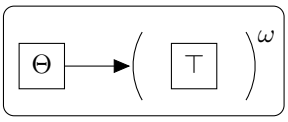
$$\Diamond\, after\_P.$$

All violations of such a property are encoded by the trace that represents an arbitrary non-terminating computation:



The above trace specifies that there should be some arbitrary, infinitely repeating transition in a non-terminating computation.

Conditional Termination. The above concurrent trace specifies that P does not terminate irrespective of the initial state in which P has started its computation. A refined concurrent trace encodes all non-terminating computations that start from some initial system state:



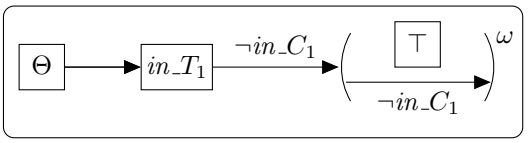
Accessibility. Consider again the scheme in Figure 5.2, where two processes P_1 and P_2 coordinate their access to critical sections C_1 and C_2. The natural progress property of *accessibility* for that program specifies that

> any process that wants to enter its critical section will eventually succeed.

The accessibility property for process P_1 may be expressed by the formula

$$\Box\big(\, in\_T_1 \implies \Diamond\, in\_C_1 \,\big).$$

The violations of the property are described by the concurrent trace



The trace represents all infinite system computations where process P_1 was in the trying section T_1, but after that it was never admitted to the critical section C_1.

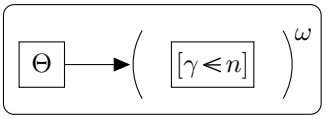
Eventual Boundedness. Let us reconsider the prime-printing program from Section 5.2. A relevant liveness property states that

> the sequence of printed numbers will eventually be bounded below by any positive integer.

This property can be specified, using a rigid integer variable n, by the formula

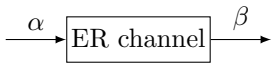
$$\Diamond\Box\neg[\gamma \lessdot n].$$

It states that any integer can be printed only finitely many times. All violations of such a property are represented by the trace



The above trace describes a computation, where there is an integer that is printed infinitely often.

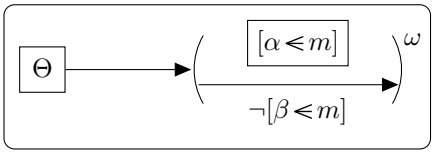
Eventual Reliability. Let S implement a channel that is *eventually reliable* (ER): if the sender keeps resubmitting the same message on α, then the ER channel will eventually transmit the message on β:



This property of eventual reliability can be specified by the formula

$$\Box\Diamond[\alpha \lessdot m] \implies \Box\Diamond[\beta \lessdot m].$$

It states that the message that is submitted infinitely many times is also transmitted infintely many times. The violations of the property are described by the concurrent trace



The trace specifies the infinite computations where some message is submitted infinitely often, but is transmitted only finitely many times.

6.3 Liveness Trace Transformers

Here we describe the trace transformers that are specific to infinite concurrent traces and the analysis of liveness properties.

Safety trace transformers are applicable to the *stem* of an infinite concurrent trace without any modifications. There are also variants of these transformers that are applicable to the *cycle* part of an infinite concurrent trace; we will denote such transformers using superscript $^\omega$, as in $\mathit{OrderSplit}^\omega$ or $\mathit{EventSplit}^\omega$. We will not repeat the full description for such transformers, and refer the reader to Section 5.3; the only change is that they are applied to the cycle part. Nevertheless, the soundness of such transformers needs to be argued separately; therefore we provide soundness proofs specific to infinite traces. Here we describe also another category of trace transformers that are applicable only to infinite concurrent traces.

As before, we introduce now some notation that will be used uniformly in all the soundness proofs further down. For any trace transformer τ, let $I = \langle F, F_\omega \rangle$ be a concurrent trace such that $I \subseteq_\nu \mathit{pre}(\tau)$ for a trace morphism ν. Let $\pi = s_0, s_1, s_2, \ldots \in \mathcal{L}(I)$ be some system computation from the language of I. Then there exists a run $\sigma_0, \sigma_1, \ldots$ of I on π, and an infinite sequence of indices $i_1, i_2, i_3, \ldots$ satisfying the conditions of definition 6.2. In each soundness proof below we show that there exists a trace production $\tau_i \in \tau$, and an infinite sequence of indices $i'_1, i'_2, i'_3, \ldots$ such that such that $s_0, s_1, s_2, \ldots \in \mathcal{L}\big(\tau_i^\nu(I)\big)$ for that sequence.

We start with the presentation of the liveness-specific trace transformers, and then describe the specializations of the safety transformers for infinite concurrent traces.

Invariance Split (Figure 6.2). The *InvarianceSplit*(φ) trace transformer makes a case distinction about the program behavior at infinity: for a given predicate φ either all events in the cycle part satisfy it, or a violating event should happen infinitely often. We exploit the rule when we introduce new events based on the ranking function: in that case the first branch is terminating, and we may consider only the second one. But, in general, the rule is useful without the *a priori* knowledge of a ranking function: it considers two cases, where each one is easier to reason about individually. Formally we have the following:

$pre\big(\mathit{InvarianceSplit}(\varphi)\big) = \big\langle F, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle$, where:

- $\mathcal{E}_\omega = \{e_{\triangleleft\omega}, e_{\triangleright\omega}\}$;
- $\lambda_{\mathcal{E}\omega} = \{e_{\triangleleft\omega} \to \top, e_{\triangleright\omega} \to \top\}$;
- $\mathcal{C}_\omega = \lightning_\omega = \lambda_{\mathcal{C}\omega} = \emptyset$.

$post\big(\mathit{InvarianceSplit}(\varphi)\big) = \{R_1, R_2\}$, where:

- $R_1 = \big\langle F, \langle \mathcal{E}_\omega, \mathcal{C}'_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda'_{\mathcal{C}\omega}\rangle\big\rangle$, where:
  - $\mathcal{C}'_\omega = \mathcal{C}_\omega \cup \{(e_{\triangleleft\omega}, e_{\triangleright\omega})\}$;
  - $\lambda'_{\mathcal{C}\omega} = \lambda_{\mathcal{C}\omega} \cup \{(e_{\triangleleft\omega}, e_{\triangleright\omega}) \to \varphi\}$.
- $R_2 = \big\langle F, \langle \mathcal{E}'_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda'_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle$, where:
  - $\mathcal{E}'_\omega = \mathcal{E}_\omega \cup \{a\}$;
  - $\lambda'_{\mathcal{E}\omega} = \lambda_{\mathcal{E}\omega} \cup \{a \to \neg\varphi\}$.

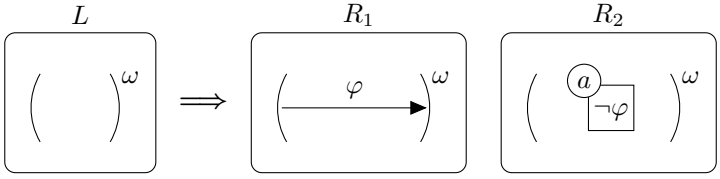


Figure 6.2: The *InvarianceSplit* trace transformer

Proposition 6.8. The *InvarianceSplit* trace transformer is sound.

