CAUSALITY-BASED MODEL CHECKING FOR REAL-TIME SYSTEMS

BACHELOR'S THESIS

JULIAN SIBER

UNIVERSITÄT
DES
SAARLANDES

Faculty of Mathematics and Computer Science
Department of Computer Science

Saarbrücken, 26 February 2019

ADVISOR:
Prof. Bernd Finkbeiner, PhD.
Universität des Saarlandes
Saarbrücken, Germany

REVIEWERS:
Prof. Bernd Finkbeiner, PhD.
Universität des Saarlandes
Saarbrücken, Germany

Dr. Andrey Kupriyanov
Barracuda Networks AG
Vienna, Austria

SUBMITTED:
26 February 2019

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

STATEMENT IN LIEU OF AN OATH

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

EINVERSTÄNDNISERKLÄRUNG

Ich bin damit einverstanden, dass die (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

DECLARATION OF CONSENT

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 26 February 2019

Julian Siber

ABSTRACT

For many of today's digital systems, correctness not only depends on the exact result that is computed, but also on whether the computation finishes in time. These real-time systems often fill safety-critical roles and interact with their physical environment, circumstances that call for rigorous verification of their correctness before deployment.

A common technique for verifying whether a given system satisfies a property is model checking. With timed automata as a formal model, there exist multiple highly-optimized tools that facilitate specification and verification of real-time systems. However, even state-of-the-art model checkers struggle with the complexity introduced by concurrency. This is due to the large number of states present in a network of timed automata.

For discrete systems, a promising technique for efficient model checking in concurrent settings is causality-based verification. Unlike conventional model checking methods, the causality-based approach does not rely on a comprehensive state space traversal. Instead, it analyzes the cause-effect relationships of events leading to a hypothetical error, aiming to prove the error's non-existence by contradiction. The analysis is driven by domain-specific proof rules that facilitate inference of the causal dependencies between events. These rules potentially allow for considerable shortcuts, in this way mitigating the state space explosion inherent to the model checking problem.

In this thesis, we extend causality-based verification to networks of timed automata. We apply the causality-based concurrency model of concurrent traces to a real-time setting and provide the necessary operations for model checking. Further, we define a number of appropriate proof rules that capture the cause-effect relationships in timed automata and provide an algorithm for constructing proofs of safety properties. We conclude by demonstrating that our algorithm proves the safety of Fischer's protocol for mutual exclusion in polynomial space and time.

ACKNOWLEDGEMENTS

First of all, I would like to express my deep gratitude to my advisor Prof. Bernd Finkbeiner, not only for offering me a topic very much in accord with my academic interests but also for the countless, insightful discussions on the intricacies of time and his guidance throughout this thesis.

Moreover, many thanks go to Dr. Andrey Kupriyanov for taking the time to answer my questions on the topic of causality and reviewing this thesis.

Lastly, I would like to take this opportunity to thank my family and friends for their unconditional support, without which undoubtedly neither this thesis nor my studies would have been possible.

CONTENTS

1 INTRODUCTION

Cyber-physical systems can be found in a multitude of applications ranging from the smartphones in our pockets to aircraft control in the sky. In many, safety-critical instances, failures threaten particularly grave consequences for their users, because these systems interact with their environment physically. Unfortunately, bugs are hard to spot and the highest level of assurance is key. In fact, it is safe to say a guarantee of correctness is desired in most cases. After all, who would not prefer a controller for the nearby nuclear power plant to be designed without errors?

A solution to guarantee the correctness of digital systems is formal verification. The intended purpose of these methods is to formally prove that a system satisfies all necessary properties before deployment. One such method, model checking, automatically verifies whether the implementation of a system, given as a model in form of a state-transition-graph, satisfies some specification that expresses the desired behavior. If the model is correct, a proof is constructed. If it is not, the model checking algorithm provides a counterexample violating the property. The potential value of model checking for the reliability of system design cannot be overstated. However, there are multiple aspects that make the model checking problem very challenging for real-world systems.

Firstly, correctness is often a question of the right timing. It is not enough that the desired result is produced eventually, it has to be produced in time, with respect to other, environmental or computational events. For instance, the gate of a train crossing should fully descend in a predefined time interval after initiation by an approaching train. Systems that have to meet such quantified real-time constraints are called real-time systems. A popular formalism for specification of these time-dependent systems are timed automata [1], finite state machines enriched with real-valued clock variables used to track time. Model checking timed automata is a mature discipline, starting from the first decidability results in the early '90s based on the region abstraction [2] and culminating in sophisticated tools like Uppaal [4] utilizing more efficient abstractions and data structures. At their core, most state-of-the-art model checkers for real-time systems rely on a comprehensive state space traversal, whereby the infinite state space of a timed automaton is abstracted to a finite symbolic representation.

However, the applicability of even highly-optimized tools such as Uppaal is diminished by another common aspect of real-world systems: concurrency. Most practically relevant systems are complex net-

works of multiple participating processes. This poses a problem for conventional model checking methods, as the state space scales exponentially with the number of processes.

Recently, causality-based verification [10] has been proposed by Andrey Kupriyanov as a model checking framework for discrete systems that allows for considerable savings in concurrent settings. For certain, practically relevant classes of programs such as multi-threaded programs with binary semaphores, the heavy toll resulting from state space explosion can be avoided. In fact, the causality-based model checking algorithm reduces the complexity of verifying this class of programs from exponential to polynomial [11]. The innovation of this approach stems from the focus on causality, which is the dependency relation between two events, where the first (the cause) is partly responsible for the second (the effect). This line of reasoning is usually more common to manual proofs, where an exhaustive search of all possibilities is not feasible. Instead, one infers from a given situation (the effect) all possible explanations (the causes).

The causality-based model checking framework captures this line of reasoning by replacing system states as the atomic proof object by concurrent traces, which are abstractions of (sets of) system computations. They describe computational scenarios, and it is possible to derive for a given scenario that describes some effect implicitly, such as a change of system states, the necessary existence of an event causing the effect. This process is used to refine the abstraction and formalized with so-called trace transformers. When starting from scenarios that describe all possible errors in a system, this process of refinement either produces a causal chain of computational events leading from start to error or infers that no such violation is possible.

When applying causality-based verification to other domains, it is, however, necessary to recognize domain-specific dependencies for inference of causal dependencies. In the case of timed automata, this mainly concerns the real-valued clock variables. The main catch is that these variables are implicitly synchronized, that is to say, all clocks in a network of timed automata advance at the same speed. In timed systems, this synchronization is often the main force of correctness. This circumstance makes them a particularly suited target for causal analysis, because human reasoning, the basis for causality-

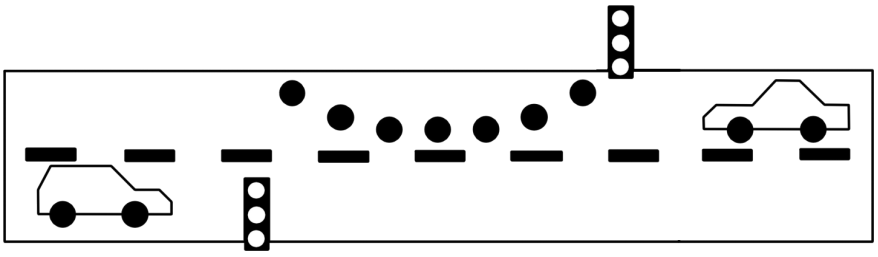


Figure 1.1: Roadworks with traffic lights.

based analysis, naturally assumes time to be a monolithic factor. Consider for instance roadworks on one lane of a two-way street, secured by traffic lights, as depicted in Figure 1.1. Let us assume that between the green phase of any traffic light and the green phase of the other, there is a delay of ten seconds. On the other hand, without outside interference, a car will never need more than seven seconds to fully pass the roadworks. Intuitively, it is quite clear that in a closed system with only cars, roadworks and traffic lights, correctness is guaranteed. After all, even when a car enters just before its traffic light turns red, it will always need less time to pass than the other traffic light to turn green. If we consider a scenario with other actors, what could possibly cause an error? For two cars to both be passing the roadworks at the same time, the first car somehow has to spend more than the seven seconds in this section. One might think of children passing the street, causing the driver to use his breaks. In any case, we can infer from the hypothetical scenario of two cars passing the roadworks at the same time, the effect, the existence of some cause, that is, the first car taking more than seven seconds.

In this thesis, we will capture this line of reasoning about timing consistency in a causality-based model checking approach for real-time systems modeled as networks of timed automata.

OUTLINE

We provide the necessary preliminaries on timed automata and their symbolic abstractions in Chapter 2. Further, we introduce the relevant parts of the causality-based verification framework in Chapter 3. We extend the causality-based concurrency model of concurrent traces to timed automata in Chapter 4. This also includes necessary operations we later need for model checking, such as satisfiability and emptiness checking. In Chapter 5 we define trace transformers appropriate for timed automata, and comprehensively describe our causality-based model checking algorithm for timed automata and safety properties. We conclude the Chapter by demonstrating this algorithm at the example of Fischer's protocol, where we achieve polynomial complexity for a scaling number of processes. In Chapter 6, we expand upon how our approach relates to other methods concerned with concurrency in a real-time setting. The results of this thesis and an assessment of promising avenues of future work are presented in Chapter 7.

2 TIMED AUTOMATA

Timed automata [1, 2] are finite state machines augmented with a finite set of real-valued clocks. At the start of any execution, all clocks start with value zero and subsequently advance at the same speed. Clocks can be reset on taking a transition. Timing behavior is further modeled with guards and invariants, which are labels of transitions and locations, respectively. Guards have to be satisfied by the current clock valuation on taking the transition. An invariant has to be fulfilled while staying at or entering the corresponding location.

2.1 SYNTAX AND SEMANTICS

Before formally defining timed automata, we develop some auxiliary definitions. We closely follow [9] in notation. A *clock* is a non-negative, real valued variable. The set of clocks is denoted by C. A *clock valuation* is a mapping $v^c : C \to \mathbb{R}_{\geqslant 0}$ from a set of clocks to their corresponding values. For a given clock valuation v^c and some $\delta \in \mathbb{R}_{\geqslant 0}$, $v^c + \delta$ denotes a clock valuation after the elapse of δ time units. It is defined by: $\forall x \in C : (v^c + \delta)(x) = v^c(x) + \delta$. For a set of clocks $R \subseteq C$ and a clock valuation v^c, $v^c[R]$ is a clock valuation where all clocks $x \in R$ are reset to zero, i.e.

$$\forall x \in C : v^c[R](x) = \begin{cases} 0 & \text{if } x \in R \\ v_c(x) & \text{else} \end{cases}$$

A *clock constraint* ϕ is defined by $\phi := x \bowtie n \mid \phi_1 \wedge \phi_2 \mid \mathrm{true}$, with $x \in C, n \in \mathbb{N}$ and $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$. The set of clock constraints over C is denoted by $\Phi(C)$.

In this thesis, we consider a class of timed automata that also allows timing-unrelated integer variables. These can be used in additional conditions and assignments on transitions. We denote the set of integer variables by V. An *integer valuation* $v^i : V \to \mathbb{Z}$ maps all integer variables to their respective values. An *integer constraint* ψ is defined by $\psi := x \bowtie n \mid \psi_1 \wedge \psi_2 \mid \mathrm{true}$, with $x \in V, n \in \mathbb{N}$ and $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$. The set of clock constraints over C is denoted by $\Psi(V)$. While clocks can only ever be reset to zero on taking a transition, we allow more complex assignments for integer variables. An *integer assignment* ω is given by $\omega := x := n \mid x := x + n \mid \omega_1 \wedge \omega_2 \mid \mathrm{true}$, with $x \in V, n \in \mathbb{Z}$. To avoid ambiguity, we allow an integer variable x to be present in a single conjunct only. The set of integer assignments is denoted by $\Omega(V)$. The semantics of applying an integer assignment

$\omega \in \Omega(V)$ to an integer valuation ν^i, denoted $\nu^i[\omega](x)$, is given by the following equation:

$$\forall x \in V : \nu^i[\omega](x) = \begin{cases} n & \text{if } \omega = x := n \\ \nu^i(x) + n & \text{if } \omega = x := x + n \\ \nu^i[\omega_1](x) & \text{if } \omega = \omega_1 \wedge \omega_2 \text{ and } x \text{ is defined in } \omega_1 \\ \nu^i[\omega_2](x) & \text{if } \omega = \omega_1 \wedge \omega_2 \text{ and } x \text{ is defined in } \omega_2 \\ \nu^i(x) & \text{else} \end{cases}$$

In networks of timed automata, we allow synchronization through explicit sender (a!) and receiver (a?) actions. To this end, we extend the alphabet Σ to $\Sigma_{sync} = \{\epsilon\} \cup (\Sigma \times \{!, ?\})$. We can now define timed automata as they will be used in this thesis.

DEFINITION 2.1 (Timed automaton). A *timed automaton* is a tuple $\langle L, l_0, \Sigma, C, V, I, E \rangle$, where

- L is a finite set of *locations*;
- $l_0 \in L$ is the *initial location*;
- Σ is a finite set of *actions*;
- C is a finite set of *clocks*;
- V is a finite set of *integer variables*;
- $I : L \to \Phi(C)$ is an *invariant assignment function*;
- $E \subseteq L \times \Sigma_{sync} \times \Phi(C) \times \Psi(V) \times \Omega(V) \times 2^C \times L$ is the set of *edges*.

For an edge $l_1 \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} l_2 \in E$, l_1 specifies the outgoing location, α is an action, β is the guard in form of a clock constraint, γ is an additional integer constraint that has to hold on taking the transition, ω is an integer assignment, ρ is a set of clocks to be reset and l_2 is the target location.

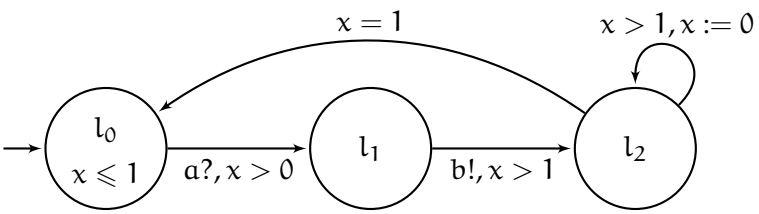


Figure 2.1: Timed automaton

EXAMPLE 2.2 (Timed automaton). We visualize timed automata with graphs as seen in Figure 2.1. Location labels and invariants can be found in the corresponding node (e.g. $x \leqslant 1$ in location l_0). Edges are labeled with their action (e.g. a), guard in form of clock (e.g. $x > 0$) and integer constraints as well as integer assignments and clock resets (e.g. $x := 0$). Differentiation is achieved through the proper variables (between clocks and integer variables) and syntax (between constraints and assignments). We omit the empty synchronization label ϵ, i.e. the transition from location l_2 to l_3 is implicitly labeled with action ϵ.

A timed automaton not only allows explicit transitions corresponding to an enabled edge, but also delay transitions which model staying in a state for a certain amount of time. We call the former *edge transitions*, the latter *delay transitions*. We follow [3] in defining the semantics of timed automata as a transition system. In the following, v_0^c denotes the clock valuation where all clock variables are mapped to zero, v_0^i denotes the similar integer valuation.

DEFINITION 2.3 (Timed automata semantics). The *semantics of a timed automaton* $A = \langle L, l_0, \Sigma, C, V, I, E\rangle$ are defined by the labeled transition system $TS = (S, s_0, \rightarrow)$, with:

- $S = L \times \mathbb{R}^C_{\geqslant 0} \times \mathbb{Z}^V$ is the set of states;
- $s_0 = (l_0, v_0^c, v_0^i) \in S$ is the initial state;
- $\rightarrow \subseteq S \times S$ contains delay ($\xrightarrow{\delta}$) and edge ($\xrightarrow{\alpha}$) transitions:
  - $(l, v^c, v^i) \xrightarrow{\delta} (l, v^c + \delta, v^i)$ iff $\forall 0 \leqslant \delta' \leqslant \delta : (v^c + \delta') \models I(l)$;
  - $(l_1, v_1^c, v_1^i) \xrightarrow{\alpha} (l_2, v_2^c, v_2^i)$ iff $\exists l_1 \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} l_2 \in E$ s.t. $v_1^c \models \beta, v_2^c = v_1^c[\rho], v_2^c \models I(l_2), v_1^i \models \gamma, v_2^i = v_1^i[\omega]$.