Proof. Consider the infinite sequence of state pairs $(s_{i_1}, s_{i_1+1}), (s_{i_1+1}, s_{i_1+2}), \ldots$ Now, consider the subsequence $(s_{j_1}, s_{j_1+1}), (s_{j_2}, s_{j_2+1}), \ldots$, where all state pairs satisfy $\neg\varphi$. If this subsequence is infinite, then construct the infinite sequence $i'_1, i'_2, \ldots$, where, for each l, i'_l is selected as the maximum $i_k \leq j_l$. Then $\pi \in \mathcal{L}\big(R_2\big)$. If the subsequence is finite, then construct the sequence $i'_1, i'_2, \ldots$ as the suffix of the sequence $i_1, i_2, \ldots$, where i'_1 is equal to the minimum i_k such that all state pairs (s_j, s_{j+1}), for $j \geq k$, satisfy φ. Then $\pi \in \mathcal{L}\big(R_1\big)$. □

The following lemma is applied in the combination with the *InvarianceSplit* trace transformer.

Lemma 6.9. *Assume that a set S is well-ordered by a relation $\preceq$. If, for an infinite sequence $s_1, s_2, \ldots$ of elements from S, for an infinite number of pairs (s_i, s_{i+1}) it holds that $s_i \succ s_{i+1}$, then there exists an infinite number of pairs (s_j, s_{j+1}) such that $s_j \prec s_{j+1}$.*

Proof. By the contrary, suppose that there is only a finite number of pairs (s_j, s_{j+1}) such that $s_j \prec s_{j+1}$. Then there exists an index k, such that for all $i > k$ we have $s_i \succ s_{i+1}$ or $s_i \approx s_{i+1}$. Thus, there exists an infinite number of indices $i_1, i_2, \ldots$, such that for all j we have $s_{i_j} \succ s_{i_{j+1}}$. This is a contradiction with the well-order of $\preceq$. □

Here is how this lemma is used together with the *InvarianceSplit* trace transformer. Suppose that a ranking relation φ was found for some cycle trace F_ω. This fact implies that F_ω is terminating: a contradiction with the assumption that it has an infinite run. Therefore, we can apply the *InvarianceSplit*(φ) trace transformer, and discard the branch R_1 as contradictory, i.e., avoid a case split and transform F_ω by introducing the event labeled with $\neg\varphi$.

The next two rules exploit the repetitive character of the cyclic part of an infinite concurrent trace. They are defined for arbitrary and maximal finite subtraces F and F_ω: these subtraces comprise all events and links in the stem and the cycle part of a trace.

Cycle To Stem (Figure 6.3). The *CycleToStem* trace transformer shifts a copy of all events in the cycle part of a trace to its stem part.

$$pre\big(CycleToStem\big) = \big\langle\langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C}\rangle, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle.$$

$$post\big(CycleToStem\big) = \big\langle\langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_\mathcal{E}, \lambda'_\mathcal{C}\rangle, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle,$$

where:

- $\mathcal{E}' = \mathcal{E} \cup \mathcal{E}_\omega$;
- $\mathcal{C}' = \mathcal{C} \cup \{\, (e, e') \mid e \in \mathcal{E}, e' \in \mathcal{E}_\omega \,\}$;
- $\lightning' = \lightning_\omega \cup \{\, (e, e') \mid e \in (\mathcal{E} \setminus \{e_\triangleright\}), e' \in \mathcal{E}_\omega \,\}$;
- $\lambda'_\mathcal{E} = \lambda_\mathcal{E} \cup \lambda_{\mathcal{E}\omega}$;
- $\lambda'_\mathcal{C} = \lambda_\mathcal{C} \cup \lambda_{\mathcal{C}\omega}$.

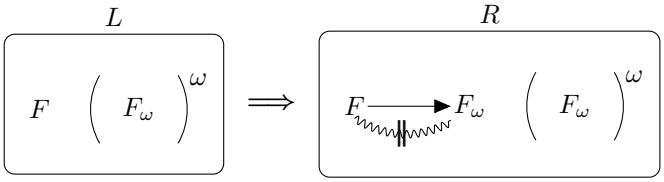


Figure 6.3: The *CycleToStem* trace transformer

Proposition 6.10. The *CycleToStem* trace transformer is sound.

Proof. We select the sequence of indicies $i'_1, i'_2, \ldots$ such that $i'_k = i_{k+1}$ for all $k \geq 1$. For this sequence of indicies we have $\pi \in \mathcal{L}\big(R\big)$. □

Cycle Unrolling (Figure 6.4). The *CycleUnrolling* trace transformer duplicates the cyclic part of a concurrent trace.

$pre\big(CycleToStem\big) = \big\langle F, F_\omega = \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \lightning_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle$.
Let $\langle \mathcal{E}'_\omega, \mathcal{C}'_\omega, \lightning'_\omega, \lambda'_{\mathcal{E}\omega}, \lambda'_{\mathcal{C}\omega}\rangle$ be the copy of all the elements of F_ω.
Then $post\big(CycleToStem\big) = \big\langle F, \langle \mathcal{E}''_\omega, \mathcal{C}''_\omega, \lightning''_\omega, \lambda''_{\mathcal{E}\omega}, \lambda''_{\mathcal{C}\omega}\rangle\big\rangle$, where:

- $\mathcal{E}''_\omega = \mathcal{E}_\omega \cup \mathcal{E}'_\omega$;
- $\mathcal{C}''_\omega = \mathcal{C}_\omega \cup \mathcal{C}'_\omega \cup \{\, (e, e') \mid e \in \mathcal{E}_\omega, e' \in \mathcal{E}'_\omega \,\}$;
- $\lightning''_\omega = \lightning_\omega \cup \lightning'_\omega{}' \cup \{\, (e, e') \mid e \in (\mathcal{E}_\omega \setminus \{e_{\triangleright\omega}\}), e' \in \mathcal{E}'_\omega \,\}$;
- $\lambda''_{\mathcal{E}\omega} = \lambda_{\mathcal{E}\omega} \cup \lambda'_{\mathcal{E}\omega}$;
- $\lambda''_{\mathcal{C}\omega} = \lambda_{\mathcal{C}\omega} \cup \lambda'_{\mathcal{C}\omega}$.

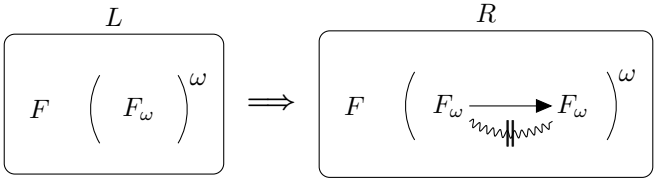


Figure 6.4: The *CycleUnrolling* trace transformer

Proposition 6.11. The *CycleUnrolling* trace transformer is sound.

Proof. We select the sequence of indicies $i'_1, i'_2, \ldots$ such that $i'_1 = i_1$ and $i'_k = i_{2k-1}$, for all $k \geq 2$. For this sequence of indicies we have $\pi \in \mathcal{L}\big(R\big)$. □

Often it is not possible to prove a liveness property without making any assumptions on the behavior of the environment. For concurrent programs, when the choice of the next transition to be executed is highly non-deterministic, these assumptions constraint the scheduler and force it to execute some transitions if they are enabled. These requirements are expressed formally by enriching a transition system $S = \langle V, T, \Theta\rangle$ with two sets of *just* and *compassionate* transitions $J, C \subseteq T$. Let $En(t)$ denote the enabling condition for a transition $t \in T$.

The requirement of a *weak fairness* (or *justice* in the terminology of Manna and Pnueli [52]) is that a just transition $t \in J$ that is *continuously* enabled (i.e., $En(t)$ holds continuously after some point), should be infinitely often taken. In some cases justice is not enough to guarantee fairness; then the requirement of a *strong fairness* (or *compassion*) is helpful: it states that a compassionate transition $t \in C$ that is *infinitely often* enabled (i.e., $En(t)$ holds infinitely often), should be infinitely often taken.

Causality-based verification offers two trace transformers that allow for a direct account of weak and strong fairness in liveness proofs.

Weak Fairness (Figure 6.5). The *WeakFairness*(t) trace transformer allows to introduce a just transition t in the cycle part of a trace in case it is continuously enabled.

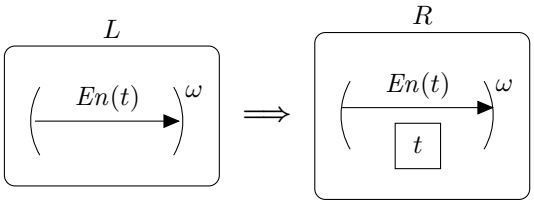


Figure 6.5: The *WeakFairness* trace transformer

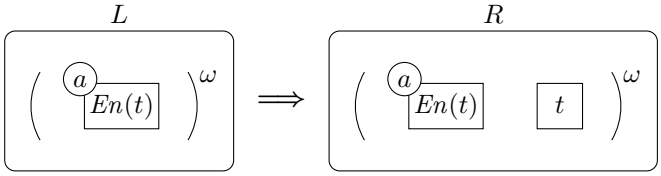


Figure 6.6: The *StrongFairness* trace transformer

$pre\big(WeakFairness(t)\big) = \big\langle F, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle$, where:

- $\mathcal{E}_\omega = \{e_{\triangleleft\omega}, e_{\triangleright\omega}\}$;
- $\lambda_{\mathcal{E}\omega} = \{e_{\triangleleft\omega} \to \top, e_{\triangleright\omega} \to \top\}$;
- $\mathcal{C}_\omega = \{(e_{\triangleleft\omega}, e_{\triangleright\omega})\}$;
- $\lambda_{\mathcal{C}\omega} = \{(e_{\triangleleft\omega}, e_{\triangleright\omega}) \to En(t)\}$.
- $\not\downarrow_\omega = (e_{\triangleleft\omega}, e_{\triangleright\omega})$.

$post\big(WeakFairness(t)\big) = \big\{\big\langle F, \langle \mathcal{E}'_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda'_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle\big\}$, where:

- $\mathcal{E}'_\omega = \mathcal{E}_\omega \cup \{a\}$;
- $\lambda'_{\mathcal{E}\omega} = \lambda_{\mathcal{E}\omega} \cup \{a \to t\}$.

Strong Fairness (Figure 6.6). The $StrongFairness(a, t)$ trace transformer allows to introduce a compassionate transition t in the cycle part of a trace if it is enabled infinitely often.

$pre\big(StrongFairness(a,t)\big) = \big\langle F, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle$, where:

- $\mathcal{E}_\omega = \{a\}$;
- $\lambda_{\mathcal{E}\omega} = \{a \to En(t)\}$;
- $\mathcal{C}_\omega = \lambda_{\mathcal{C}\omega} = \not\downarrow_\omega = \emptyset$.

$post\big(StrongFairness(a,t)\big) = \big\{\big\langle F, \langle \mathcal{E}'_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda'_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle\big\}$, where:

- $\mathcal{E}'_\omega = \mathcal{E}_\omega \cup \{b\}$;
- $\lambda'_{\mathcal{E}\omega} = \lambda_{\mathcal{E}\omega} \cup \{b \to t\}$.

Proposition 6.12. The *WeakFairness* and *StrongFairness* trace transformers are sound.

Proof. By a direct application of the definitions of weak and strong fairness. □

We finish this section with the specializations of some safety rules to infinite concurrent traces.

Necessary Cycle Event (Figure 6.7). The *NecessaryCycleEvent*(a, b, φ) trace transformer employs the same reasoning as the *NecessaryEvent*, but inside the cycle part of a concurrent trace. Suppose that we have two causally related and events a and b in the cycle of a concurrent trace, and a state predicate $\varphi \in \Phi(\mathcal{V})$, such that the label of a implies $\neg\varphi$, and the label of b implies φ'. Given the repetitive character of the cycle, we have that a should follow b again, but this is not possible due to the contradiction between their labelings. The transformer introduces a new "bridging" event c after b and before next a. In a formal setting we have:

$$pre\big(\mathit{NecessaryCycleEvent}(a,b,\varphi)\big) \;=\; \big\langle F, \langle \mathcal{E}_\omega, \mathcal{C}_\omega, \not\downarrow_\omega, \lambda_{\mathcal{E}\omega}, \lambda_{\mathcal{C}\omega}\rangle\big\rangle,$$

where:

- $\mathcal{E}_\omega = \{a, b\}$;
- $\mathcal{C}_\omega = \{(a, b)\}$;
- $\not\downarrow_\omega = \emptyset$;
- $\lambda_{\mathcal{E}\omega} = \{a \to \neg\varphi, b \to \varphi'\}$;
- $\lambda_{\mathcal{C}\omega} = \{(a, b) \to \top\}$.

$$post\big(\mathit{NecessaryCycleEvent}(a,b,\varphi)\big) \;=\; \big\langle F, \langle \mathcal{E}'_\omega, \mathcal{C}'_\omega, \not\downarrow'_\omega, \lambda'_{\mathcal{E}\omega}, \lambda'_{\mathcal{C}\omega}\rangle\big\rangle,$$

where:

- $\mathcal{E}'_\omega = \mathcal{E}_\omega \cup \{c\}$;
- $\mathcal{C}'_\omega = \mathcal{C}_\omega \cup \{(b, c)\}$;
- $\not\downarrow'_\omega = \not\downarrow_\omega \cup \{(b, c)\}$;
- $\lambda'_{\mathcal{E}\omega} = \lambda_{\mathcal{E}\omega} \cup \{c \to \varphi \wedge \neg\varphi'\}$;
- $\lambda'_{\mathcal{C}\omega} = \lambda_{\mathcal{C}\omega} \cup \{(b, c) \to \top\}$.

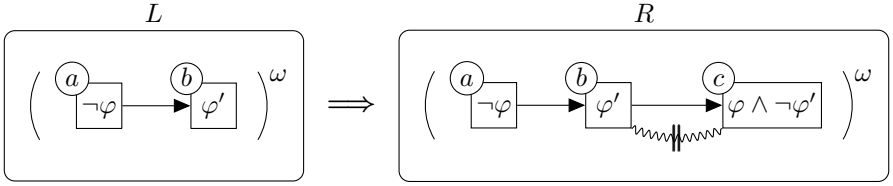


Figure 6.7: The *NecessaryCycleEvent* trace transformer

Proposition 6.13. The *NecessaryCycleEvent* trace transformer is sound.