Timed automata read *timed words* $\xi = (a_1, t_1)(a_2, t_2)...(a_i, t_i)...$ where $t_i \leqslant t_{i+1}$ for all $i \geqslant 0$ and $a_i \in \Sigma$ is an action taken by the automaton after $t_i \in \mathbb{R}_{\geqslant 0}$ time units.

DEFINITION 2.4 (Run of a timed automaton). A *run* of a timed automaton $A = \langle L, l_0, \Sigma, C, V, I, E\rangle$ over a timed word $\xi = (a_1, t_1)(a_2, t_2)...$ $(a_i, t_i)...$ is a sequence of alternating delay and edge transitions in TS:

$$(l_0, v_0^c, v_0^i, t_0) \xrightarrow{d_1}\xrightarrow{a_1} (l_1, v_1^c, v_1^i, t_1) \xrightarrow{d_2}\xrightarrow{a_2} (l_2, v_2^c, v_2^i, t_2) \xrightarrow{d_3}\xrightarrow{a_3} ...$$

that satisfies $t_0 = 0$ and $t_i = t_{i-1} + d_i$ for all $i \geqslant 1$.

We call a sequence $\pi = s_0, ..., s_n$ with $s_i = (l_i, v_i^c, v_i^i, t_i)$ a *timed computation* if $s_0 = (l, v_0^c, v_0^i, t_0)$ with $t_0 = 0$ and for all $1 \leqslant i \leqslant n : t_i \geqslant t_{i-1}$ and $v_i^c = v_{i-1}^c + t_i - t_{i-1}[R]$ for some set of clocks R. For a timed automaton $A = \langle L, l_0, \Sigma, C, V, I, E\rangle$, a *timed automaton*

computation $\pi = s_0, ..., s_n$, with $s_0 = (l_0, v_0^c, v_0^i, t_0)$, of A is a timed computation where there exists a corresponding run r of A, so that

$$r = s_0 \xrightarrow{d_1} \xrightarrow{a_1} s_1 \xrightarrow{d_2} \xrightarrow{a_2} s_2 \xrightarrow{d_3} \xrightarrow{a_3} ... \xrightarrow{d_n} \xrightarrow{a_n} s_n ...$$

Note that the notion of a timed automaton computation can be seen as equivalent to a finite path fragment in traditional literature. We further diverge in defining the language of a timed automaton A, denoted by $\mathcal{L}(A)$, as the set of timed automaton computations.

2.2 NETWORKS OF TIMED AUTOMATA

In this thesis, we are first and foremost interested in concurrent system composed of several timed automata. We call these compositions *networks of timed automata*. As stated earlier, we model synchronization with explicit sender (a!) and receiver (a?) actions. Furthermore, all integer variables in a network are shared and in this way allow for further communication.

DEFINITION 2.5 (Network of timed automata). Let $A_1, ..., A_n$ be timed automata with $A_j = \langle L^j, l_0^j, \Sigma, C^j, V, I^j, E^j \rangle$ for $1 \leqslant j \leqslant n$. Σ and V are shared and therefore equal for all n automata. We require C^j to be distinct for all timed automata. A *network of timed automata* $A_1, ...A_n$ is defined by the product automaton $A = \langle L, l_0, \Sigma, C, V, I, E \rangle$, with:

- $L = L^1 \times ... \times L^n$;
- $l_0 = (l_0^1, ..., l_0^n)$;
- $C = C^1 \cup \ ... \ \cup C^n$;
- $I(l^1, ..., l^n) = I^1(l^1) \wedge \ ... \ \wedge I^n(l^n)$;
- E is composed of:
  - $\forall i \in \{1, ..., n\} : (..., l^i, ...) \xrightarrow{\sigma,\beta,\gamma,\omega,\rho} (..., l^{i'}, ...)$ if $l^i \xrightarrow{\sigma,\beta,\gamma,\omega,\rho} l^{i'} \in E^i$;
  - $\forall i \neq j \in \{1, ..., n\} : (..., l^i, ..., l^j, ...) \xrightarrow{\epsilon,\beta,\gamma,\omega,\rho} (..., l^{i'}, ..., l^{j'}, ...)$ if $l^i \xrightarrow{a!,\beta_1,\gamma_1,\omega_1,\rho_1} l^{i'} \in E^i$ and $l^j \xrightarrow{a?,\beta_2,\gamma_2,\omega_2,\rho_2} l^{j'} \in E^j$, with $\beta = \beta_1 \wedge \beta_2, \gamma = \gamma_1 \wedge \gamma_2, \omega = \omega_1 \wedge \omega_2, \rho = \rho_1 \cup \rho_2$.

2.3 FINITE ABSTRACTIONS

The semantics of timed automata as seen in Definition 2.3 describe an infinite transition system, due to the real-valued clock valuations that produce an infinite state space. Early model checking proposals were mainly concerned with decidability. To this end, Alur and Dill [2] developed a finite abstraction of timed automata based on *regions*.

The region abstraction utilizes the observation, that the behavior of a timed automaton does not depend on the exact clock values, e.g. for the timed automaton in Figure 2.1 and location l_1, it does not matter whether clock x equals 0.3 or 0.9 - both clock valuations do not allow taking the transition to l_2. This notion is formalized in the following definition, where for a clock $x \in C$, $\lfloor \nu(x) \rfloor$ and a clock valuation $\nu \in C \to \mathbb{R}_{\geqslant 0}$, denotes the integral part of the clock value, $\mathrm{fract}(\nu(x))$ the fractal part, and c_x is the largest constant x gets compared to.

DEFINITION 2.6 (Region). For a timed automaton $\langle L, l_0, \Sigma, C, V, I, E \rangle$, two clock valuations $\nu, \nu' \in C \to \mathbb{R}_{\geqslant 0}$, are in the same *region*, also denoted by $\nu \cong \nu'$, iff the following conditions are met:

- for each clock $x \in C$, either $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$ or both $\nu(x)$ and $\nu'(x)$ are greater than c_x;
- for each pair of clocks $x, y \in C$ and $\nu(x) \leqslant c_x$, $\nu(y) \leqslant c_y$:
  1. $\mathrm{fract}(\nu(x)) \leqslant \mathrm{fract}(\nu(y))$ iff $\mathrm{fract}(\nu'(x)) \leqslant \mathrm{fract}(\nu'(y))$, and
  2. $\mathrm{fract}(\nu(x)) = 0$ iff $\mathrm{fract}(\nu'(x)) = 0$.

Extending the equivalence relation $\cong$ to composite states so that $(s, \nu) \cong (s', \nu')$ iff $s = s'$ and $\nu \cong \nu'$ yields the quotient $[TA]_{\cong}$, called the *region graph* of a timed automaton TA.

The number of regions is bounded by the largest constant in the automaton and therefore finite. However, due to its fine granularity, the size of the region graph scales exponentially with the number of clocks and constants. Therefore, while the region abstraction proves he decidability of timed automata model checking, it is generally unsuited for practical purposes.

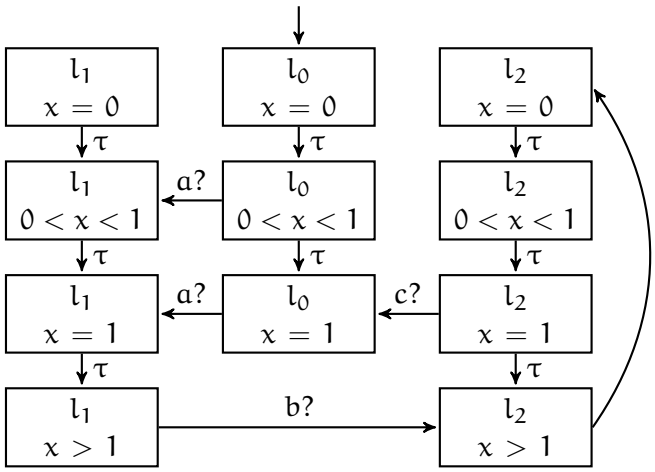


Figure 2.2: Region graph of timed automaton seen in Figure 2.1

A more efficient abstraction of the state space are *zones* [8], which provide the basis of most state-of-the-art verification tools for timed automata.

DEFINITION 2.7 (Zone). A *zone* D is the maximal set of clock assignments satisfying its clock constraint of the form

$$D ::= x \sim c \mid c \sim x \mid x - y \sim c \mid D \wedge D$$

with $x, y \in C, c \in \mathbb{N}$ and $\sim \in \{<, \leqslant\}$.

The zone graph for the timed automaton from Figure 2.1 can be seen in Figure 2.3.

But not only do zones provide a coarser symbolic representations, but the following operations for zones D, D' are also zone preserving and facilitate analysis of timed automata:

- Future of D: $\overrightarrow{D} = \{v + t \mid v \in D \text{ and } t \in \mathbb{R}_{\geqslant 0}\}$;
- Past of D: $\overleftarrow{D} = \{v - t \mid v \in D \text{ and } t \in \mathbb{R}_{\geqslant 0}\}$;
- Intersection of D and D': $D \cap D' = \{v \mid v \in D \text{ and } v \in D'\}$;
- Reset to zero of D with respect to set of clocks R: $[R \leftarrow 0]D = \{v[R] \mid v \in D\}$;
- Inverse reset to zero of D with respect to set of clocks R: $[R \leftarrow 0]^{-1}D = \{v \mid [R \leftarrow 0]v \in D\}$.

With the help of these elementary operations, it is possible to express for some timed automaton $A = \langle L, l_0, \Sigma, C, V, I, E\rangle$ and an edge $l_1 \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} l_2 \in E$ the effect taking this edge has on zone D. With $[\![\beta]\!] = \{v \in C \to \mathbb{R}_{\geqslant 0} \mid v \models \beta\}$, $[\![\rho_0]\!] = \{v \in C \to \mathbb{R}_{\geqslant 0} \mid v(x) = 0 \ \forall x \in \rho\}$:

- $\mathrm{Post}_e(D) = [\rho \leftarrow 0](\overrightarrow{D} \cap \{v \in C \to \mathbb{R}_{\geqslant 0} \mid v \models \beta\})$;
- $\mathrm{Pre}_e(D) = [\rho \leftarrow 0]^{-1}(\overrightarrow{D} \cap [\![\rho_0]\!]) \cap [\![\beta]\!]$.

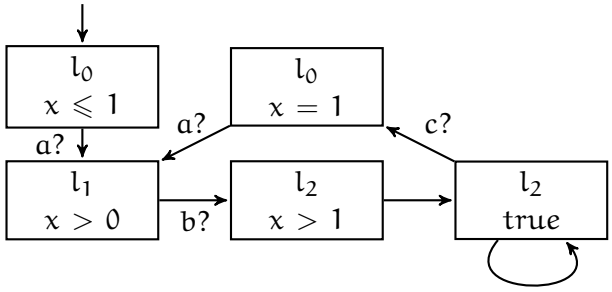


Figure 2.3: Zone graph of timed automaton seen in Figure 2.1

3

CAUSALITY-BASED VERIFICATION

The causality-based verification proposed by Andrey Kupriyanov [10] fundamentally changes the proof object under consideration. Instead of comprehensively traversing the state space to conclude whether an error state is reachable, it is based on the question: *What events have to happen necessarily for the system to go from the initial state to some error state? And is it possible to arrange them in a way that is not contradictory?* To be more exact, the model checking algorithm works on partial representations of executions called *concurrent traces*, which get refined until it is clear whether or not they correspond to actual system computations.

In this chapter, if not specified further, definitions and notations are presented as proposed by Andrey Kupriyanov in [10]. To demonstrate causality-based verification of safety properties for discrete-time, we will use the following definition of transition systems as a model to describe concurrent systems with interleaving semantics. In this chapter we denote by $\phi(\mathcal{V})$ the set of first-order formulas over a set of variables $\mathcal{V}$. For a variable $x \in \mathcal{V}$, we define a *primed variable* x', which describes the value of x in the next state. Further, we call formulas from the set $\phi(\mathcal{V})$ *state predicates* and from the set $\mathcal{V} \cup \mathcal{V}'$ *transition predicates*, respectively.

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

- $\mathcal{V}$ is a finite set of system variables;
- $\mathcal{T} \subseteq \mathcal{V} \cup \mathcal{V}'$ is a finite set of system transitions;
- $\Theta \in \phi(\mathcal{V}')$ defines the systems starting states.

A *state* of $\mathcal{S}$ is a valuation of the system variables $\mathcal{V}$. We call a sequence of states $s_0, s_1, s_2 ...$ a *computation*. Moreover, a *system computation* of $\mathcal{S}$ is a computation, where $\Theta(s_0)$ holds, and for all $i \geqslant 1$ there is a system transition $t_i \in \mathcal{T}$ such that $t_i(s_{i-1}, s_i)$ holds. For a system $\mathcal{S}$, we denote the set of systems computations by $\mathcal{L}(\mathcal{S})$.

3.1 CONCURRENT TRACES

The smallest building block of the larger causality-based proof object is a *concurrent trace*, whose role can be seen as corresponding to that of a state in standard model checking.

DEFINITION 3.2 (Finite concurrent trace). A *finite concurrent trace* is a tuple $\mathcal{F} = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E}$ is a set of *events*;
- $\mathcal{C} \subset \mathcal{E} \times \mathcal{E}$ is a set of *causal links*;
- $\lightning \subseteq \mathcal{E} \times \mathcal{E}$ is a symmetric *conflict relation*;
- $\lambda_{\mathcal{E}} : \mathcal{E} \to \phi(\mathcal{V} \cup \mathcal{V}')$ and $\lambda_{\mathcal{C}} : \mathcal{C} \to \phi(\mathcal{V} \cup \mathcal{V}')$ are labelings of events and causal links with transition predicates.

A concurrent trace defines a set of computations that share the state changes specified by its events. Further, the causal links describe the partial order of the changes. While other state changes can occur between events, the labelings of causal links allow to formulate additional constraints on these changes. Multiple logical distinct events can be mapped to a single system transition, however, the conflict relation allows to restrict this. In the remainder of this thesis, we will often simply refer to concurrent traces as *traces*. We denote the set of all concurrent traces by $\mathbb{F}$.

DEFINITION 3.3 (Trace language). The *language* of a concurrent trace $\mathcal{F} = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$ is defined as a set $\mathcal{L}(\mathcal{F})$ of finite computations, such that for each computation $\pi = s_0, ..., s_n \in \mathcal{L}(\mathcal{F})$ there exists a mapping $\sigma : \mathcal{E} \to \{s_0, ..., s_n\}$, called a *run* of $\mathcal{F}$ on π, such that:

1. for each event $e \in \mathcal{E}$ and $s_i = \sigma(e)$ it holds that $\lambda_{\mathcal{E}}(e)(s_i, s_{i+1})$;
2. for each causal link $(e_1, e_2) \in \mathcal{C}$, and $s_i = \sigma(e_1)$ and $s_j = \sigma(e_2)$, we have:
   a) $i \leqslant j$, and
   b) for all $i < k < j$, the formula $\lambda_{\mathcal{C}}(c)(s_k, s_{k+1})$ holds.
3. for each pair of conflicting events $e_1 \lightning e_2$, with $s_1 = \sigma(e_1)$ and $s_2 = \sigma(e_2)$, we have $i \neq j$.

EXAMPLE 3.4 (Concurrent traces). Consider the two concurrent traces shown in Figure 3.1 and the transition system with:

$$
\begin{aligned}
&\mathcal{V} = \{x, y\};\\
&\mathcal{T} = \{\mathsf{x{+}{+}} : x' = x + 1 \wedge y' = y,\ \mathsf{y{+}{+}} : y' = y + 1 \wedge x' = x,\\
&\qquad \mathsf{x{-}{-}} : x' = x - 1 \wedge y' = y,\ \mathsf{y{-}{-}} : y' = y - 1 \wedge x' = x,\\
&\qquad \mathsf{x\&y{+}{+}} : x' = x + 1 \wedge y' = y + 1,\ \mathsf{x\&y{-}{-}} : x' = x - 1 \wedge y' = y - 1\};\\
&\Theta \equiv x' = 0 \wedge y' = 0.
\end{aligned}
$$

Concurrent trace $\mathcal{F}_1$ requires that variables x and y get increased somewhere in the computation. But as the two events are in conflict, the computation characterized by the transition sequence $\Theta, \mathsf{x\&y{+}{+}}, e$ is not accepted because the two events are not allowed to be mapped to the same system transition. Following this line of reasoning, Θ,

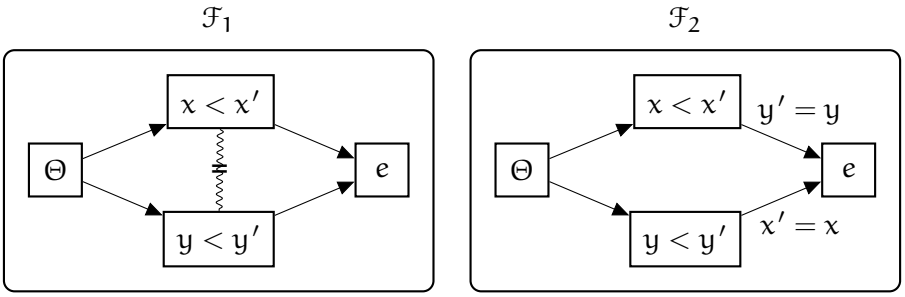


Figure 3.1: Example of two concurrent traces.

x&y++, x++, y++, e is accepted. The same sequence is, however, not accepted by $\mathcal{F}_2$. This concurrent trace requires that after some transition that increases x the value of y does not change, and vice versa. In our system this is only realizable with x&y++ as the last transition.