Proof. For an arbitrary $k \geq 2$, let $\sigma(b) = s_{j_b}$ and $\sigma(a) = s_{j_a}$, where $i_{k-1} \leq j_b < i_k$ and $i_k \leq j_a < i_{k+1}$; such j_a, j_b should exist. We have that the formula $\varphi(s_{j_b+1})$ holds, and the formula $\neg\varphi(s_{j_a})$ holds. For any $j_b < j < j_a$, suppose that the formula $\varphi(s_j)$ holds, and consider two cases: either the formula $\varphi(s_{j+1})$ holds, or the formula $\neg\varphi(s_{j+1})$ holds. Starting from $j = j_b+1$, where the formula $\varphi(s_j)$ holds, we will find such j_x that the formula the formula $\neg\varphi(s_{j_x})$ holds. Otherwise we would have that for all $j_b < j \leq j_a$ the formula $\varphi(s_j)$ holds – a contradiction with the fact that $\neg\varphi(s_{j_a})$ holds. We then select i'_k to be equal to $j_x + 1$. As k was chosen arbitrarily, we obtain an infinite sequence of indices $i_1, i'_2, i'_3, \ldots$ for which $\pi \in \mathcal{L}\big(R\big)$. □

Order Split $^{\omega}$ (Figure 6.8). The $\mathit{OrderSplit}^{\omega}(a, b)$ trace transformer considers alternative orderings of two events a and b in the cycle part of a trace.

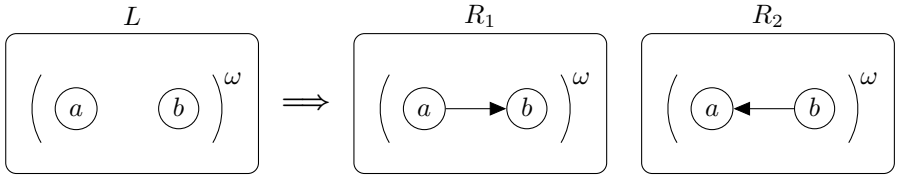


Figure 6.8: The $\mathit{OrderSplit}^{\omega}$ trace transformer

Proposition 6.14. The $\mathit{OrderSplit}^{\omega}$ trace transformer is sound.

Proof. For an arbitrary $k \geq 1$, let $\sigma_k(a) = s_{j_a}$ and $\sigma_k(b) = s_{j_b}$, where $i_k \leq j_a, j_b \leq i_{k+1}$. Two cases are possible: either $j_a \leq j_b$ or $j_a > j_b$. Let the sequence of indices $k_1^1, k_2^1, \ldots$ be such where $i_{k_l^1} < j_a \leq j_b \leq i_{k_l^1+1}$ holds, and the sequence of indices $k_1^2, k_2^2, \ldots$ be such where $i_{k_l^2} \leq j_b < j_a \leq i_{k_l^2+1}$ holds. At least one of these sequences should be infinite. W.l.o.g., let it be $k_1^1, k_2^1, \ldots$ Then we select $i'_l = i_{k_l^1}$, and have that $\pi \in \mathcal{L}\big(R_1\big)$. □

Event Split $^{\omega}$ (Figure 6.9). The $\mathit{EventSplit}^{\omega}(a, \varphi, \psi)$ trace transformer, given some event a in the cycle part of a trace, labeled with a transition predicate ψ, and another arbitrary transition predicate φ, considers two alternatives: a satisfies either φ or $\neg\varphi$ infinitely often.

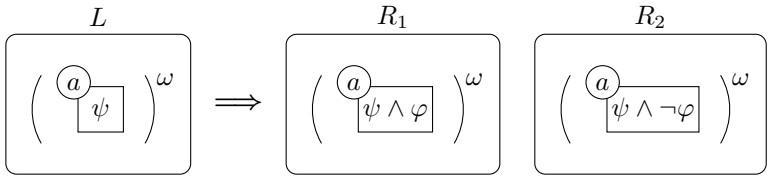


Figure 6.9: The $\mathit{EventSplit}^{\omega}$ trace transformer

Proposition 6.15. The $\mathit{EventSplit}^{\omega}$ trace transformer is sound.

Proof. There should exist an infinite sequence of indices $i_k \leq j_k \leq i_{k+1}$, for $k \geq 1$, such that $\sigma_k(a) = s_{j_k}$. Let j_l^1 and j_l^2 be two subsequences of j_k where the formulas $\varphi(s_{j_k}, s_{j_{k+1}})$ and $\neg\varphi(s_{j_k}, s_{j_{k+1}})$ hold. At least one of the subsequences should be infinite. W.l.o.g., let it be subsequence j_l^1. Then we construct the sequence of indices $i'_1, i'_2, \ldots$, where, for $l \geq 1$, we select i'_l as the maximum index $i_l \leq j_l^1$. Then $\pi \in \mathcal{L}\big(R_1\big)$. □

Instantiate $^{\omega}$ (Figure 6.10). Similarly to the case of finite traces, the $\mathit{Instantiate}^{\omega}(a, \varphi, \psi)$ trace transformer, given some event a in the cycle part of a trace, labeled with a transition predicate φ, instantiates it with all possible system transitions that satisfy φ. An additional predicate ψ can be used to restrict the potentially large set of conclusions of this trace transformer. The

underlying reasoning is, nevertheless, different to the finite-trace case: here we argue that some of the system transitions satisfying $\varphi \wedge \psi$ should happen *infinitely often*.

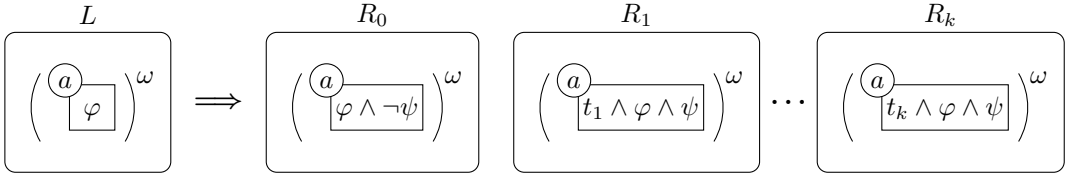


Figure 6.10: The *Instantiate*$^\omega$ trace transformer

Proposition 6.16. The *Instantiate*$^\omega$ trace transformer is sound.

Proof. Let $\{t_1, \ldots, t_k\} = \{\, t \in T \mid sat(t \wedge \varphi \wedge \psi) \,\}$. Consider the infinite sequence of state pairs (s_{j_k}, s_{j_k+1}) such that $\sigma_k(a) = s_{j_k}$; we have that $\varphi(s_{j_k}, s_{j_k+1})$ holds for all pairs from the sequence. For each pair exactly one of the following should hold: either $\neg\psi(s_{j_k}, s_{j_k+1})$, or $(\psi \wedge t_i)(s_{j_k}, s_{j_k+1})$, because π is a system computation. Due to the infinite pigeonhole principle, one of the finite number of alternatives should happen infinitely often. W.l.o.g., suppose that $\neg\psi(s_{j_l}, s_{j_l+1})$ hold for an infinite sequence $j_1, j_2, \ldots$ Then we construct the sequence of indices $i'_1, i'_2, \ldots$, where, for $l \geq 1$, we select i'_l as the maximum index $i_l \leq j_l$. Then $\pi \in \mathcal{L}\big(R_0\big)$. □

6.4 Tableau Proofs of Liveness Properties

Using the liveness trace transformers described above, we can generalize the notions of trace unwinding and different kinds of trace tableaux, which were introduced in Chapter 5, to liveness properties. The only changes needed are the replacement of finite concurrent traces with infinite ones (with the simultaneous replacement of trace operations), as well as the application of liveness trace transformers in the proofs. As these changes are straightforward, we do not detail them here.

All soundness proofs also carry over to the case of unwinding and tableaux for liveness properties. The only additional restriction is for the looping trace tableau: applications of *CycleToStem* and *CycleUnrolling* trace transformers should be disallowed to occur on the tableau causal loops, because they can pump the stem and the cycle part of an infinite concurrent trace indefinitely without restricting its language. If these trace transformers are not present in a looping tableau, then the proof of Theorem 5.33 generalizes to infinite traces. When a forgetful trace inclusion holds for the cycle part of the trace, this implies that a causal loop involving it will pump the cycle part indefinitely, thus contradicting the assumption of the cycle part repeating after a finite number of computation steps.

Relative completeness of trace unwinding also generalizes to liveness properties: the theorem below highlights that the *InvarianceSplit* trace transformer is the most important one between all liveness transformers.

Theorem 6.17 (**Relative completeness of trace unwinding for liveness properties**). *If a transition system S satisfies a liveness property φ, then there exists a sound and complete trace unwinding for S and φ.*

Proof. Let, according to Definition 6.7, the set of violating computations be represented by the set $\mathit{Abstract}(S, \varphi)$, and assume that S satisfies φ; then all traces in the set $\mathit{Abstract}(S, \varphi)$ should be terminating. Construct the initial unwinding, where nodes are labeled with these traces, and for each node consider separately two cases:

Unconditional Termination: in that case the node trace terminates for any initial configuration. Then there exists a (perfect) ranking function f from the states of S into some well-ordered by $\preceq$ domain, such that for all system transitions $t \in T$ we have that $t(s, s') \implies f(s) \succ f(s')$. In step 1, we perform the *InvarianceSplit* using $f(s) \succ f(s')$ as the transition predicate to split with. The first branch is terminating, and is closed as contradictory. According to Lemma 6.9, the second branch contains event a labeled with $f(s) \prec f(s')$. So, in step 2, we apply the *EventSplit* trace transformer as many times as there are system transitions, and obtain a separate branch for each system transition. Because each transition actually decreases f, all the branches are closed as contradictory.

Conditional Termination: in that case the node trace terminates from initial configurations, described by the Θ predicate, but may not terminate when started from other configurations. Then there exists a (perfect) ranking function f from the states of S into some well-ordered by $\preceq$ domain, such that *for any reachable state* s, and for all system transitions $t \in T$, we have that $t(s, s') \implies f(s) \succ f(s')$. Similar to [53], we use as given the assertion Acc that characterizes the set of reachable (accessible) system states.

Now, we apply the same construction as for unconditional termination, with the following changes. In step 1 we perform *InvarianceSplit* using the predicate $\big(f(s) \succ f(s')\big) \wedge \mathit{Acc}(s)$. As a result, event a of the second branch is labeled with $\big(f(s) \prec f(s)\big) \vee \neg\mathit{Acc}(s)$ Using an additional application of *EventSplit*, we split the right branch into two, where events a_1 and a_2 are labeled with $\big(f(s) \prec f(s')\big)$ and $\neg\mathit{Acc}(s)$, respectively. From the branch containing a_1 we proceed as in step 2 of the unconditional case. To the branch containing a_2 we apply the *CycleToStem* trace transformer, and move event a_2 to the stem part of the trace. As no state satisfying $\neg\mathit{Acc}(s)$ is reachable from the initial configuration, this branch is closed as contradictory. Thus, all the branches are closed as contradictory, and we obtain a sound and complete trace unwinding. □

Algorithms for the trace unwinding and trace tableau exploration can be adjusted appropriately. Here we outline the modifications to the last algorithm of the previous chapter, Algorithm 5, which explores an abstract trace tableau; see Algorithm 6. The modifications are as follows. If the covering attempt was unsuccessful, we do the following sequence of checks:

1. We check whether the stem and the cycle, concatenated, form a valid computation. If not, we apply the standard safety refinement procedure, which was outlined in the preceding chapter.

Algorithm 6: Exploration of Abstract Trace Tableau for Liveness

Input : transition system $S = \langle V, T, \Theta \rangle$, liveness property φ
Output: **property holds/possible counterexample**
Data: abstract trace tableau $\Delta = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, comprising unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$, and looping tableau $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$, queue $Q \subseteq N_L$, trace transformer τ, trace morphism m, ranking relation ψ

begin
  set $\Delta \longleftarrow \mathit{InitialAbstractTableau}(S, \varphi)$, $Q \longleftarrow N$
  while Q *not empty* **do**
    select some n from Q
    if $\langle \mathit{pre}(\tau), m \rangle \longleftarrow \mathit{TryCover}(\Delta, n)$ **then goto next**
    else if $\neg \mathit{Satisfiable}\big(\mathit{stem}\big(\mathit{CycleToStem}(\gamma(n))\big)\big)$ **then**
      set $\langle \tau, m \rangle \longleftarrow \mathit{SafetyRefinement}\big(\gamma(n)\big)$
    else if $\neg \mathit{Satisfiable}\big(\mathit{cycle}\big(\mathit{CycleUnrolling}^k(\gamma(n))\big)\big)$ **then**
      set $\langle \tau, m \rangle \longleftarrow \mathit{SafetyRefinement}^\omega\big(\gamma(n)\big)$
    else if $\psi \longleftarrow \mathit{Terminating}\big(\gamma(n)\big)$ **then**
      set $\langle \tau, m \rangle \longleftarrow \mathit{Instantiate}^\omega(a, \neg\psi, \top) \circ \mathit{InvarianceSplit}(\psi)$
    else
      return *possible counterexample* $\gamma(n)$
    $\mathit{Apply}\big(\Upsilon, \tau, m, n\big)$
    next: set $Q \longleftarrow Q \cup \{\, n' \mid (n, n') \in E \,\} \setminus \{n\}$
      $\mathit{PropagateUp}(\hat{\Gamma}, \mathit{pre}(\tau), m, n)$
  return *property holds*

Function Terminating(I)

In : infinite concurrent trace $I = \langle F, F_\omega \rangle$
Out: ranking relation $\psi \in \Phi(V \cup V')$ / $\bot$
begin
  set $\psi_s \longleftarrow \mathit{SSA}(F)$
  set $\psi_c \longleftarrow \mathit{SSA}(F_\omega)$
  return $\mathit{rank}(\psi_s, \psi_c)$

Figure 6.11: Exploration of abstract trace tableau for liveness properties

2. We check whether the cycle part, repeated some finite number of times, forms a satisfiable computation. if not, we apply the variants of the safety trace transformers inside the cycle part.

3. If both of the previous checks give us valid computations, we check, whether the cycle part is terminating, using some ranking function synthesis procedure. If it is, which contradicts the assumption of the cycle repeating infinitely often, we apply the *InvarianceSplit* trace transformer, and instantiate the new event to all possible system transitions that violate the ranking relation.