Concurrent traces are, in essence, directed acyclic graphs. In the following, we introduce some definitions for graph transformations in order to lift them to concurrent traces later. The notations used are borrowed from [6] [7].

A *directed graph* is a tuple $G = \langle N, E \rangle$, where N is a set of *nodes* and $E \in N \times N$ is a set of *edges*. We use $s, t : E \to N$ as functions that map each edge to its source and target, respectively.

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

A *graph production* $p : (L \xrightarrow{r} R)$ is an injective graph morphism $r : L \to R$. We call L and R the *left-hand side* and *right-hand side* of r, respectively. A production $p : (L \xrightarrow{r} R)$ can be applied to a graph G if there is a *match*, that is, an injective graph morphism $m : L \to G$. The resulting graph H is obtained by adding to G 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 that have the same image. The application of a production p to a graph G with match m is called a *direct derivation* and will be denoted by $p^m(G)$.

Concurrent traces are in general a very expressive structure which makes problems such as emptiness checking undecidable as shown in [10]. However, for developing causality-based proofs, it is possible to restrict the classes of concurrent traces under consideration far enough that the problems become tractable. Most important is the notion of a *compactization* of some concurrent trace F, which in essence is a gross underapproximation obtained by only allowing the explicit events existing in the trace. To formalize this notion, we first need some auxiliary definitions.

DEFINITION 3.5 (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 linear traces by $\mathbb{F}_L$. For any concurrent trace F', some *linearization* can be obtained by ordering all unordered events. We denote the set of linearizations of F' by $\mathtt{Linearizations}(F')$.

For a linear trace F with an ordered set of events $\mathcal{E} = \{e_1, e_2, ..., e_n\}$, we call the sequence of formulas $\phi_1, \psi_1, \phi_2, \psi_2, ..., \psi_{n-1}, \phi_n$, where $\phi_i = \lambda_{\mathcal{E}}(e_i)$ and $\psi_i = \lambda_{\mathcal{C}}((e_i, e_{i+1}))$, the *linear normal form* of F, denoted by $N_L(F)$.

DEFINITION 3.6 (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 compact traces by $\mathbb{F}_C$. For any concurrent trace F', some *contraction* can be obtained by uniting all events that are not in conflict. We denote the set of contractions of F' by $\mathtt{Contractions}(F')$.

DEFINITION 3.7 (Compactization). For a given concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, its *compactization* is any trace $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$ such that:

- for traces $F_L \in \mathtt{Linearizations}(F)$ and $F_{CL} \in \mathtt{Contractions}(F_L)$; let $\phi_1, \psi_1, \phi_2, \psi_2, ..., \psi_{n-1}, \phi_n$ be the linear normal form of F_{CL};
- $\mathcal{E}' = \mathcal{E}$ is an ordered set of events $\{e_1, e_2, ..., e_n\}$;
- $\mathcal{C}' = \{(e_i, e_{i+1}) \mid 1 \leqslant i < n\}$;
- $\not\downarrow = \{(e_i, e_j) \mid 1 \leqslant i \leqslant n, i \neq j\}$;
- $\lambda'_{\mathcal{E}} = \{e_i \to \phi_i \mid 1 \leqslant i \leqslant n\}$;
- $\lambda'_{\mathcal{C}} = \{(e_i, e_{i+1}) \to \bot \mid 1 \leqslant i < n\}$.

We call the sequence of predicates $\phi_1, \phi_2, ..., \phi_n$ a *compactization normal form* of F', denoted by $N_C(F')$. The set of all compactizations of F is given by $\mathtt{Compactizations}(F)$.

A compactization of a given concurrent trace is obtained by ordering any unordered events in some unspecified order, uniting events that are not in conflict, and disallowing any further state changes between events. This makes emptiness checking trivial, as the resulting trace leaves no room for hidden, implicit transitions. The language of a compactization (and consequently, of the original concurrent trace) is not empty, when the SSA-shaped conjunction of formulas from its normal form are satisfiable. As a compactization is an underapproximation of the original trace though, unsatisfiability and emptiness of the compactization language does not necessarily mean emptiness of the origianl trace's language. It is, however, possible to identify which parts of the trace are responsible for the unsatisfiability by extracting

the unsatisfiable core of the SSA-form. In the next sections, we will focus on the refinement process of this unsatisfiable core that drives the causality-based verification algorithm.

3.2 TRACE TRANSFORMERS

Standard, state-based model checking is effectively driven by the transition relation of the system under consideration. Introducing the more powerful proof object of a concurrent trace, however, also necessitates more powerful proof rules to facilitate refinement of the concurrent trace abstraction.

This is where the concept of *causality* comes into play. The idea is to infer from a pre-existing event in a concurrent trace (the *effect*) the necessity of another (the *cause*). This notion is best described with the traces seen in Figure 3.2. Initially, $\mathcal{F}$ includes two events that do not agree on post- and precondition: On the one hand, we have an initial event ending with x having value 0. On the other hand, we have an event requiring x to be 1. This lets us infer the existence of another, necessary bridging event in between. Take note that we do not know, however, which system transition is responsible for the logical state change. The analysis therefore resorts to case distinctions in subsequent steps.

In order to formalize causal inference on concurrent traces, we extend the graph transformations seen earlier in this chapter.

DEFINITION 3.8 (Trace production). A *trace morphism* behaves similarly to a graph morphism and maps events and links to their counterparts in a different trace while preserving source and target functions. Subsequently, a *trace production* $\tau : (L \xrightarrow{r} R)$, where L and R are concurrent traces and r is a trace morphism defines a transformation of one concurrent trace into another. The structural changes are defined by the corresponding graph production, the changes in labelings are defined by the operations of boolean algebra. Essentially, a trace production is a graph production on *attributed graphs*, more details can be found in [7].

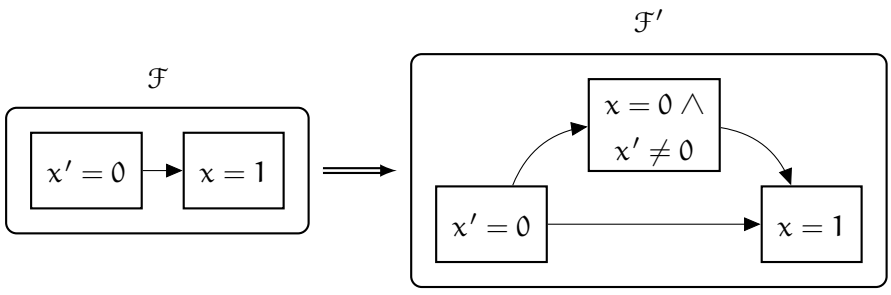


Figure 3.2: Causal inference.

DEFINITION 3.9 (Context-bounded trace production). We call a trace production $\tau : (L \xrightarrow{r} R)$ *context-bounded* if all events newly 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 preimage in L, 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 $(e_1', e'), (e', e_2') \in \mathcal{C}(R)$.

This requirement ensures that new events only get introduced between preexisting events. All trace productions we will employ later are trivially context-bounded by construction, but the property is needed for some proofs later.

DEFINITION 3.10 (Trace transformer). For a given transition system $\mathcal{S} = \langle \mathcal{V}, \mathcal{T}, \Theta \rangle$, a *trace transformer* $\tau = \{\tau_1, ..., \tau_n\}$ is an ordered set of trace productions $\tau_i : (L \xrightarrow{r_i} R)$ that share the left-hand side L. We denote L by $\mathrm{pre}(\tau)$ and call it the transformer *premise*, while we denote the set $R = \{\tau_1, ..., \tau_n\}$ by $\mathrm{post}(\tau)$ and call it transformer *conclusions*.

A trace transformer should preserve the language of concurrent traces in the context of a given system, i.e. no computations will be lost in the combined language of the transformer conclusions. However, if the conclusions provide an over-approximation of the system computations, the transformer is still useful for proof construction: if there is no error trace in the over-approximated set, the property also holds for the subset of system computations. In the same vein, an under-approximation can still be useful for refutation: if there is an error trace in a subset of the actual system computations, the property does not hold. The following definition formalizes these transformer properties.

DEFINITION 3.11 (Sound, precise, and exact trace transformer). A trace transformer τ is called *sound*, iff:

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

A trace transformer τ is called *precise*, iff:

$$\forall F \in \mathcal{F} : F \subseteq_m \mathrm{pre}(\tau) \implies \mathcal{L}(F) \cap \mathcal{L}(S) \supseteq \bigcup_{\tau_i \in \tau} \mathcal{L}(\tau_i^m(F)) \subseteq \mathcal{L}(S)$$

A trace transformer τ that is both sound and precise is called *exact*.

In the following, we list some trace transformers proposed in [10] that demonstrate the concept behind causality-based verification best. For a more detailed listing of the rules providing a foundation of our own model checking algorithm, see Appendix A.

Order Split (Figure 3.3)

The $\mathtt{OrderSplit}(a, b)$ trace transformer considers the alternate interleavings of two concurrent events a and b. Formally, we have

$\mathtt{pre}(\mathtt{OrderSplit}(a, b)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

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

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

- $R_1 = \langle \mathcal{E}, \{(a, b)\}, \lightning, \lambda_{\mathcal{E}}, \{(a, b) \to \top\} \rangle$;
- $R_2 = \langle \mathcal{E}, \{(b, a)\}, \lightning, \lambda_{\mathcal{E}}, \{(b, a) \to \top\} \rangle$.

Necessary Event (Figure 3.4)

Given two causally related and conflicting events a, b and a predicate ϕ such that the labeling of a implies ϕ' and the one of b implies $\neg\phi$, the $\mathtt{NecessaryEvent}(a, b, \phi)$ trace transformer introduces a new event in between that resolves the contradiction. Formally:

$\mathtt{pre}(\mathtt{NecessaryEvent}(a, b, \phi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\mathcal{C} = \{(a, b)\}$;
- $\lightning = \{(a, b)\}$;
- $\lambda_{\mathcal{E}} = \{a \to \phi', b \to \neg\phi\}$;
- $\lambda_{\mathcal{C}} = \{(a, b) \to \top\}$.

$\mathtt{post}(\mathtt{NecessaryEvent}(a, b, \phi)) = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, with

- $\mathcal{E}' = \{a, b, c\}$;
- $\mathcal{C}' = \{(a, b), (a, c), (c, b)\}$;

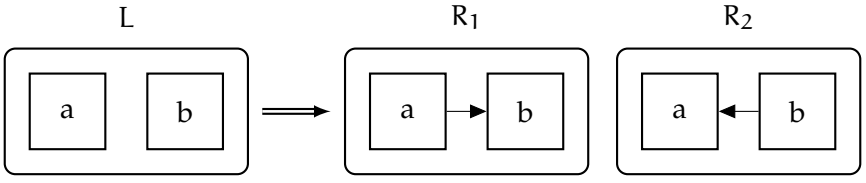


Figure 3.3: OrderSplit trace transformer.

- $\lightning' = \{(a, b), (a, c), (c, b)\}$;
- $\lambda'_{\mathcal{E}} = \lambda_{\mathcal{E}} \cup \{c \to \phi \wedge \neg\phi'\}$;
- $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \top, (c, b) \to \top\}$.

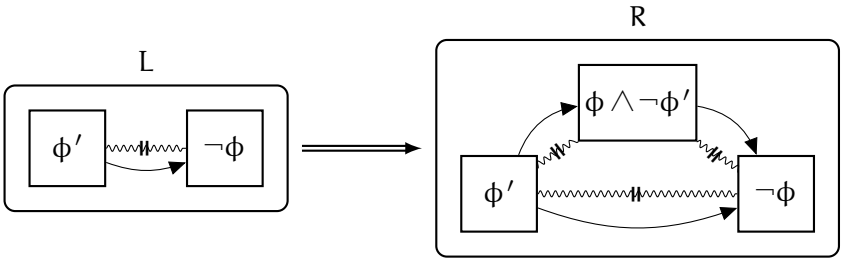


Figure 3.4: NecessaryEvent trace transformer.

First/Last Necessary Event (Figure 3.5)

$\mathtt{NecessaryEvent}(a, b, \phi)$ can be strengthened to specify that the newly introduced event c is the *first* or *last*) in the sequence of events that satisfy the predicate $\phi \wedge \neg\phi'$. This is done with the trace transformers $\mathtt{FirstNecessaryEvent}(a, b, \phi)$ and $\mathtt{LastNecessaryEvent}(a, b, \phi)$, respectively. The corresponding causal link gets labeled with predicate $\neg\phi \wedge \neg\phi'$ (last) or $\phi \wedge \phi'$ (first), in this way only events that do not change the truth value of the relevant predicate ϕ are allowed in the link. Formally:

$\mathtt{pre}(\mathtt{FirstNecessaryEvent}(a, b, \phi)) = \mathtt{pre}(\mathtt{NecessaryEvent}(a, b, \phi))$

$\mathtt{post}(\mathtt{FirstNecessaryEvent}(a, b, \phi)) = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, with

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

$\mathtt{pre}(\mathtt{LastNecessaryEvent}(a, b, \phi)) = \mathtt{pre}(\mathtt{NecessaryEvent}(a, b, \phi))$

$\mathtt{post}(\mathtt{LastNecessaryEvent}(a, b, \phi)) = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, with

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

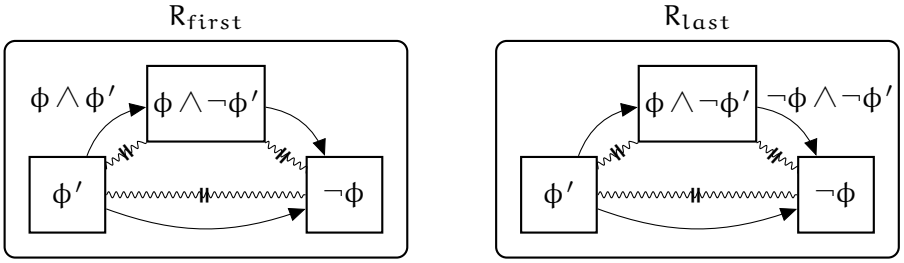


Figure 3.5: The First/Last NecessaryEvent trace transformers.

3.3 TRACE UNWINDING

Trace transformers provide a way to transition from a given concurrent trace to other, more concise traces by performing case splits. Starting from some initial abstract error traces that encode any hypothetical system violations, applying transformers repeatedly produces multiple tree like structures. This notion is formalized as a *trace unwinding*.

DEFINITION 3.12 (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* N_I and *leaves* N_L;
- $\gamma : N \to \mathcal{F}$ is a labeling of nodes with concurrent traces;
- $\delta : E \to \Pi$ is labeling of edges with trace productions. We require that for all edges with the same source n, the corresponding productions have the same left-hand side. Consequently, we have an induced labeling of internal nodes $n \in N_I$ with trace transformers: $\delta(n) = \{\delta(n, n') \mid (n, n') \in E\}$;
- μ is a labeling of internal nodes with trace morphisms: for all $n \in N_I$ we have $\mu(n) : pre(\delta(n)) \to \gamma(n)$. This function defines to which subtrace a transformer gets applied to.

DEFINITION 3.13 (Properties of trace unwinding). For a given transition system $S = \langle V, T, \Theta \rangle$, 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:

- $\gamma(n) \subseteq_{\mu(n)} pre(\delta(n))$: the trace transformer $\delta(n)$ can be applied to the concurren trace $\gamma(n)$ under the trace morphism $\mu(n)$;
- for all $(n, n') \in E$ it holds that $\delta((n, n'))^{\mu(n)}(\gamma(n)) = \gamma(n')$, i.e. the trace production of each edge (n.n') transforms trace $\gamma(n)$ into trace $\gamma(n')$.