4. Finally, if none of the previous checks reveals any contradictions, we report a possible counterexample.

Notice that due to the incompleteness of the ranking function synthesis, as well as of the underapproximating check in step 2, we cannot be sure that the

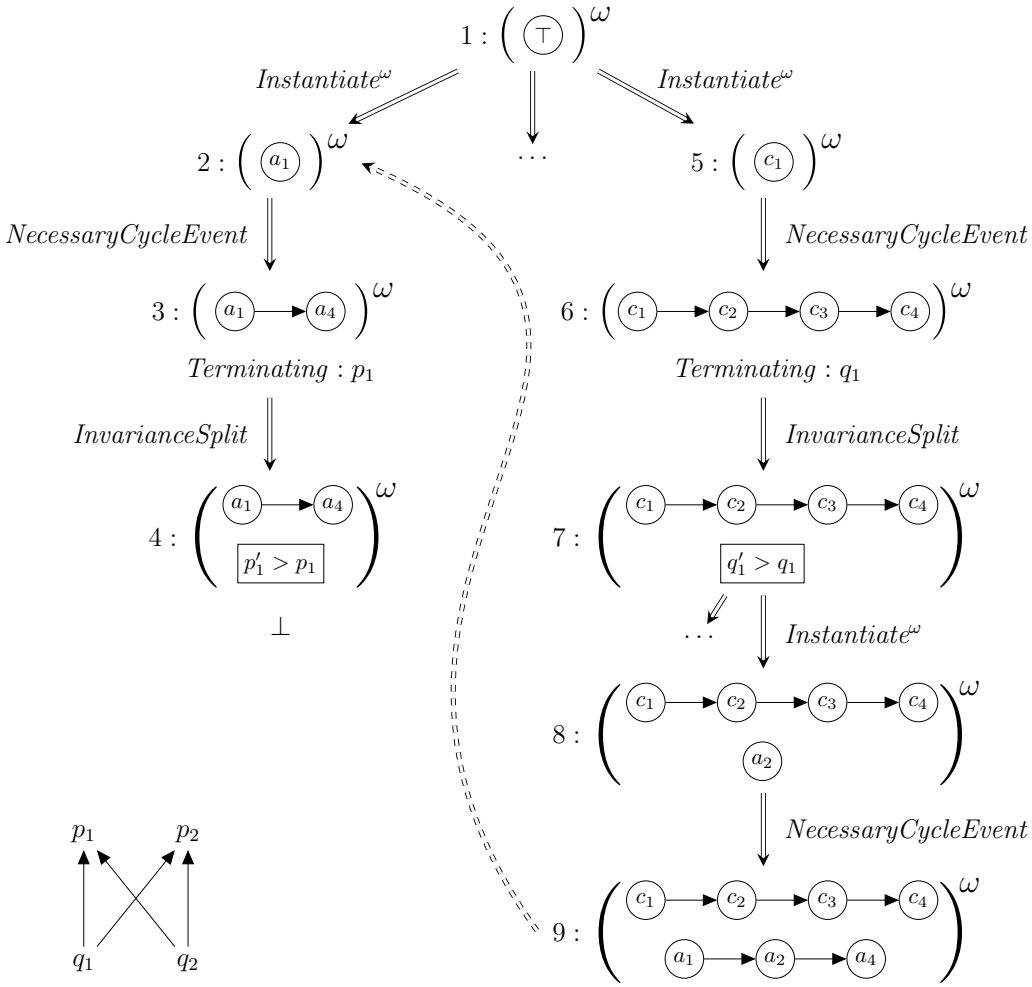


Figure 6.12: Termination proof for the Producer-Consumer example of Figure 6.1. *Bottom left*: partially ordered ranking function discovered in the analysis

counterexample is not spurious. This incompleteness is in a strong contrast with the safety case, where a satisfiable trace does represent a valid counterexample; it reflects the undecidability of the termination problem.

We finish this chapter with the tableau-based termination proof for the Producer-Consumer example of Figure 6.1; see Figure 6.12. Our analysis starts with the assumption (by way of contradiction) that there exists some infinite run. The assumption is expressed as the concurrent trace of node 1 in Figure 6.12: infinitely often some transition should occur. The transition is so far unknown, and therefore characterized by the predicate $\top$. Our argument proceeds by instantiating this unknown event with the transitions of the transition system, resulting in one new trace per transition. The *Instantiate*$^\omega$ trace transformer represents a case distinction, and we will need to discharge all cases.

For example, transition a_1 of Producer 1, gives us the trace of node 2. A consequence of the decision that a_1 occurs infinitely often is that a_4 must also

occur infinitely often: after the execution of a_1, the program counter of producer 1 equals 2, and the precondition for the execution of a_1 is that it is equal to 1. The only transition that can achieve that goal is a_4 (here we oversimplify to make the presentation clearer; in the algorithm we derive the necessity of event a_4 by the interpolation-based local safety analysis). The requirement that both a_1 and a_4 occur infinitely often is expressed as the trace of node 3, obtained from the trace of node 1 by the *NecessaryCycleEvent* trace transformer. The trace of node 3 is terminating: p_1 is decreased infinitely often and is bounded from below; it is therefore a ranking function. The only remaining situation in which an infinite run might exist is if some transition increases p_1, i.e., that satisfies the predicate $p_1' > p_1$, is executed infinitely often. This situation is expressed by the trace of node 4, obtained by the application of the *InvarianceSplit* trace transformer. Since there is no transition in the program transition relation that satisfies $p_1' > p_1$, we arrive at a contradiction.

Let us explore another instantiation of the unknown event in the trace of node 1, this time with transition c_1 of Consumer 1: we obtain the trace of node 5. Again, exploring causal consequences, local safety analysis gives us that events c_2, c_3, and c_4 should also occur infinitely often in the trace: we insert them, and get the trace of node 6. Termination analysis for that trace gives us the ranking function q_1: it is bounded from below by event c_1 and decreased by event c_4. Again, we conclude that the event increasing q_1 should occur infinitely often, and introduce it in the trace of node 7. Next, we try all possible instantiations of the event characterized by the predicate $(q_1' > q_1)$: there are two transitions that satisfy the predicate, namely a_2 and b_2. We explore the instantiation with a_2 in the trace of node 8; for b_2, the reasoning proceeds similarly. The local safety analysis allows us to conclude that, besides a_2, transitions a_1 and a_4 should occur infinitely often (node 9). At this point, we realize that the trace of node 9 contains as a subgraph the trace of node 2, namely the transition a_1. We can conclude, without repeating the analysis done of nodes 2–4, that there is no infinite run corresponding to the trace of node 9.

The other possible instantiations for the unknown event in the trace of node 1 are analyzed similarly. The resulting tableau for the case of two producers and two consumers will have the shape shown in the bottom left part of Figure 6.12. It can be interpreted as a *partially ordered ranking function*: arrows represent dependencies between individual components of a ranking function. This particular ranking function shows that all threads satisfying the function components p_1 and p_2 terminate unconditionally, while the threads that satisfy the function components q_1 and q_2 terminate under the condition that both of the previous components terminate. Notice also that the tableau is of quadratic size with respect to the number of threads.

Chapter 7

Experimental Evaluation

We have instantiated the framework of causality-based verification for the area of termination analysis of concurrent programs. While termination analysis of sequential programs is a mature discipline, termination analysis in a concurrent setting is much more challenging, and before our work [44] there was no termination prover able to handle even a multi-threaded program consisting of only two modestly complicated threads.

Our implementation is called Arctor [43] (for Abstraction Refinement of Concurrent Temporal Orderings). The implementation is written in Haskell, and can currently handle multi-threaded programs with arbitrary control flow, finite data variables, and unbounded counters. We have evaluated Arctor on a number of multi-threaded benchmarks, and compared it to the state-of-the-art termination provers Terminator, T2 and AProVE. All experiments were performed on an Intel Core i7 CPU running at 2.7 GHz. As Terminator and T2 are available only as a 32-bit Windows executable with a memory limited to 2 GB, the same memory limit was used for the other tools. The benchmarks are explained below, and are available for download from the tool homepage [43].