Further, we call Υ *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).

3.4 LOOPING TRACE TABLEAU

While trace unwinding is sufficient for acyclic systems where no state space cycles occur, cycles in the system potentially allow to revisit similar configurations infinitely often. For constructing causality-based proofs, revisiting some configuration means that this particular path in the unwinding has been unfruitful for getting closer to a counterexample and the language is potentially subsumed by an earlier seen concurrent trace. To represent these properties in a concise manner, trace unwinding is lifted to a more expressive proof object, *looping trace tableau*. This allows, in essence, to stop the exploration at some node in the tree and refer back to an earlier step in the proof object, as long as the language of the former is included in the latter and repeated application of the proof steps leads to an infinitely long trace. We therefore first introduce structural trace inclusion and the notion of causal loops.

DEFINITION 3.14 (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 $\lambda'_{\mathcal{E}}(\mu_{\mathcal{E}}(e)) \implies \lambda_{\mathcal{E}}(e)$;
2. labels of the causal links in F' are stronger than those in F: for all $c \in \mathcal{C}$ we have $\lambda'_{\mathcal{C}}(\mu_{\mathcal{C}}(c)) \implies \lambda_{\mathcal{C}}(c)$;
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 trace morphism μ.

DEFINITION 3.15 (Causal path). We define a *causal path* as an ordered sequence $\tau_1, ..., \tau_k$ of trace productions $\tau_i : (L_i \xrightarrow{r_i} R_i)$ such that for all $1 \leqslant i < k$ 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 $1 \leqslant i \leqslant k$.

DEFINITION 3.16 (Causal loop). A *causal loop* is a causal path $\tau_1, ..., \tau_k$ with $\tau_i : (L_i \xrightarrow{r_i} R_i)$, where $R_k \subseteq L_1$. For any trace $F \subseteq L_1$ we define the *application* of the causal loop to the trace as a sequence $F^0_0 = F$, $F^j_i = \tau_i(F^j_{i-1})$, for $1 \leqslant i \leqslant k$ and $F^{j+1}_0 = F^j_k$.

A causal path is, therefore, a cyclic sequence of trace transformations. However, allowing these loops in a trace tableau is only sound,

as long as applying them repeatedly would yield an infinitely long trace.

DEFINITION 3.17 (Soundness of causal loops). A *causal loop* $\tau_1, ..., \tau_k$, 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 application of the causal loop:

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

where the *size* $|F|$ of a concurrent trace F is defined as the minimum number of events between all contractions of F: $|F| \triangleq min\{|\mathcal{E}(F')| \mid F' \in Contractions(F)\}$.

DEFINITION 3.18 (Looping trace tableau). For a transition system $S = \langle V, T, \Theta \rangle$, we define a *looping 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; we extend labeling μ to covered leaf nodes: for all $(n, n') \in \rightsquigarrow$ we have $\mu : \gamma(n') \to \gamma(n)$.
- $\rightsquigarrow \subset N_L \times N_I$ is a *covering relation*; for $(n, n') \in \rightsquigarrow$ we call n a *covered* node, and n' a *covering* node;

We call a looping trace tableau *correct*, if Υ is a correct trace unwinding and for all $(n, n') \in \rightsquigarrow$ we have that $\gamma(n) \subseteq_{\mu(n)} \gamma(n')$. We call a looping trace tableau *sound*, when the corresponding trace unwinding is sound and, further, all causal loops present in Γ are sound. A looping trace tableau is *complete* if all leaves that are *uncovered* contain a contradictory label.

The soundness criterion for causal loops is problematic, however. It is a global condition and we would need to check the whole tableau every time when deciding whether a node can be covered by another node. A more efficient, local criterion is *forgetful trace inclusion*.

DEFINITION 3.19 (Forgetful trace inclusion). Let the trace inclusion $F' \subseteq_\mu F$ hold for traces F, F' and 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 the trace inclusion is *left-forgetful* (resp. *right-forgetful*), if for all $e' \in \mu^\mathcal{E}$ there exists $e'_x \in \mathcal{E}' \setminus \mu^\mathcal{E}$ such that $(e'_x, e') \in \mathcal{C} \cap \lightning$ (resp. $(e', e'_x) \in \mathcal{C} \cap \lightning$). We call the trace inclusion *forgetful* if it is either left- or right-forgetful.

PROPOSITION 3.20. 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 $\Lambda = (n_1, n_2), (n_2, n_3), ..., (n_k, n_1)$, where $(n_i, n_j) \in (E \cup \rightsquigarrow)$. The loop should contain atleast 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 $1 \leqslant i \leqslant k$ 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_x, and sets $\mathcal{E}_0, \mathcal{E}_1$) : $F_0^j = \mathcal{E}_0 \uplus \mathcal{E}(\gamma(n_k))$, and $\mathcal{E}(\gamma(n_k)) = \mathcal{E}(\gamma(n_1)) \uplus \{e_x\} \uplus \mathcal{E}_1$, where event e_x is in conflict with all events from $\mathcal{E}(\gamma(n_1))$. Thus, after j iterations of the causal loop we have at least j new events which are linearly ordered and in conflict with each other (one event e_x from each iteration). Therefore, $|F_0^j| \geqslant j$, and the causal loop is sound. □

3.5 ABSTRACT TRACE TABLEAU

The last step in constructing causality-based proofs is in abstracting away unnecessary information present in the explicit trace unwinding. This is important to correctly identify situations that necessitate the same proof steps (and thus, can be covered by one another) even when a direct trace inclusion might not hold.

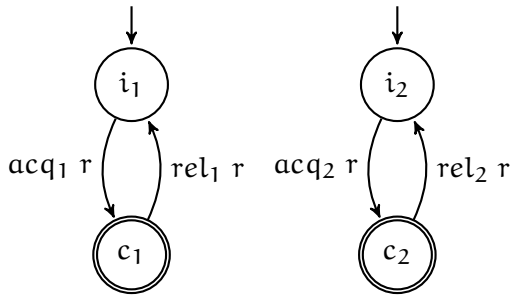
DEFINITION 3.21 (Abstract trace tableau). We define an *abstract trace tableau* as a tuple $\Lambda = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, where:

- $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$ 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*.

The label $\gamma(n)$ of the concrete trace unwinding at some node n contains all information gather by applying a chain of trace transformers defined int the path from some root to node n. The abstract label, however, only tracks the information necessary to repeat all proof steps taken in the subtree. Therefore, it has to be updated after every new proof step, by propagating the premise of the corresponding trace transformer upwards in the abstract looping trace tableau. Further, any old covering of updated nodes has to be reevaluated.

DEFINITION 3.22 (Properties of abstract trace tableau). We call an abstract trace tableau $\Lambda = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$ *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 we further have for all $n \in N$: $\gamma(n) \subseteq_{\sigma(n)} \hat{\gamma}(n)$. We call Λ *sound*, when both Υ and $\hat{\Gamma}$ are sound. We call Λ *complete*, when $\hat{\Gamma}$ is complete.

EXAMPLE 3.23. We demonstrate proofs with an abstract trace tableau for the system shown in Figure 3.6. Its a simple binary semaphore ensuring mutually exclusive access to the critical sections of two processes. The critical sections are c_1 and c_2, the error condition for this systems therefore is $e \equiv c_1 \wedge c_2$. Mutual exclusion is ensured by the



Syntax	Semantics
$acq_i\ r$	$r = 1 \wedge r' = 0$
$rel_i\ r$	$r' = 1$

Figure 3.6: Simple binary semaphore.

shared variable r, which gets set to zero by process i on taking the transition $acq_i\ r$, and set back to one on taking $rel_i\ r$. The abstract trace tableau for this system and error condition e is shown in Figure 3.7. On the left, you can see the concrete trace unwinding. On the right is the abstract looping trace tableau. Node 1 captures the initial, abstract error trace. At first, the postcondition of Θ is $i_1 \wedge i_2$, which is in conflict with the precondition of e. Two applications of trace transformer $\mathsf{LastNecessaryEvent}$ infers the existence of two necessary events that satisfy $\neg c_1 \wedge c_1'$ and $\neg c_2 \wedge c_2'$, which can only be instantiated by the system transitions $acq_1\ r$ and $acq_2\ r$ as seen in node 3. Next, a case distinction about the order of the newly introduced event is made. We omit the case where $acq_1\ r$ precedes $acq_2\ r$, as both cases work symmetrically. However, in any order the new events again do not agree on post- and precondition and $\mathsf{LastNecessaryEvent}$ gets applied. A transition that changes the value of r to one is needed in between. This can be instantiated by two system transitions, namely $rel_1\ r$ as seen in node 5 and $rel_2\ r$ as seen in node 6. Node 6 contains a contradiction, as the newly introduced $rel_2\ r$ leaves c_2 for i_2, which is not allowed by the causal link between $acq_2\ r$ and e. The concurrent trace in node 5 requires the first process to be in its critical section already, so c_1 holds for this event. Through the causal link we also infer that c_2 holds at this point. However, $c_1 \wedge c_2$ was our original error condition, so the concurrent trace in the abstract looping trace tableau of the first node can be mapped into the trace in node 5 with some $\hat{\mu}$ and the trace inclusion $\hat{\gamma}(5) \subseteq_\mu \hat{\gamma}(1)$ is (right-)forgetful. Repeated application of this causal path would lead to an infinitely long trace, so we can cover node 5 with node 1 and stop the exploration at this point.

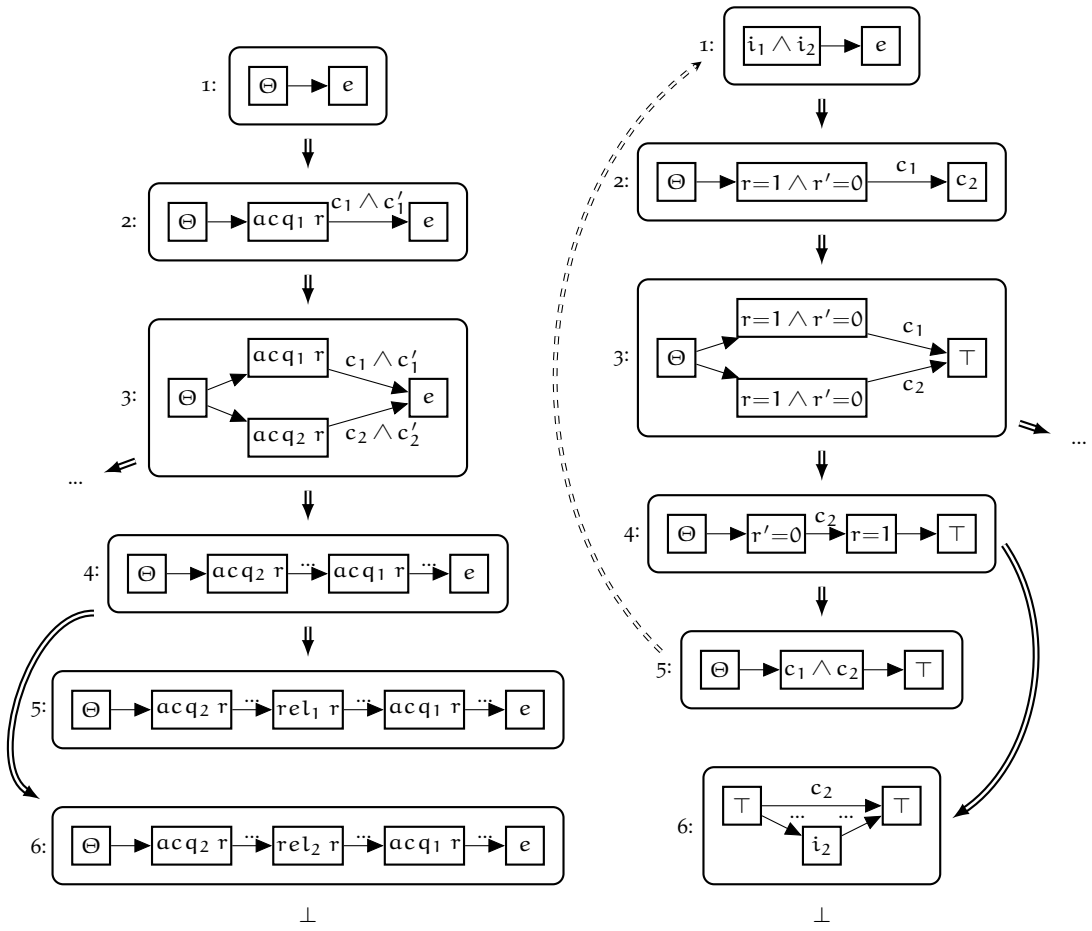


Figure 3.7: Abstract trace tableau for binary semaphore (Figure 3.6). Concrete trace unwinding is seen on the left, abstract looping trace tableau on the right. Error condition is $e \equiv c_1 \wedge c_2$, initial condition is $\Theta = i_1' \wedge i_2'$.

4

TIMED CONCURRENT TRACES

Syntactically, we need not alter much to adapt concurrent traces as outlined in Chapter 3 to timed automata. Labels of events and causal links now not only reason about discrete variables, but also about the clocks and locations of timed automata. To this end, we encode the transitions of timed automata in first-order logic. The finite set of locations L can be encoded in a fixed number of discrete variables. We denote the location variables by $\mathcal{L}$ and $\mathcal{L}'$ to specify the source and target location of some transition, respectively. For readability, we will omit explicit valuations of location variables in this thesis and write l_1 to mean a valuation corresponding to location l_1. Guards are encoded as conditions over unprimed integer or clock variables, while assignments and resets are conditions over primed variables.

DEFINITION 4.1 (Timed concurrent trace). A *timed concurrent trace* is a tuple $\mathcal{F}_t = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, where:

- $\mathcal{E}$ is a set of *events*;
- $\mathcal{C} \subset \mathcal{E} \times \mathcal{E}$ is a set of *causal links*;
- $\lightning \subseteq \mathcal{E} \times \mathcal{E}$ is a symmetric *conflict relation*;
- $\lambda_\mathcal{E} : \mathcal{E} \to (\phi^D(\mathcal{L} \cup \mathcal{L}' \cup V \cup V'), \phi^C(C), \phi^R(C))$ and $\lambda_\mathcal{C} : \mathcal{C} \to (\phi^D(\mathcal{L} \cup \mathcal{L}' \cup V \cup V'), \phi^C(C), \phi^R(C))$ are labelings of events and causal links with transition predicates.

We split the labels of events and links between the discrete part composed of location variables and integer variables, clock constraints and clock resets, because as we will soon see, we need to treat constraints on integer and clock variables differently.

4.1 TRACE LANGUAGE

DEFINITION 4.2 (Trace language). The *language* of a timed concurrent trace $\mathcal{F}_t = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ is defined as a set $\mathcal{L}(\mathcal{F}_t)$ of timed computations, such that for each computation $\pi = s_0, ..., s_n \in \mathcal{L}(\mathcal{F}_t)$, with $s_i = \langle l_i, \nu_i^c, \nu_i^i, t_i \rangle$, there exists a mapping $\sigma : \mathcal{E} \to \{s_0, ..., s_n\}$, called a *run* of $\mathcal{F}_t$ on π, such that:

1. for each event $e \in \mathcal{E}, \lambda_\mathcal{E}(e) = (\phi^D, \phi^C, \phi^R)$ and $s_i = \sigma(e)$ it holds that $(l_{i-1}, l_i, \nu_{i-1}^i, \nu_i^i) \models \phi^D, \nu_{i-1}^c + t_i - t_{i-1} \models \phi^C$ and for all $x \in R$ so that $\nu_i^c = \nu_{i-1}^c + t_i - t_{i-1}[R] : \phi^R(x)$ and for all $y \in C \setminus R : \neg\phi^R(y)$;

2. for each causal link $(e_1, e_2) \in \mathcal{C}$, and $s_i = \sigma(e_1)$ and $s_j = \sigma(e_2)$, we have:
   a) $i \leqslant j$, and
   b) for all $i < k < j$, the formula $\lambda_{\mathcal{C}}(c)(s_k, s_{k+1})$ holds.
3. for each pair of conflicting events $e_1 \not\downarrow e_2$, with $s_1 = \sigma(e_1)$ and $s_2 = \sigma(e_2)$, we have $i \neq j$.

4.2 CHECKING EMPTINESS

As outlined in [10], checking an arbitrary concurrent trace for emptiness is NP-complete already for untimed concurrent traces. Therefore, causality-based verification utilizes the compactization of a concurrent trace as an underapproximation for emptiness checking. Recall that a compactization is a concurrent trace that allows no transitions besides the events explicitly specified by the trace. This makes emptiness checking trivial in the untime case. One simply needs to construct the *static single assignment (SSA)* form of a trace and check the resulting formula for satisfiability.