Simple Benchmarks. These benchmarks are quite simple multi-threaded programs, intended to compare how well different techniques can handle concurrency. Typical thread here consists just of several instructions. Table 7.1 shows the performance of the termination provers on the benchmarks, described below.

Chain. The *Chain* benchmark consists of a chain of n threads, where each thread decreases its own counter x_i, but the next thread in the chain can counteract, and increase the counter of the previous thread. Only the last thread in the chain terminates unconditionally.

Phase. The *Phase* benchmark is similar to the Chain benchmark, except that now each thread can either increase or decrease its counter x_i. Each such *phase change* is, however, guarded by the next thread in the chain, which limits the number of times the phase change can occur.

Producer-Consumer. The *Producer-Consumer* benchmark is a simplified model of the *Map-Reduce* architecture from distributed processing: producers model the mapping step for separate data sources, consumers model

Tool	Terminator		T2		AProVE		Arctor		
Benchmark / Threads	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.	Nodes
Chain 2	0.65	20	0.52	20	1.58	131	0.002	2.0	3
Chain 4	1.45	25	0.54	22	2.13	153	0.002	2.2	7
Chain 6	24.4	57	0.58	24	2.58	171	0.002	2.5	11
Chain 8	×	MO	0.63	26	3.48	210	0.002	2.5	15
Chain 20	×	MO	2.36	55	16.5	941	0.007	2.5	39
Chain 40	×	MO	40.5	288	536	1237	0.023	2.8	79
Chain 60	×	MO	Z3-TO	×	×	MO	0.063	3.0	119
Chain 80	×	MO	Z3-TO	×	×	MO	0.145	3.3	159
Chain 100	×	MO	Z3-TO	×	×	MO	0.320	3.9	199
Phase 1	×	MO	U(4.53)	48	1.60	132	0.002	2.4	2
Phase 2	×	MO	U(4.53)	48	2.16	144	0.002	2.4	11
Phase 3	×	MO	U(30.6)	301	3.83	199	0.002	2.5	20
Phase 4	×	MO	×	MO	8.89	336	0.003	2.6	29
Phase 8	×	MO	×	MO	47.0	1506	0.003	2.6	65
Phase 10	×	MO	×	MO	×	MO	0.012	2.7	83
Phase 20	×	MO	×	MO	×	MO	0.061	3.3	173
Phase 40	×	MO	×	MO	×	MO	0.35	4.0	353
Phase 60	×	MO	×	MO	×	MO	1.18	4.2	533
Phase 80	×	MO	×	MO	×	MO	3.21	5.1	713
Phase 100	×	MO	×	MO	×	MO	7.38	6.1	893
Producer 1	3.37	26	2.42	38	3.17	237	0.002	2.3	6
Producer 2	1397	1394	3.25	44	6.79	523	0.002	2.6	11
Producer 3	×	MO	U(29.2)	253	U(26.6)	1439	0.002	2.6	21
Producer 4	×	MO	U(36.6)	316	U(71.2)	1455	0.003	2.7	30
Producer 5	×	MO	U(30.7)	400	U(312)	1536	0.007	2.7	44
Producer 10	×	MO	Z3-TO	×	×	MO	0.027	3.0	135
Producer 20	×	MO	Z3-TO	×	×	MO	0.30	4.2	470
Producer 40	×	MO	Z3-TO	×	×	MO	4.30	12.7	1740
Producer 60	×	MO	Z3-TO	×	×	MO	20.8	35	3810
Producer 80	×	MO	Z3-TO	×	×	MO	67.7	145	6680
Producer 100	×	MO	Z3-TO	×	×	MO	172	231	10350
Semaphore 1	3.05	26	2.81	46	3.22	230	0.002	2.6	8
Semaphore 2	622	691	U(20.7)	219	U(6.52)	465	0.002	2.6	16
Semaphore 3	×	MO	U(15.8)	239	U(10.42)	1138	0.003	2.6	24
Semaphore 10	×	MO	U(83.5)	470	U(246)	1287	0.023	2.8	80
Semaphore 20	×	MO	×	MO	×	MO	0.073	3.3	160
Semaphore 40	×	MO	×	MO	×	MO	0.264	4.0	320
Semaphore 60	×	MO	×	MO	×	MO	0.58	4.0	480
Semaphore 80	×	MO	×	MO	×	MO	1.02	4.6	640
Semaphore 100	×	MO	×	MO	×	MO	1.59	5.1	800

Table 7.1: Experimental evaluation for the set of simple multi-threaded benchmarks. Execution time is measured in seconds, and memory in megabytes. MO stands for memout; the time spent until memout was in all cases more than 1 hour. U indicates that the termination prover returned "unknown"; Z3-TO indicates a timeout in the Z3 SMT solver.

the reducing step for different types of input data. The natural requirement for such an architecture is that the distributed processing terminates for any finite amount of input data.

Semaphore. The *Semaphore* benchmark represents a model of a concurrent system where access to a critical resource is guarded by semaphores. We verify *individual accessibility* for a particular thread (i.e., the system without the thread should terminate) under the assumption of a *fair scheduler*. Since other tools do not support fairness, we have eliminated the fairness assumption for all tools using the transformation from [59], which enriches each `wait` statement with a decreasing and bounded counter.

In the *Chain* benchmark, the shape of the system allows for an easy construction of the lexicographic ranking function $\langle x_n, \ldots, x_2, x_1 \rangle$; indeed T2, AProVE, and Arctor find and validate this termination argument quickly for a large number of threads. Terminator, on the other hand, needs exponential time. The *Phase* benchmark is especially difficult for the CEGAR-based tools except Arctor: Terminator and T2 cannot verify the system even for a single thread. Terminator runs out of memory, while T2 cannot find a lexicographic ranking function, as no such function exists. AProVE is able to verify the system up to 8 threads, but appears to consume exponential time and memory. In the *Producer-Consumer* and the *Semaphore* benchmarks, Arctor scales well, while all other tools can handle at most two threads. In general, Arctor verifies all benchmarks efficiently, requiring little time and memory to handle even 100 threads.

Models of Industrial Programs. These benchmarks represent abstractions of real-life industrial multi-threaded programs. Typical thread here consists of dozens of instructions. In our experiments we have found out that only Arctor can handle these benchmarks. All other termination provers reached either a memout, or an internal timeout, or returned "termination unknown" already for two threads. Therefore, in Table 7.2 we show the experimental results only for Arctor, together with the detailed information about the benchmark parameters.

CUDA. The *CUDA* benchmark represents an abstraction of the parallel computation of binomial option pricing model on NVidia GPUs [3].

Make. The *Make* benchmark models a parallel execution of the make program (achieved by the `make -j N` command), doing a compilation of some program. The model accounts for dependencies between targets and for the workload restrictions.

Map-Reduce. The *Map-Reduce* benchmark is a model of Google's implementation of the Map-Reduce framework for its *App Engine* distributed computation platform [1].

Comparison of process parameters, such as the number of transitions and instructions per thread, shows that Arctor is rather insensitive to them. On the other hand, the behavior of Arctor on the last benchmark demonstrates its current limitation: it does not handle very well processes with a high branching degree, due to the resulting combinatorial explosion in the number of traces.