This treatment, however, is insufficient for timed concurrent traces. Consider, for instance, the trace shown in Figure 4.1. The SSA-form of its compactization looks like this:

$$l_0 = 0 \wedge x_0 = 0 \wedge l_0 = 0 \wedge l_1 = 1 \wedge x_0 > 1$$

This formula is unsatisfiable, as it requires x_0 to be zero and greater than one at the same time. However, the language of the compactization should not be empty, as x is a clock variable and the constraint can be satisfied with any number of delay transitions that let more than one time unit pass. The problem is that, while the SSA-form is enough for the discrete parts of the trace, the naive approach fails for the clock variables. This is because it does not allow for delay transitions between the events. In our method for emptiness checking of timed concurrent traces, we will therefore split the constraint system for satisfiability checking between discrete variables and continuous clocks. We will first formalize the discrete, SSA part.

DEFINITION 4.3 (SSA-form). For a compactization $\mathcal{F}_t = \phi_0, \phi_1, ..., \phi_n$, where $\phi_i = (\phi_i^D, \phi_i^C, \phi_i^R)$, we define the *static single assignment form* of $\mathcal{F}_t$ as:

$$SSA(\mathcal{F}_t) = \bigwedge_{0 \leqslant i \leqslant n} \phi_i^D[\mathcal{L} \cup V / \mathcal{L}_{i-1} \cup V_{i-1}][\mathcal{L}' \cup V' / \mathcal{L}_i \cup V_i]$$

.

When checking the constraints of the continuous clock variables, the most important part is to model the implicit synchronization between clocks. The question we therefore have to answer is whether

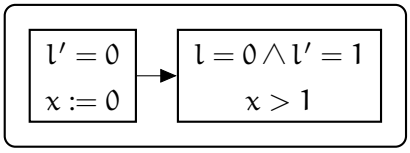


Figure 4.1: Timed concurrent trace $\mathcal{F}_t$.

there are delays $d_{0,\dots,n} \in \mathbb{R}$ between the events so that all clocks fulfill the constraints put upon them. We do this by linking each event e_i to some abstract time point t_i and modeling the clock values at each constraint based on these abstract values. After all, at any point in time t_i, a clock evaluates to the amount of time that has passed since the last reset - or in other words: the difference between the current point of time t_i and the time of the last reset t_k. The following lemma formalizes this notion.

LEMMA 4.4. For some timed computation $\pi = s_0, \dots, s_n$ with $s_i = \langle l_i, \nu_i^c, \nu_i^i, t_i \rangle$ it holds for any i with $0 < i \leqslant n$ and any clock $x \in C$: $\nu_{i-1}^c(x) + t_i - t_{i-1} = t_i - t_k$ for all k so that $0 \leqslant k < i$ and there exist $R, N \subseteq C$ so that $\nu_k^c = \nu_{k-1}^c + t_k - t_{k-1}[R]$ and $\nu_j^c = \nu_{j-1}^c + t_j - t_{j-1}[N]$ for all $j : k < j \leqslant i$ and $x \in R \setminus N$.

Proof. By induction. Induction basis: For s_1 and any clock $x \in C$ it holds that $\nu_0^c(x) + t_1 - t_0 = t_1 - t_0$.
Induction step: Consider some i with $1 < i \leqslant n$ and any clock $x \in C$. We make a distinction between two cases: For $\nu_{i-1}^c = \nu_{i-2}^c + t_{i-1} - t_{i-2}[R]$, either $x \in R$ or not.
If $x \in R$, then $k = i - 1$ and $\nu_{i-1}^c(x) = 0$, it follows $\nu_{i-1}^c(x) + t_i - t_{i-1} = t_i - t_{i-1} = t_i - t_k$.
If $x \notin R$, we have by induction hypothesis that $\nu_{i-2}^c(x) + t_{i-1} - t_{i-2} = t_{i-1} - t_k$. With $x \notin R$, we know that $\nu_{i-1}^c(x) = \nu_{i-2}^c(x) + t_{i-1} - t_{i-2}$, it follows $\nu_{i-1}^c(x) = t_{i-1} - t_k$. By adding $t_i - t_{i-1}$ on both sides we get $\nu_{i-1}^c(x) + t_i - t_{i-1} = t_i - t_k$. □

DEFINITION 4.5 (Temporal consistency constraint). For a compactization $\mathcal{F}_t = \phi_0, \phi_1, \dots, \phi_n$, where $\phi_i = (\phi_i^D, \phi_i^C, \phi_i^R)$, we define the *temporal consistency constraint* of $\mathcal{F}_t$ as:

$$TCC(\mathcal{F}_t) = t_0 = 0 \wedge \bigwedge_{0<i\leqslant n} \phi_i^C[t_i - t_k/C] \wedge t_i \geqslant t_{i-1}$$

with $0 \leqslant k < i : \phi_k^R(x)$ and $\forall k < j < i : \neg\phi_j^R(x)$ for any clock $x \in C$.

EXAMPLE 4.6. Consider the timed concurrent trace seen in Figure 4.1. The temporal consistency constraint of the concurrent trace is

$$t_0 = 0 \wedge t_1 \geqslant t_0 \wedge t_1 - t_0 \geqslant 1$$

This formula is satisfiable for any $t_1 > 1$ and accurately models the constraints put upon clock variables.

We will use the temporal consistency constraint in conjunction with the discrete parts of the compactization in SSA-form as a means to underapproximate the emptiness check for a timed concurrent trace. Lemma 4.7 provides the formal basis for this method.

LEMMA 4.7. For a timed concurrent trace $\mathcal{F}_t$ and its compacitzation $\mathcal{F}'_t = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ it holds that $\mathcal{L}(\mathcal{F}'_t)$ is not empty iff $SSA(\mathcal{F}'_t) \wedge TCC(\mathcal{F}'_t)$ is satisfiable.

Proof. Let $\mathcal{F}_t$ be a timed concurrent trace and $\mathcal{F}'_t = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$ a compactization of $\mathcal{F}_t$. Let the compactization normal form of $\mathcal{F}'_t$ be $\phi_0, \phi_1, ..., \phi_n$ with $\phi_i = (\phi_i^D, \phi_i^C, \phi_i^R)$.

1. Lets assume $\mathcal{L}(\mathcal{F}'_t)$ is not empty, then there exists some timed computation $\pi = s_0, ..., s_n$ with $s_i = \langle l_i, \nu_i^c, \nu_i^i, t_i \rangle$ and a mapping $\sigma : \mathcal{E} \to \{s_0, ..., s_n\}$. Since $\mathcal{F}'_t$ is a compactization and does not allow implicit edge transitions, it holds for any i with $0 \leqslant i \leqslant n$, $\sigma(e_i) = s_i$. From the definition of $\mathcal{L}(\mathcal{F}'_t)$, it holds for each e_i and $s_i = \sigma(e_i)$ with $\lambda_\mathcal{E}(e_i) = (\phi_i^D, \phi_i^C, \phi_i^R)$: $(l_{i-1}, l_i, \nu_{i-1}^i, \nu_i^i) \models \phi_i^D$, so $\phi_i^D[\mathcal{L} \cup V / \mathcal{L}_{i-1} \cup V_{i-1}][\mathcal{L}' \cup V' / \mathcal{L}_i \cup V_i]$ is satisfiable for any i. It follows that $SSA(\mathcal{F}_t)$ is satisfiable. We further now that $\nu_{i-1}^c + t_i - t_{i-1} \models \phi_i^C$. With Lemma 4.4 we know for any clock $x \in C : \nu_{i-1}^c(x) + t_i - t_{i-1} = t_i - t'_k$ for all k' so that $0 \leqslant k' < i$ and there exist $R, N \subseteq C$ so that $\nu_{k'}^c = \nu_{k'-1}^c + t_{k'} - t_{k'-1}[R]$ and $\nu_j^c = \nu_{j-1}^c + t_j - t_{j-1}[N]$ for all $j : k' < j \leqslant i$ and $x \in R \setminus N$. Therefore, it especially holds true for a $k < i$ with $\phi_k^R(x)$ and $\forall k < j < i : \neg\phi_j^R(x)$ and it follows that $\phi_i^C[t_i - t_k / C]$ is satifiable for any i. Therefore, $TCC(\mathcal{F}'_t)$ is satisfiable.

2. Lets assume $SSA(\mathcal{F}'_t) \wedge TCC(\mathcal{F}'_t)$ is satisfiable, then there exist $t_{0,1,...,n} \in \mathbb{R}$ that are a solution for $TCC(\mathcal{F}'_t)$. Let $\pi = s_0, ..., s_n$ be some timed computation with $s_i = \langle l_i, \nu_i^c, \nu_i^i, t_i \rangle$ where ν_i^i is formed by $\nu_i^i = \nu_{i-1}^i + t_i - t_{i-1}[R]$ with $R = \{x \in C \mid \phi_i^C(x)\}$. We now show that we can construct π so that is in $\mathcal{L}(\mathcal{F}'_t)$. Consider mapping $\sigma : \mathcal{E} \to \{s_0, ..., s_n\} : \sigma(e_i) = s_i$. From the satisfiability of subformula $TCC(\mathcal{F}'_t)$, we know that $t_i \geqslant t_{i-1}$ for all $1 \leqslant i \leqslant n$ and also $t_0 = 0$. We have already ensured for all $x \in R$ with $\nu_i^c = \nu_{i-1}^c + t_i - t_{i-1}[R] : \phi^R(x)$ and for all $y \in C \setminus R : \neg\phi^R(y)$ by construction. With Lemma 4.4 we know that for all $i > 0$ and any clock $x \in C$: $\nu_{i-1}^c(x) + t_i - t_{i-1} = t_i - t_k$ for all k so that $0 \leqslant k < i$ and there exist $R, N \subseteq C$ so that $\nu_k^c = \nu_{k-1}^c + t_k - t_{k-1}[R]$ and $\nu_j^c = \nu_{j-1}^c + t_j - t_{j-1}[N]$ for all $j : k < j \leqslant i$ and $x \in R \setminus N$. These requirements are met by a k with $0 \leqslant k < i : \phi_k^R(x)$ and $\forall k < j < i : \neg\phi_j^R(x)$. Since $\phi_i^C[t_i - t_k / C]$ is satisfiable, we know that $\nu_{i-1}^c + t_i - t_{i-1} \models \phi_i^C$. $SSA(\mathcal{F}'_t)$ being satisfiable means there is a solution $l'_0, ..., l'_n$ and for any variable $\nu' \in V : \nu'_0, ..., \nu'_n$. We construct $l_i = l'_i$ and $\nu_i^i(\nu) = \nu'_i$. Therefore, for $\sigma(e_i) = s_i$ it holds that $(l_{i-1}, l_i, \nu_{i-1}^i, \nu_i^i) \models \phi^D$

and σ fully satisfies the first requirement. For the second and third requirements, recall that $\mathcal{F}'_t$ is a compactization and therefore does not allow any additional transitions between events. Further, no events are allowed to be mapped to the same index. We have already ensured both requirements by constructing σ as a bijection. It follows that $\pi \in \mathcal{L}(\mathcal{F}'_t)$.

□

In the following, we provide four auxiliary functions $SSA(F)$, $TCC(F)$, $Satisfiable(F)$ and $UnsatSubtrace(F)$, which we will use for emptiness checking at a later stage. For some compactization F, $SSA(F)$ builds its SSA formula and $TCC(F)$ its timed consistency constraint. $Satisfiable(F)$ checks both formulas for satisfiability. Our goal, however, will eventually be to identify and refine the parts of the timed concurrent trace that are responsible for a potential unsatisfiability result. This process is realized in the $UnsatSubtrace(F)$ function. For both SSA and TCC form, it is possible to link any conjunct to an event or link of the original trace. $UnsatSubtrace(F)$ does this for the conjuncts in the unsatisfiable core of the formulas and in this way produces the part of the original trace that needs further refinement.

Function : SSA(F)

In : compactization F
Out : formula in SSA form
begin
  let $N_C(F) = \phi_1, ..., \phi_n$ where $\phi_i = (\phi_i^D, \phi_i^C, \phi_i^R)$
  set $\psi \leftarrow \top$ **foreach** $i \in [1, n]$ **do**
    set $\psi \leftarrow \psi \wedge \phi_i^D(L^{-1} \cup V^{-1}, L \cup V)$
  return ψ

Function : TCC(F)

In : compactization F
Out : formula in TCC form
begin
  let $N_C(F) = \phi_1, ..., \phi_n$ where $\phi_i = (\phi_i^D, \phi_i^C, \phi_i^R)$
  let $r : C \to \mathbb{N}$ with $r(x) = 0 \; \forall x \in C$
  set $\psi \leftarrow t_0 = 0$
  foreach $i \in [1, n]$ **do**
    set $\psi \leftarrow \psi \wedge \phi_i^C(t_i - t_{r(x \in C)}/C) \wedge t_i \geqslant t_{i-1}$
    foreach $x \in C$ **do**
      if $\phi_i^R(x)$ **then**
        set $r(x) = i$
  return ψ

Function : Satisfiable(F)

In : timed concurrent trace F
Out : $\mathtt{true}/\mathtt{false}$
begin
  select $\mathsf{F}' \in \mathtt{Compactizations}(\mathsf{F})$
  set $\phi \leftarrow \mathtt{SSA}(\mathsf{F}')$
  set $\psi \leftarrow \mathtt{TCC}(\mathsf{F}')$
  return $\mathtt{sat}(\phi \wedge \psi)$

Function : UnsatSubtrace(F)

In : timed concurrent trace F
Out : timed concurrent trace $\mathsf{F}' \supseteq \mathsf{F}$
begin
  select $\mathsf{F}' \in \mathtt{Compactizations}(\mathsf{F})$
  set $\psi \leftarrow \mathtt{SSA}(\mathsf{F}') \wedge \mathtt{TCC}(\mathsf{F}')$
  set $\Psi \leftarrow \mathtt{mininmal\_unsat\_core}(\psi)$
  set $\mathsf{F}' \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$
  let $\mathtt{add}(c) \equiv$ **if** $c \notin \mathcal{E}'$ **then**
    set $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{c\}$
  foreach $\phi_i \in \Psi$ **do**
    if ϕ_i *is from* $\lambda_{\mathcal{E}}(c_j)$ **then**
      $\mathtt{add}(c_j)$
      $\lambda'_{\mathcal{E}}(c_j) \leftarrow \phi_i$
    if ϕ_i *is from* $\lambda_{\mathcal{C}}((c_j, c_k))$ **then**
      $\mathtt{add}(c_j); \mathtt{add}(c_k)$
      $\mathcal{C}' \leftarrow \mathcal{C}' \cup (c_j, c_k)$
      $\lambda'_{\mathcal{C}}(c_j, c_k) \leftarrow \phi_i$
  set $\lightning' \leftarrow \lightning \cap \mathcal{E} \times \mathcal{E}'$
  return F'

5

CAUSALITY-BASED VERIFICATION IN REAL-TIME

In the previous chapter, we have already seen that clock variables require special treatment for emptiness checks because of their implicit synchronization. We will now focus on the refinement process of temporally inconsistent traces. Our goal will be to causally infer operations on clock variables, given the context of an unsatisfiable timed concurrent trace. To this end, we define a number of trace transformers applicable to timed concurrent traces. These transformers then get utilized in our causality-based model checking algorithm for safety properties. Lastly, we provide an comprehensive example of applying real-time causality-based verification to Fischer's protocol, a common benchmark for timed automata model checking.

Before formally defining real-time trace transformers, we want to illustrate their major intuition by example.

EXAMPLE 5.1. Consider therefore timed concurrent trace $\mathcal{F}_t$ shown in Figure 5.1. While the discrete parts of the trace are satisfiable, its implied temporal constraint system is not. The produced unsatisfiable subtrace $\mathcal{F}'_t$ therefore can be seen as a projection of the real-time properties of the concurrent trace. The question is, how can we repair the given subtrace? It is clear that the only valid operation in a timed automaton on clock values is a reset. Consequently, the question can be reduced to introducing a clock reset to the trace that changes the validity of its temporal constraint system. Upon closer inspection of $\mathcal{F}'_t$, we can see that it is composed of both lower bounds on clocks y and z (e.g. $y > 1$ and $z > 1$) and an upper bound on x (e.g. $x \leqslant 2$). Further, due to its causal links, the time between the lower bounds and their respective resets lies in the scope of the upper bound, that is to say, between the last reset of x and the clock constraint $x \leqslant 2$. Because

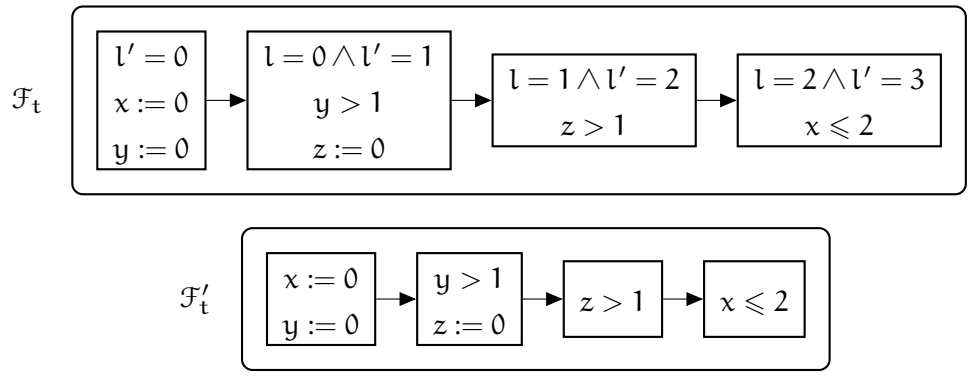


Figure 5.1: Timed concurrent trace $\mathcal{F}_t$ and unsatisfiable subtrace $\mathcal{F}'_t$.

z gets reset at the time y is greater than 1, we can safely assume that the time between the first and last event of the trace has to be more than two time units: After all, the only operation on clock values is a reset. While a reset technically decreases the clock values, the overall time spent between two events is greater or equal than before. This effectively means that no reset of y or z can repair the trace at this stage, but a reset of x can. Informally, we want to allow more time to pass in clock x between the upper bound $x \leqslant 2$ and the last reset that goes before. Figure 5.2 shows the trace after the insertion of a clock reset. Note that, in general, the time between the first and last event is now completely unrestricted, because there is no constraint on the time between first event and new reset.

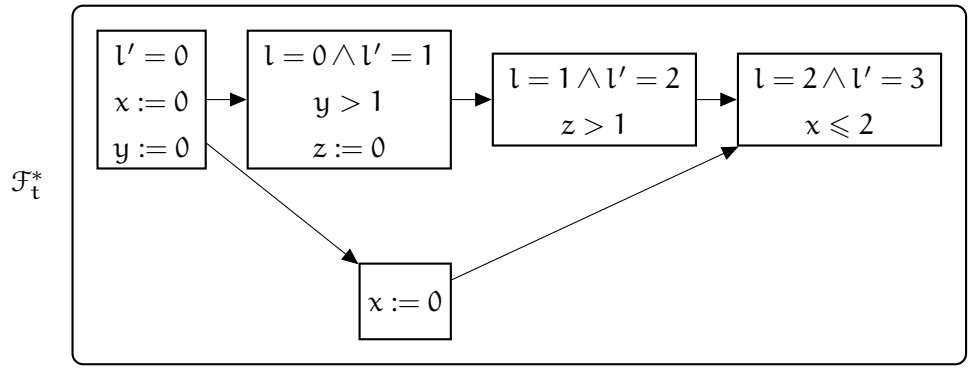


Figure 5.2: Repaired trace $\mathcal{F}_t^*$ after clock reset in $\mathcal{F}_t$ from Figure 5.1.

In the next section, we first formalize the inference rules for clock resets as trace transformers on timed concurrent traces. Subsequently, we will focus on how to identify the clocks which need to be reset.

5.1 TIMED TRACE TRANSFORMERS

Reset Split (Figure 5.3)

The $\mathtt{ResetSplit}(a, b, x)$ trace transformer performs a case split, given two conflicting concurrent events a, b and a clock x. Either x gets reset somewhere between a and b, or not. Formally, we have

$\mathtt{pre}(\mathtt{ResetSplit}(a, b, x)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

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

$post(ResetSplit(a, b, x)) = \{R_1, R_2\}$, where:

- $R_1 = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \{(a, b) \to x \notin R \wedge \neg x \leqslant 0\}\rangle$;
- $R_2 = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}}\rangle$, with
  - $\mathcal{E}' = \{a, b, c\}$;
  - $\mathcal{C}' = \not\downarrow' = \{(a, b), (a, c), (c, b)\}$;
  - $\lambda'_{\mathcal{E}} = \{a \to \top, b \to \top, c \to x \in R\}$;
  - $\lambda'_{\mathcal{C}} = \{(a, b) \to \top, (a, c) \to \top, (c, b) \to \top\}$.

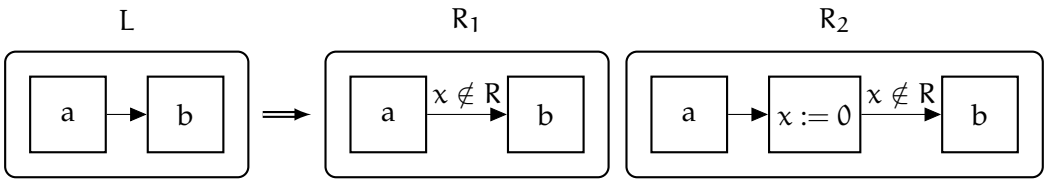


Figure 5.3: Reset Split trace transformer.

PROPOSITION 5.2. The *Reset Split* trace transformer is sound.

Proof. Let $\sigma(a) = s_i$ and $\sigma(b) = s_j$. As a and b are in conflict, it holds that $i < j$. Consider a computation $\pi \in \mathcal{L}(F)$ with $i + 1 = j$, i.e. there is no other event between a and b. Then it trivially holds that $x \notin R$ for all events between a and b and $\pi \in \mathcal{L}(R_1)$. Now consider a computation $\pi \in \mathcal{L}(F)$ with $i + 1 < j$, i.e. there is an arbitrary number of events between a and b. Either it holds for all indices k with $i + 1 \leqslant k < j$ that $v^c_{k+1} = v^c_k + t_{k+1} - t_k[R]$ with $x \notin R$, or there exists at least one index k with $i + 1 \leqslant k < j$ so that $v^c_{k+1} = v^c_k + t_{k+1} - t_k[R]$ with $x \in R$. In the former case, it holds that $x \notin R$ for all events between a and b and $\pi \in \mathcal{L}(R_1)$. In the latter case, we have an index k so that $(x \in R)(l_{k-1}, v^c_{k-1} + t_k - t_{k-1}, v_{k-1}, l_k, R, v^i_k)$ holds and that qualifies for $\sigma(c) = s_k$, it follows $\pi \in R_2$. □

Timed Backward Unrolling (Figure 5.4)

The $TimedBackwardUnrolling(a, b, l_b, Z_b, v^i_b)$ trace transformer implements one step of backwards reachability analysis by instantiating all system transitions that can precede b. Formally, given a timed automaton $\langle L, l_0, \Sigma, C, V, I, E\rangle$, let $\{e_1, ..., e_k\} = \{e \in E \mid sat(e \wedge l'_b \wedge v^i_b) \wedge Pre_e(Z_b) \neq \emptyset\}$ be the set of transitions that can immediately and validly precede b. Then:

$pre(TimedBackwardUnrolling(a, b, l_b, Z_b)) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}}\rangle$, where:

- $\mathcal{E} = \{a, b\}$;
- $\mathcal{C} = \{(a, b)\}$;

- $\lightning = \{(a, b)\}$;
- $\lambda_{\mathcal{E}} = \{a \to \neg l'_b, b \to l_b \wedge Z_b \wedge v^i_b\}$;
- $\lambda_{\mathcal{C}} = \emptyset$.

$post(TimedBackwardUnrolling(a, b, l_b, Z_b)) = \{R_1, ..., R_k\}$, and:

- $R_i = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_{\mathcal{E}_i}, \lambda'_{\mathcal{C}} \rangle$, where with $e_i = l_i \xrightarrow{\alpha_i, \beta_i, \gamma_i, \omega_i, \rho_i} l_b$:
  - $\mathcal{E}' = \mathcal{E} \cup \{c\}$;
  - $\mathcal{C}' = \mathcal{C} \cup \{(a, c), (c, b)\}$;
  - $\lightning' = \lightning \cup \{(a, c), (c, b)\}$;
  - $\lambda'_{\mathcal{E}_i} = \lambda_{\mathcal{E}} \cup \{c \to l_i \wedge Pre_{e_i}(Z_b) \wedge wp_{\omega_i}(v^i_b) \wedge \gamma\}$;
  - $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \top, (c, b) \to \bot\}$.

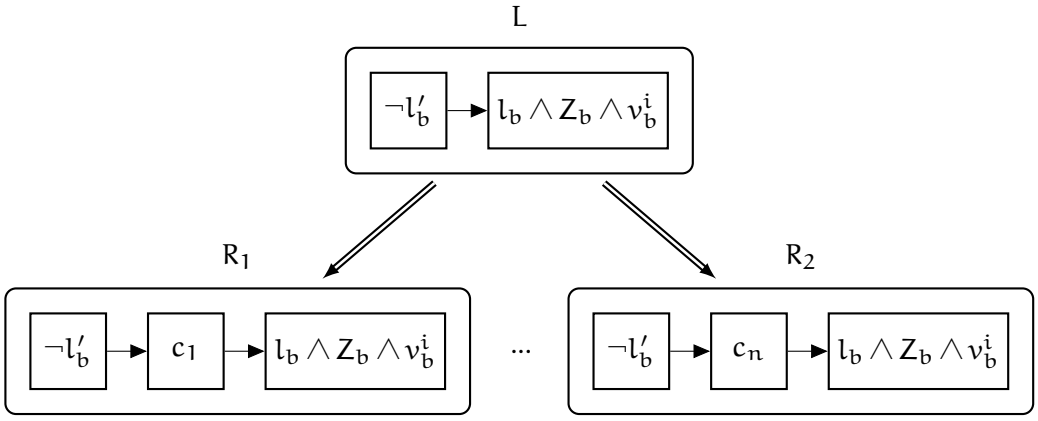


Figure 5.4: Timed Backward Unrolling trace transformer. With $\mathcal{E}(c_i) = l_i \wedge Pre_{e_i}(Z_b) \wedge wp_{\omega_i}(v^i_b) \wedge \gamma_i \wedge l'_b$ for some edge $e_i = l_i \xrightarrow{\alpha_i, \beta_i, \gamma_i, \omega_i, \rho_i} l_b \in TA$

5.2 REFINEMENT OF TIMED CONCURRENT TRACES

The trace transformers for timed concurrent traces provide a single building block for proof construction. As a next step, we need to define how to apply them in exploration of the abstract trace tableau as seen in Chapter 3.

The same way as for discrete systems, we apply the trace transformers in function *SafetyRefinement*. Many parts of the function are inherited from the discrete case [10]. First, an unsatisfiable subtrace is generated with a call to *UnsatSubtrace*. The analysis proceeds with a case distinction if there is a pair of events that are not yet in conflict by applying trace transformer *ConflictSplit*. The same is done in case there is a pair of events that are unordered.

Subsequently, for a trace that where all events are ordered and in conflict, we proceed by analyzing whether its unsatisfiability stems

Function : SafetyRefinement(F)

In : timed concurrent trace $F = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$
Out : transformer τ, morphism $m : pre(\tau) \to F$
begin
  set $F' = \langle \mathcal{E}', \mathcal{C}', \not\downarrow', \lambda'_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle \leftarrow \mathit{UnsatSubtrace}(F)$
  if $\exists e_1, e_2 \in \mathcal{E}'.(e_1, e_2) \notin \not\downarrow$ **then**
    return $\mathit{ConflictSplit}(e_1, e_2)$
  if $\exists e_1, e_2 \in \mathcal{E}'.(e_1, e_2) \notin \mathcal{C}$ **then**
    return $\mathit{OrderSplit}(e_1, e_2)$
  let $F^* = \mathit{Compactizations}(F')$
  if $\neg \mathit{TCC}(F^*)$ **then**
    if $(x, e_1, e_2) \leftarrow \mathit{ViolatedBound}(F', \lambda_{\mathcal{C}})$ *is not* null **then**
      return $\mathit{Instantiate}(c) \circ \mathit{ResetSplit}(e_1, e_2, x)$
    else
      return $\mathit{Contradiction}(\mathcal{E}')$
  switch $\mathcal{E}'$ **do**
    case $\{e_1\}$ **do**
      return $\mathit{Contradiction}(e_1)$
    case $\{e_1, e_2\}$ **do**
      $\phi \leftarrow \mathit{interpolate}(\lambda_{\mathcal{E}}(e_1)\ ;\ \lambda_{\mathcal{E}}(e_2)')$
      return $\mathit{Instantiate}(c) \circ \mathit{LastNecessaryEvent}(a \to e_1, b \to e_2, \phi_{[V'/V]})$
    otherwise do
      else
        $\phi \leftarrow \mathit{interpolate}(\lambda_{\mathcal{E}}(e_1) \wedge \lambda_{\mathcal{E}}(e_2)' \wedge \ \ldots\ \wedge \lambda_{\mathcal{E}}(e_{k-1})^{k-2}\ ;\ \lambda_{\mathcal{E}}(e_k)^{k-1})$
      return $\mathit{EventSplit}(a \to e_{k-1}, \phi_{[V^{k-1}, V']})$

from its temporal constraints. We check this by extracting a compactization, which is deterministic as all events are ordered, and checking its TCC formula. If it is unsatifiable, we search for some violated lower bound with the function *ViolatedBound*.

This function recursively searches the ordered set of events starting from the last and produces a pair of events and the clock to be reset as long as the respective causal link does not disqualify the insertion. As the unsatisfiable subtrace was produced from a minimal unsatifiable core, any constraint containing $<$ or $\leqslant$ is part of a violated upper bound and qualifies for the reset. Identifying whether an equality constraint is part of a violated bound is more challenging however, as they function as both lower and upper bounds at the same time. The key is to identify which function is essential for the given subtrace. We check this by replacing the equality constraints in the last event with ge. This changes nothing in case the constraint is responsible for the violation of another, smaller bound. But if it is the violated

Function : ViolatedBound(F)

In : unsatisfiable subtrace $F = \langle \{e_1, ..., e_k\}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, original labeling function $\lambda^*_\mathcal{C}$

Out : clock x, events e_1, e_2

begin

let $\mathtt{LastReset}(e_k) \equiv$ **if** $\exists\ e_i$ *so that* $\lambda_\mathcal{E}(e_i) \to x \in R \wedge \forall j$ *with* $i < j < k : \lambda_\mathcal{E}(e_j) \to x \notin R$ **then** e_i **else** e_0

if $k < 2$ **then**
  return *null*

if $\exists x \in C, n \in \mathbb{N}.\ \lambda_\mathcal{E}(e_k) \to x \{<, \leqslant\} n$ **then**
  $e_i \to \mathtt{LastReset}(e_k)$
  if $\lambda^*_\mathcal{C}(e_i, e_k) \to \neg x \notin R$ **then**
    return (x, e_i, e_k)

foreach $x \in C$ *where* $\exists n \in \mathbb{N}.\ \lambda_\mathcal{E}(e_k) \to x\ =\ n$ **do**
  set $\lambda^*_\mathcal{E} \leftarrow$ replace $x\ =\ n$ in $\lambda_\mathcal{E}(e_k)$ with $x\ \geqslant\ n$
  set $F' \leftarrow \mathtt{UnsatSubtrace}(\langle \{e_1, ..., e_k\}, \mathcal{C}, \lightning, \lambda^*_\mathcal{E}, \lambda_\mathcal{C} \rangle)$
  let $F' = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_\mathcal{E}, \lambda'_\mathcal{C} \rangle$
  if $e_k \notin \mathcal{E}' \vee \neg\lambda_\mathcal{E}(e_k) \to x\ \geqslant\ n$ **then**
    $e_i \to \mathtt{LastReset}(e_k)$
    if $\lambda^*_\mathcal{C}(e_i, e_k) \to \neg x \notin R$ **then**
      return (x, e_i, e_k)

return $\mathtt{ViolatedBound}(\langle \{e_1, ..., e_{k-1}\}, \mathcal{C}, \lightning, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle)$

bound itself, the minimal unsatisfiable core collapses and upon applying UnsatSubtrace again, the constraint will not be found. In a way, this simulates the insertion of a clock reset.

In the case we do not find a place to insert a reset, but the TCC formula is unsatifiable, we can safely assume that the trace is contradictory. We close the branch by applying the Contradiction trace transformer, which does not produce a new node.