Benchmark / Threads	Avg. trans.	Avg. instr.	Time	Memory	Nodes
CUDA 2			0.04	3.3	86
CUDA 3			0.09	3.7	129
CUDA 4			0.15	4.3	172
CUDA 5			0.24	4.5	215
CUDA 6	22	18	0.33	4.5	258
CUDA 7			0.45	4.6	301
CUDA 8			0.58	5.5	344
CUDA 9			0.72	5.5	387
CUDA 10			0.88	5.5	430
Make 2			0.04	3.6	126
Make 3			0.10	4.3	189
Make 4			0.17	4.5	252
Make 5			0.26	4.5	315
Make 6	30	54	0.36	4.5	378
Make 7			0.48	4.5	441
Make 8			0.62	4.6	504
Make 9			0.79	5.5	567
Make 10			0.97	5.5	630
Map-Reduce 2			0.42	4.5	238
Map-Reduce 3			2.50	4.5	393
Map-Reduce 4			8.22	5.5	547
Map-Reduce 5			31.3	6.5	767
Map-Reduce 6	10	13	78.7	6.5	986
Map-Reduce 7			219	7.3	1271
Map-Reduce 8			457	8.3	1555
Map-Reduce 9			1053	9.3	1905
Map-Reduce 10			1924	11.4	2254

Table 7.2: Experimental evaluation for the set of industrial multi-threaded benchmarks. Execution time is measured in seconds, and memory in megabytes. The second and the third columns represent the average number of transitions and instructions per thread (one transition may comprise several instructions).

Chapter 8

Conclusion and Future Work

In this thesis we have presented our solution to the verification problem for concurrent programs with infinite data. The solution we propose heavily rests on the new concurrency model, called concurrent traces. For this model, we have fully described its syntactic and language-based properties. We have also characterized the complexity of such operations as emptiness checking and language inclusion. We have identified two restrictions, namely transitivity and determinism, which help to reduce the complexity to such complexity classes as **NP** or **GI**, which are very well suitable for automation.

Building on concurrent traces as a foundation, we have developed proof systems for safety and liveness properties. We have demonstrated that for practically relevant classes of programs, such as multi-threaded programs with binary semaphores, the constructed proofs are of polynomial size, and can be also checked in polynomial time. The methods of the thesis have been implemented in Arctor, the first scalable termination prover for concurrent programs, which is able to handle programs with hundreds of non-trivial threads.

Though safety and liveness verification is very important, and, in principle sufficient (because every temporal property can be represented as the intersection of a safety property and a liveness property), it is tempting to extend the concurrent trace model and the proof system to full LTL properties. Here we briefly outline how this extension could be done. First, concurrent traces need to be extended to allow labeling of events with LTL formulas. A violation of a temporal property φ could be then described by a concurrent trace where the entry event is labeled with $\neg\varphi$. Second, specialized proof rules need to be developed, which would operate directly on the LTL formulas that label events. E.g., if an event in a trace is labeled with the formula $\Diamond\varphi$, then the trace could be transformed into another one, where a new event, labeled with φ, is introduced after the considered event. The main complication here is that the new proof rules need to account for induction, because the interaction of $\Box\varphi$ constraints with $\Diamond\varphi$ and $\bigcirc\varphi$ constraints is able to encode induction. Finally, the tableau algorithm for liveness needs to be extended to an algorithm which operates on LTL-labeled concurrent traces. We leave the development of these preliminary ideas for future work.

Causality-based verification can be extended with specialized proof rules and/or algorithms, which would account for such aspects as *counting* and *symmetry*: this will be useful, for example, in the realm of distributed protocols and parametrized systems. There is also a wide spectrum of application areas for which causality-based verification could be specialized. We can imagine specialized proof rules for such primitives as message-passing, different forms of synchronization, voting, etc.

Finally, we believe, the concurrent trace model can find a much wider area of application than what is described in the thesis. For example, it could be used to formalize and mechanically prove many of the hand-written proofs of concurrent and distributed protocols. It could also be used as a specification method, because it inherently allows to specify concurrent behaviors using a graphical representation; this can be an advantage compared to such textual specification languages as LTL or PSL. Alternatively, used as an output of a verification tool, it can produce a much more understandable description of a counterexample than a linear sequence of states, because it contains only the necessary events, and highlights the relationships between them.

Bibliography

[1] AppEngine: Google's implementation of the Map-Reduce framework. `http://docs.nvidia.com/cuda/samples/4_Finance/binomialOptions/doc/binomialOptions.pdf`. Accessed: 2016-03-21.

[2] International Satisfiability Modulo Theories Competition (SMT-COMP). `http://smtcomp.org`.

[3] Parallel computation of binomial option pricing model on NVidia GPUs. `http://docs.nvidia.com/cuda/samples/4_Finance/binomialOptions/doc/binomialOptions.pdf`. Accessed: 2016-03-21.

[4] L. Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[5] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[6] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In M. Burke and M. L. Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213. ACM, 2001.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[8] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16(3):227–270, 2000.

[9] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987.

[10] M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 413–429. Springer, 2013.

[11] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In E. Ábrahám

and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[12] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In F. Arbab and M. Sirjani, editors, *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings*, volume 4767 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2007.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 428–439. IEEE Computer Society, 1990.

[14] E. M. Clarke. The birth of model checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[15] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[16] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[17] M. Colón and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001.

[18] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.

[19] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41(6):415–426, June 2006.

[20] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.

[21] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In Rozenberg [68], pages 163–246.

[22] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[23] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[24] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *J. Comput. Syst. Sci.*, 72(4):547–575, 2006.

[25] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 271–274. Springer, 2010.

[26] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In Rozenberg [68], pages 247–312.

[27] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[28] J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[29] A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 129–142. ACM, 2013.

[30] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[31] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with aprove. In V. van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[32] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[33] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[34] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In J. Palsberg and Z. Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2009.

[35] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70. ACM, 2002.

[36] T. A. Henzinger, R. Majumdar, and J. Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Log.*, 6(1):1–32, 2005.

[37] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[38] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[39] G. J. Holzmann. Pan: A protocol specification analyzer. 1981.

[40] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[41] R. E. Johnson and F. B. Schneider. Symmetry and similarity in distributed systems. In M. A. Malcolm and H. R. Strong, editors, *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pages 13–22. ACM, 1985.

[42] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[43] A. Kupriyanov. Implementation of the Arctor termination prover. `http://www.react.uni-saarland.de/tools/arctor/`.

[44] A. Kupriyanov and B. Finkbeiner. Causal termination of multi-threaded programs. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 814–830. Springer, 2014.

[45] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[46] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[47] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[48] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In S. J. Eggers and J. R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008.

[49] A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2010.

[50] Z. Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974.

[51] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program.*, 4(3):257–289, 1984.

[52] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[53] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[54] A. W. Mazurkiewicz. Trace theory. In Brauer et al. [9], pages 279–324.

[55] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[56] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[57] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In G. Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 398–418. Springer, 2011.

[58] M. L. Minsky. Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines. *The Annals of Mathematics*, 74(3):437–455, Nov. 1961.

[59] E.-R. Olderog and K. R. Apt. Fairness in parallel programs: The transformational approach. *ACM Trans. Program. Lang. Syst.*, 10(3):420–455, 1988.

[60] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[61] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[62] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[63] D. A. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[64] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[65] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[66] A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, July 2004.

[67] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In M. Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007.*, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2007.

[68] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[69] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[70] A. Turing. Checking a large routine. In M. Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[71] C. Urban and A. Miné. Proving guarantee and recurrence temporal properties by abstract interpretation. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 190–208. Springer, 2015.

[72] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[73] G. Winskel. Event structures. In Brauer et al. [9], pages 325–392.