If the unsatisfiable subtrace stems from unsatisfiability of the SSA formula, different cases are considered with respect to the number of events in the unsatisfiable subtrace. A subtrace with only a single event means the label of a single event is contradictory, consequently there is no possible refinement step and the exploration stops at this point with trace transformer Contradiciton. If there are two events, we apply Craig interpolation and transformer LastNecessaryEvent to try and repair the conflict by introducing a bridging event in between. Finally, in case of more than two events, the unsatisfiable subtrace gets shortened by resolving the conflict in the last event first. To this end, the interpolant between all preceding events and the last is computed and used for trace transformer EventSplit, which, essentially, produces a case split between whether the interpolant is already satisfied in the preceding event or not.

Algorithm : Exploration of Abstract Trace Tableau

In : timed automaton $TA = \langle L, l_0, \Sigma, C, V, I, E \rangle$, safety property ϕ
Out : property holds/counterexample
Data : abstract trace tableau $\Lambda = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$ with 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 Λ ← InititalAbstractTableau(TA, φ), Q ← N
    while Q is not empty do
        select some n from Q
        if Satisfiable(γ(n)) then
            return counterexample γ(n)
        else
            if ⟨pre(τ), m⟩ ← TryCover(Λ, n) is not ⊥ then
                continue
            else
                set ⟨τ, m⟩ ← SafetyRefinement(γ(n))
                Apply(Υ, τ, m, n)
        set Q ← Q ∪ {n' | (n, n') ∈ E} \ {n}
        PropagateUp(Γ̂, pre(τ), m, n)
    return property holds
```

5.3 EXPLORATION OF TRACE TABLEAU

In the following, we expand upon how the refinement of timed concurrent trace is embedded in the algorithm for exploring the abstract trace tableau. Our algorithm is closely based on the exploration algorithm proposed by Andrey Kupriyanov in [10] and informally described in Chapter 3, with minor adjustments to accommodate timed automata. The algorithm realizes the highest level of proof search and works by keeping a queue of nodes labeled with timed concurrent traces waiting to be refined. Given a system modeled as a timed automaton and a safety property ϕ, it initializes an abstract trace tableau with some abstract error traces symbolizing any possible error traces. For this, the function $\mathtt{IntitialAbstractTableau}$ is used, which itself makes use of the $\mathtt{Abstract}$ function. This denotes a function for generating abstract system violations for an automaton and some safety property. These initial nodes are the first put into the queue. The algorithm terminates when there are no nodes in the queue left or when a counterexample is found along the way. When taking a node out of the queue, first we check whether the compactization of its label is satisfiable. In that case, the label corresponds to a valid counterexample and the property does not hold.

Function : InitialAbstractTableau(TA, ϕ)

In : timed automaton TA, property ϕ
Out : abstract trace tableau $\Lambda = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$
begin
  set all of $\{N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\delta}, \hat{\mu}\} \leftarrow \emptyset$
  foreach $F \in \mathtt{Abstract}(TA, \phi)$ **do**
    set $N \leftarrow N \cup \{n\}$, where n is a fresh node
    set $\gamma(n) \leftarrow F, \sigma(n) \leftarrow \emptyset, \hat{\gamma}(n) \leftarrow$ empty trace

Function : TryCover(Λ, ϕ)

In : abstract trace tableau $\Lambda = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \sigma \rangle$, 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 $\hat{\gamma}(n) \leftarrow \hat{\gamma}(n'), \sigma(n) \leftarrow m$
    put (n, n') into $\rightsquigarrow$
    return $\langle \hat{\gamma}(n'), m \rangle$
  else
    return $\perp$

Function : Apply(Υ, τ, m, n)

In : unwinding $\Upsilon = \langle N, E, \gamma, \delta, \mu \rangle$, transformer τ, morphism m, node n
Out : $\langle$node n, morphism $m\rangle$ / $\perp$
begin
  set $\mu(n) \leftarrow m$
  foreach $\tau_i \in \tau$ **do**
    set $N \leftarrow N \cup \{n'\}$, where n' is a fresh node
    set $E \leftarrow E \cup \{(n, n')\}$
    set $\gamma(n') \leftarrow \tau_i^m(\gamma(n))$
    set $\delta((n, n')) \leftarrow \tau_i$

If the compactization is unsatisfiable, we first check whether the node is coverable by another, earlier seen node. This is realized in function TryCover, which also updates the cover realization upon successful coverage. If the node neither is satisfiable, nor coverable, we refine the label with function SafetyRefinement, which we have detailed in the previous section. This function returns a trace transformer and morphism. Function Apply inserts all resulting, new and refined timed concurrent traces produced by the transformer into the tableau. Lastly, if a trace transformer was used, resulting in new child

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** $\nexists \hat{m} = \langle \hat{m}_{\mathcal{E}}, \hat{m}_{\mathcal{C}} \rangle : pre(\tau) \to \hat{\gamma}(n)$ *s.t.* $m = \sigma \circ \hat{m}$ **then**
  - **foreach** $o \in \gamma(n).(\exists o' \in pre(\tau).o = \xi(o')) \wedge (\nexists o'' \in \hat{\gamma}(n).o = \sigma(o''))$ **do**
    - add o' to $\hat{\gamma}(n)$, and (o, o') to $\sigma(n)$
- let $\hat{m} = \langle \hat{m}_{\mathcal{E}}, \hat{m}_{\mathcal{C}} \rangle : pre(\tau) \to \hat{\gamma}(n)$ *s.t.* $m = \sigma \circ \hat{m}$
- **if** $\hat{\gamma}(n) \not\subseteq_{\hat{m}} pre(\tau)$ **then**
  - **foreach** $e \in \mathcal{E}(pre(\tau)).(\lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \not\Rightarrow \lambda_{\mathcal{E}}(e))$ **do**
    - set $\lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \leftarrow \lambda_{\mathcal{E}}(\hat{m}_{\mathcal{E}}(e)) \wedge \lambda_{\mathcal{E}}(e)$
  - **foreach** $c \in \mathcal{C}(pre(\tau)).(\lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \not\Rightarrow \lambda_{\mathcal{C}}(c))$ **do**
    - set $\lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \leftarrow \lambda_{\mathcal{C}}(\hat{m}_{\mathcal{C}}(c)) \wedge \lambda_{\mathcal{C}}(c)$
- **foreach** $(n, n') \in \rightsquigarrow$ **do**
  - **if** $\hat{\gamma}(n') \subseteq_{\mu(n')} \hat{\gamma}(n)$ *is not forgetful* **then**
    - remove (n, n') from $\rightsquigarrow$
    - put n' into Q
- **if** $\exists\, parent\ n'.(n', n) \in E$ **then**
  - set $\langle pre(\tau)', m' \rangle \leftarrow Pullback(\delta((n', n)), m)$
  - $PropagateUp(\hat{\Gamma}, pre(\tau)', m', n')$

nodes, we need to update the abstract looping trace tableau with the transformer premise. This premise is now a vital part of information necessary to repeat not only the proof steps starting from this node, but also for the proof steps starting in all parent nodes. Therefore, procedure PropagateUp starts from the abstract label of the current node and first inserts all components that exist in the transformer premise $pre(\tau)$ but not in the abstract label into the latter. Then, their labelings get updated with the predicates from the premise. However, due to changing the abstract label, previous coverings might cease to hold. This it therefore checked and coverings are removed when necessary. Lastly, the procedure recursively moves through the tree by checking for parent nodes of a given node, and continues the process. However, the morphism m and premise $pre(\tau)$ need to be updated for the parent nodes, which is done as a pullback obejct under consideration of the trace transformer $\delta(n', n)$ between any parent node n' and node n.

We will now consider soundness and completeness of the proposed causality-based verification for timed automata. While we generally can not claim that the proposed algorithm is complete as presented, we will show that a conventional reachability algorithm can be simulated using the looping trace tableau with timed concurrent traces. This proof closely follows Andrey Kupriyanovs proof for the most

comprehensive class of infinte-state systems shown in [10] and is, in fact, a consideration under the more specialized circumstances of timed automata model checking of safety properties. The introduction of integer variables is problematic for completeness, however. These variables can lead to an infinite state space under certain circumstances, so we have to make a distinction to only allow timed automata that actually have a finite reachability quotient. We first show completeness for the easiest case, where there are no integer variables at all.

LEMMA 5.3 (Completeness for timed automata without integer variables). If a timed automaton $TA = \langle L, l_0, \Sigma, C, V, I, E \rangle$, where $V = \emptyset$, satisfies a safety property ϕ, then there exists a correct, sound and complete looping trace tableau Γ for TA and ϕ.

Proof. This proof follows the completeness proof in [10]. The idea is to simulate a standard backwards reachability analysis for timed automata, as for instance proposed in [5]. To this end, we construct a looping trace tableau $\Gamma = \langle N, E, \gamma, \delta, \mu, \rightsquigarrow \rangle$ composed solely of linear timed concurrent traces, where each trace in some node n contains only two events n_1 and n_2. We first assume that safety model checking is already reduced to reachability analysis by construction of TA. The algorithm proceeds as follows:

1. Construct an initial set of nodes labeled with the traces from function $\mathtt{Abstract}(TA, \phi)$. These traces consist of two events, the initial action Θ and the second event corresponds to some error location.

2. Apply $\mathtt{Contradiction}$ to all leaf nodes N_L with a contradictory label, thus removing them from the set of nodes. If there is a leaf node with a satisfiable label, return *Yes* (for error reachability). If there are no more leaf nodes, return *No*. We use the functions from Chapter 4 for satisfiability and emptiness checking.

3. For any leaf node $n \in N_L$, apply $\mathtt{TimedBackwardUnrolling}$ to the label of n and events n_1, n_2. Abstract the result by keeping only the first two events in the resulting nodes.

4. For any leaf node $n' \in N_L$ and each internal node $n \in N_I$, do:
   a) Apply $\mathtt{EventSplit}$ to node n' and event n'_2 with the predicate $\lambda_\mathcal{E}(n_2)$. Let the resulting nodes be n'_+ and n'_-. Event n'_{+2} is labeled with $\lambda_\mathcal{E}(n'_2) \wedge \lambda_\mathcal{E}(n_2)$. We can map it right back to event n_2 and have $\lambda_\mathcal{E}(n'_{+2})$ imply $\lambda_\mathcal{E}(n_2)$ (and n'_{+1} to n_1 in the same way, as its the initial event), we can therefore cover node n'_+ with n.
   b) Assign $n' \leftarrow n'_-$.

5. Go to step 2.

The algorithm starts from the error locations and unrolls the transition relation backwards step-by-step by utilizing the predecessor operation for clock zones as seen in Chapter 2. The Application of EventSplit ensures that only previously unseen zones of the automaton get included in the new set of leaf nodes. The set of states added in each iteration (which is defined by the label of the second event in each leaf node) is therefore defined by:

$$\bigvee_{n' \in N'_L} \lambda_\mathcal{E}(n'_2) = \bigvee_{n \in N_L} pre(\lambda_\mathcal{E}(n_2)) \wedge \bigwedge_{n \in N_I} \neg \lambda_\mathcal{E}(n_2),$$

where for a location l and zone W, pre is the symbolic backwards transitions system as seen in [5]: $pre(l, W) = (l', W')$ if there is an edge $e = l' \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} l \in E$ and $Pre_e(W) = W'$. In the formula above, the left disjunction is already enough to calculate the sets of states from the backwards reachability algorithm in [5]. The conjunction on the right, however, ensures that the computation stabilizes for acyclic systems. □

We can now lift this proof very easily to timed automata that only reach a finite number of integer valuations.

LEMMA 5.4 (Completeness for timed automata with integer variables). If a timed automaton $TA = \langle L, l_0, \Sigma, C, V, I, E \rangle$ satisfies a safety property ϕ, then there exists a correct, sound and complete looping trace tableau Γ for TA and ϕ, as long as the automaton contains no cycle that increases or decreases the value of some integer variable as a function of its previous value.

Proof. Based on Lemma 5.3, we can show this by constructing a new automaton $TA' = \langle L', l'_0, \Sigma, C, V', I', E' \rangle$ where $V' = \emptyset$ and the integer valuations of the original automaton are encoded in the locations L'. More detailed, $L' = \{(l, v) \mid (l, c, v)$ is a state of TA for some clock valuation $c\}$, $l'_0 = (l_0, v^i_0)$, $I'((l, v)) = I(l)$ and $E' = \{(l, v) \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} (l', v') \mid \exists l \xrightarrow{\alpha,\beta,\gamma,\omega,\rho} l' \in E$ s.t. $v \models \gamma, v' = v[\omega]\}$. As the original automaton TA contained no cycle that would lead to infinitely large integer values, the new automaton TA' only has a finite number of locations. According to Lemma 5.3, there exists a correct, sound and complete looping trace tableau for TA', and by construction, also for TA. □

Since we only use sound trace transformers in the algorithm presented in this chapter, we can use the proof of soundness from [10].

THEOREM 5.5 (Soundness of abstract trace tableau exploration). Let a timed automaton TA and a safety property ϕ be given. If the exploration algorithm terminates with *property holds*, and only sound trace transformers are applied in SafetyRefinement, then all computations of TA satisfy ϕ.

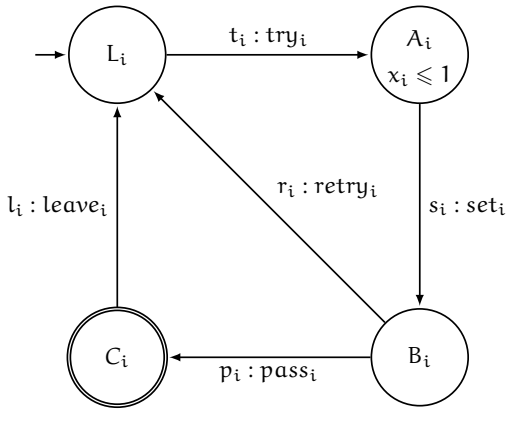
Proof. The only way for the algorithm to terminate with *property holds* is when the queue Q is empty. By construction, the final abstract looping trace tableau $\hat{\Gamma} = \langle N, E, \hat{\gamma}, \hat{\delta}, \hat{\mu}, \rightsquigarrow \rangle$ is correct, sound and complete. Moreover, for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

5.4 POLYNOMIAL VERIFICATION OF FISCHER'S PROTOCOL

EXAMPLE 5.6 (Fischer's Protocol). Consider timed automaton TA_i in Figure 5.5. It describes a single process in a larger system modeled as a network of n timed automata. The system uses Fischer's protocol for mutual exclusion. Each process has a critical section (corresponding to location C_i). Only a single process may be in its critical section at any time. The mutual exclusion is ensured by each process setting a shared variable id, waiting a predefined amount of time and then only entering the critical section when id has not been changed by another process. Processes can only start setting id while its value is zero and have to wait strictly longer than it takes to set the variable for any other process that started later. This way, only one process (the last to start setting id) manages to enter the critical section at any time. Fischer's protocol does not ensure bounded overtaking.

The trace tableau for a system composed of 3 timed automata is shown in Figure 5.6. We hide many irrelevant details with a simple "...". Initially, $\mathit{Abstract}(S, \phi)$ yields the nodes 1-3, corresponding to all abstract error traces that might violate the property ϕ. We focus on the unwinding of node 1. All other root nodes produce similar unwindings but with different variables.

As $\Theta \equiv L_1' \wedge L_2' \wedge L_3'$, Θ and e_1 do not agree on the location in post- and precondition, respectively. Consequently, the two necessary events $pass_1$ and $pass_2$ are inserted by the LastNecessaryEvent



Syntax	Semantics
try_i	$(id = 0)?,\ x_i := 0$
set_i	$id := i,\ x_i := 0$
$pass_i$	$(id = i \wedge x_i > 1)?$
$retry_i$	$(id \neq i \wedge x_i > 1)?$
$leave_i$	$id := 0$

Figure 5.5: Timed automaton TA_i for Fischer's mutual exclusion algorithm.

trace transformer and the trace of node 4 is produced. The events are yet to be ordered, so in the next step, two different interleavings are explored. We omit the exploration of the case where p_2 precedes p_1, as both cases work symmetrically.

In node 5, $pass_1$ requires $id = 1$ and does not change the value (so it holds that $id' = 1$), while $pass_2$ requires $id = 2$. Therefore event set_2 is necessary in between, as it is the only system transition that could possibly change id to 2. Consequently, node 6 is produced by the application of the LastNecessaryEvent trace transformer.

Node 6 is the first time in the unwinding where an unsatisfiable subtrace is produced due to timing inconsistency. This is because transition set_2 starts in a location labeled with invariant $x_2 \leqslant 1$, which therefore is a precondition for the corresponding event. However, event $pass_1$ is labeled with $x_1 > 1$, and further succeeded by and in conflict with set_2. Both clocks do not get reset by any other event than Θ, so their last reset before $pass_1$ and set_2 happens at the same time. This means the timing information of the timed concurrent trace is inconsistent, its timing consistency constraint is unsatisfiable. This lets us infer the existence of a necessary reset of clock x_2 between Θ and set_2.

Application of ResetSplit and Initialize yields nodes 7 and 8. It further yields an immediately contradictory node, as a reset is strictly necessary and the second case of ResetSplit will not get explored further. We omit this node from the tableau. Node 7 and 8 explore the occurrence of all possible system transitions that reset x_2. One possibility is another occurrence of set_2 as seen in node 7. Subsequently, all possible interleavings of the new set_2 and $pass_1$ are explored, as these events are unordered.

Ordering the new set_2 after $pass_1$ yields a trace that has the trace of node 6 as a subtrace. Furthermore, repeating the proof steps would pump the trace infinitely - the forgetful trace inclusion holds and we can cover this trace with node 6.

The trace unwinding from node 8, where set_2 is ordered before $pass_1$, is very similar to the case where the other possible reset of x_2, that is try_2, is chosen and ordered before $pass_1$, we therefore only show the latter in Figure 5.6.

Consider therefore the trace of node 11. Events try_1 and $pass_1$ do not agree on the value of id, therefore an event that changes id to 1 is necessary and set_1, the only such event, is introduced to the trace with LastNecessaryEvent.

However,the resulting trace in node 12 again has inconsistent timing. Clock x_1 gets reset in event set_1, after x_2 in event try_2. At the same time, it is supposed to surpass 1 (at event $pass_1$) before event set_2, where x_2 is still less or equal 1 - a contradiction.

Recall that we inserted try_2 with ResetSplit and the causal link between try_2 and set_2 therefore does not allow another reset. This

leads to the trace in node 13, where the introduction of another reset yields a single event labeled with a contradictory predicate. The language of this trace is obviously empty and we can close this branch as contradictory. Again, we omitted the case where no new reset occurs as this trace is immediately closed as contradictory.

Lets return to node 8 and see what happens if try_2 is ordered after $pass_1$. This leads to a special case, because while the two events again disagree on the value of id, try_2 does require it to be zero, not 2. However, id can be set to zero by the $leave_i$ transition of an arbitrary automaton i. Consequently all possible events get instantiated. But $leave_i$ requires automaton i to be at location C_i. While the system is also at C_1 because it is preceded by event $pass_1$. Therefore the label of the new event implies $C_1 \wedge C_i$, that is some actual error condition, which means it can be covered by some root node.

Upon inspection of the trace tableau in the previous example, it easy to see that its size is proportional to the cubic power of the number of critical sections. Moreover, the size of the timed concurrent traces labeling the nodes is independent of the number of automata. Therefore, we can make the following statement regarding the execution time of our algorithm for Fischer's protocol:

THEOREM 5.7. The causality-based model checking algorithm for real-time systems proves the safety of Fischer's protocol in deterministic polynomial time with respect to the number of automata in the network under consideration.

Proof. For k critical sections, we get $O(k^2)$ root nodes corresponding to some pair of transitions by two automata, both trying to access a critical section. Each root nodes yields a subtree of size $O(k)$: Aside from the branch where a third automaton is responsible for conflict resolution (as seen in node 9 in Figure 5.6), and which consequently has size $O(k)$, the size of the branches is limited by some constant independent of k. Overall, we consequently get $O(k^3)$ nodes in the tableau. Application of trace transformers takes time independently from k and the size of the tableau. Searching for possible coverings, however, depends on the size of the trace tableau and examines each existing node for a new vertex in the worst case scenario. This results in quadratic time with respect to the size of the tableau, thus we have a worst-case running time of $O(k^6)$ □

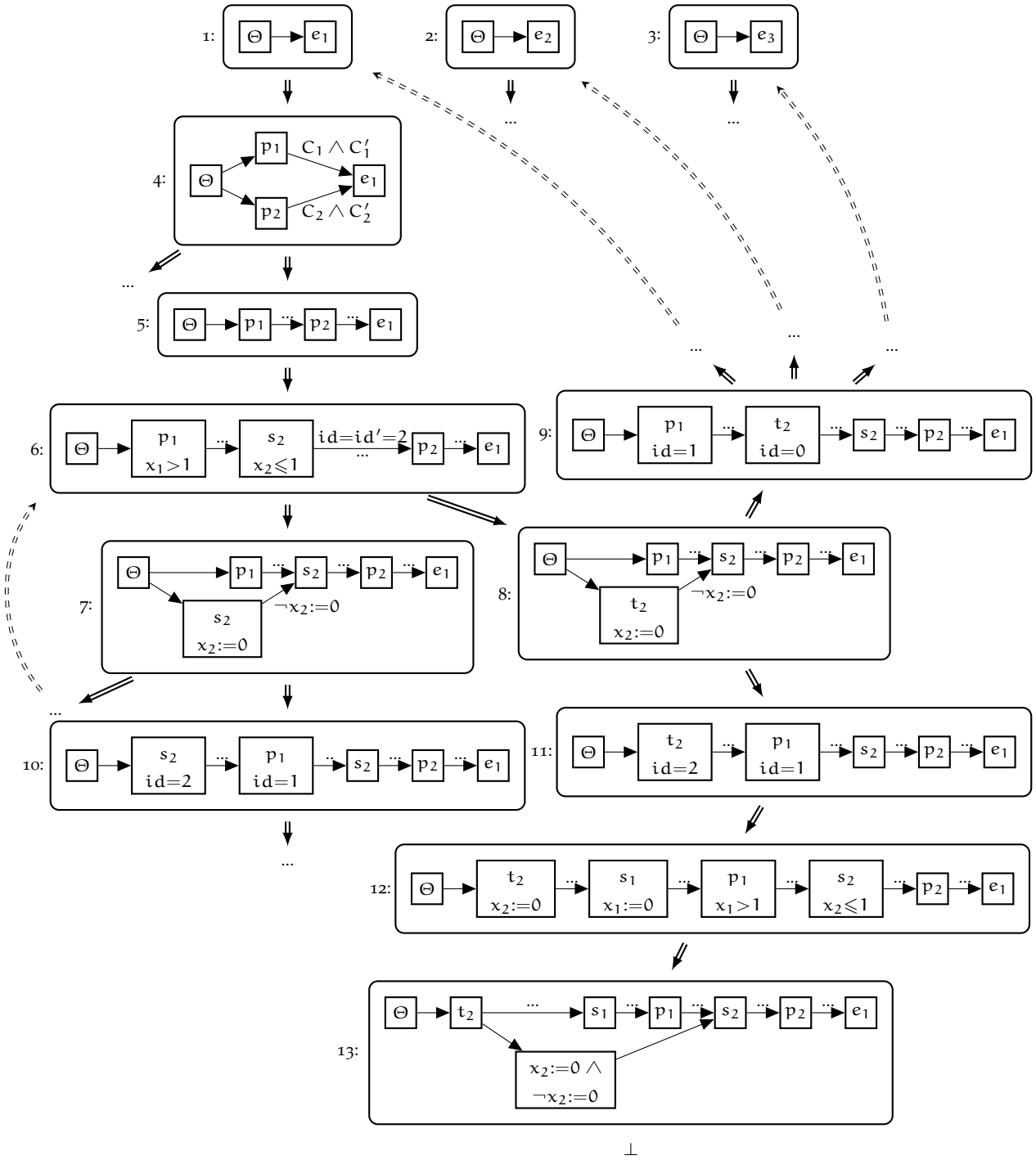


Figure 5.6: Trace tableau for Fischer's protocol (Figure 5.5) with 3 processes. Error conditions are $e_1 \equiv C_1 \wedge C_2$, $e_2 \equiv C_2 \wedge C_3$ and $e_3 \equiv C_1 \wedge C_3$.

6 RELATED WORK

Since the state space explosion is a large obstacle on the way to make timed automata model checking scale to more complex systems, numerous techniques have been introduced to mitigate the problem. Most often, they are based on methods that were originally developed for discrete-time model checking and are applied on basis of the region or zone abstraction of a timed system. In this chapter, we outline how other methods for model checking real-time systems try to overcome the problem of state space explosion and further relate them to our causality-based approach.

Minea [14] presents an approach for applying Partial Order Reduction to networks of timed automata. This method reduces the number of explored interleavings of independent concurrent transitions. The traversed subset of the state space remains representative regarding the verified property by choosing the transitions explored in each state according to predefined rules. This way, the state space never gets fully expanded, which has an especially positive impact on memory consumption. However, networks of timed automata generally have less independent transitions, as they are implicitly synchronized by all clocks advancing at the same pace. Minea solves this by using local-time semantics where delay transitions are not global, but per automaton. The notion of independent transitions is somewhat contrarily related to causality, as it argues that the effect of two independent transitions remains the same irrespective of ordering. This can be seen as the total absence of a causal relation between two independent events. This principle is also captured in the causality-based verification algorithm in form of the `OrderSplit` trace transformer. When applied in compliance with the conditions developed for Partial Order Reduction, `OrderSplit` could realize similar properties and would not explore the two cases explicitly.

Time Petri nets, as introduced by Merlin and Faber [13], provide an alternative to timed automata as system models. A Time Petri net is a Petri net where transitions are associated with an earliest and latest firing time. Transitions must fire in this interval relative to the time they became enabled. Unlike networks of timed automata, all processes are modeled in a single Time Petri net. This effectively requires defining cause-effect relationships in the model at design time. Therefore our causality-based algorithm explores dependencies in networks of timed automata that are an integral part of a Time Petri net from the start. However, just like with the automata-based approach, complex systems with a large number of participating pro-

cesses modeled as Time Petri nets suffer from state space explosion. Methods to combat this issue, like Partial Order Reduction [16], have been studied and applied to Time Petri nets.

Isenberg and Wehrheim [9] extend IC3 to timed automata. This algorithm incrementally computes an inductive invariant for the transition system and property under consideration. To this end, the verified property is strengthened by approximating the states reachable in an increasing number of transitions. If a counterexample to induction is found, the algorithm strengthens the property by adding non-reachability of the counterexample's predecessor to the formula. The new, refined property, however, might now generate another counterexample in an earlier reachability step. The subsequent exclusion of this new counterexample leads to backwards exploration and either terminates in the initial states (when a counterexample trace is found) or eventually excludes this counterexample. The property is successfully strengthened once the set of reachable states does not change when considering one more transition. For timed automata, excluding counterexamples is not trivial, as there can be infinitely many due to the infinite number of clock valuations. Isenberg and Wehrheim therefore infer the zone corresponding to the counterexample to induction and subsequently exclude the whole zone. The backwards exploration starting from a counterexample of induction is strongly related to our causality-based verification algorithm. However, while we start with the hypothesis of (possibly multiple) error traces, IC3 initiates the exploration only on finding a counterexample during approximative forward reachability analysis.

7 CONCLUSION & FUTURE WORK

This thesis originated from the idea to capture causal, human reasoning about time in an automatic proof system for real-time systems modeled as networks of timed automata.

Based on the causality-based verification framework proposed by Andrey Kupriyanov [10], we extended concurrent traces, which are abstractions of sets of traces and the basic building block of causality-based proofs, to timed automata. The implicit synchronization of clocks and advancing nature of time present in these automata necessitates a different approach to satisfiability and emptiness checking of timed concurrent traces. We solved this by defining a constraint system that captures the dependencies between different clock resets and constraints in a given trace and allows for emptiness checking of an important subset of concurrent traces, compactizations, as well as extraction of unsatisfiable subtraces.

In the next step, we defined trace transformers that allow case distinctions about the existence of clock resets in a trace. Building on these, we proposed a causality-based safety model checking algorithm for timed automata. We captured human reasoning about time in this algorithm in the sense that the algorithm observes a timing inconsistency, the effect of a necessary clock reset, and refines the trace by considering all possible causing resets. It is hybrid in the sense that, when no timing inconsistency is present, it resorts to repairing discrete conflicts. Concluding, we demonstrated that our algorithm proves the safety of Fischer's protocol in polynomial space and time.

In future work, we will strive for an implementation of our algorithm in order to evaluate its practical performance. For now, it remains open how well our algorithm will perform compared to tools like Uppaal, as despite the theoretical complexity, the high optimization of these tools should make for an interesting comparison.

Furthermore, the example of Fischer's protocol opens the question whether our algorithm performs equally well on other practically relevant models. Errors in timed systems often depend on timing inconsistencies, so our algorithm should be posed to extract these contradictions efficiently. Comprehensive, empirical analysis of causality-based model checking for real-time systems therefore promises to yield further insight in future work.

There is also the open question of alternative exploration algorithms. While our algorithm is firmly based on backward insertion of clock resets and producing contradictions in the back of the unsatisfiable subtrace, one can easily imagine alternative strategies. This has

interesting implications when considered with regards to the function returning the unsatisfiable subtrace of a given timed concurrent trace, which for sake of generality we assumed to be nondeterministic. In an implementation, it is highly desirable for both aspects to be in tune to achieve termination as soon as possible.

Lastly, an interesting avenue of future work is to extend the properties under consideration. For discrete systems, causality-based verification has also been proposed for liveness properties [12]. When thinking of expanding the proof system to a real-time logic like MTL, however, it is important to note that the model checking problem for MTL is undecidable [15]. Decidability can be attained by restriction to certain fragments of MTL. Finding interesting fragments that are a good fit for causality-based model checking is an exciting topic for future work.

A

TRACE TRANSFORMERS FOR DISCRETE SYSTEMS

Event Split

Given some event a labeled with predicate ψ and an arbitrary predicate ϕ, the $\mathit{EventSplit}(a, \phi, \psi)$ trace transformer considers two alternatives: either a satisfies ϕ, or it does not. Formally:

$\mathit{pre}(\mathit{EventSplit}(a, \phi, \psi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a\}$
- $\lambda_{\mathcal{E}} = \{a \to \psi\}$
- $\mathcal{C} = \lightning = \lambda_{\mathcal{C}} = \emptyset$

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

- $R_1 = \langle \mathcal{E}, \mathcal{C}, \lightning, \{a \to \psi \wedge \phi\}, \lambda_{\mathcal{C}} \rangle$
- $R_1 = \langle \mathcal{E}, \mathcal{C}, \lightning, \{a \to \psi \wedge \neg\phi\}, \lambda_{\mathcal{C}} \rangle$

Conflict Split

Given events a and b labeled with predicates ϕ and ψ respectively, and an arbitrary predicate ϕ, the $\mathit{ConflictSplit}(a, b, \phi, \psi)$ trace transformer considers two alternatives: either a and b coincide in time, or they do not. Formally, we define $\mathit{ConflictSplit}(a, b, \phi, \psi)$ as follows:

$\mathit{pre}(\mathit{ConflictSplit}(a, b, \phi, \psi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$
- $\lambda_{\mathcal{E}} = \{a \to \phi, b \to \psi\}$
- $\mathcal{C} = \lightning = \lambda_{\mathcal{C}} = \emptyset$

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

- $R_1 = \langle \{ab\}, \mathcal{C}, \lightning, \{ab \to \phi \wedge \psi\}, \lambda_{\mathcal{C}} \rangle$
- $R_1 = \langle \mathcal{E}, \mathcal{C}, \lightning \cup \{(a, b)\}, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$

Conflict

Given two events a and b labeled with predicates ϕ_1, ϕ_2 such that $\mathit{unsat}(\phi_1 \wedge \phi_2)$ holds, $\mathit{Conflict}(a, b, \phi_1, \phi_2)$ establishes a conflict relation between the two events. Formally:

$\mathit{pre}(\mathit{Conflict}(a, b, \phi_1, \phi_2)) = \langle \mathcal{E}, \mathcal{C}, \not\downarrow, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$, where:

- $\mathcal{E} = \{a, b\}$
- $\lambda_\mathcal{E} = \{a \to \phi_1, b \to \phi_2\}$ with $\mathit{unsat}(\phi_1 \wedge \phi_2)$
- $\mathcal{C} = \not\downarrow = \lambda_\mathcal{C} = \emptyset$

$\mathit{post}(\mathit{Conflict}(a, b, \phi_1, \phi_2)) = \langle \mathcal{E}, \mathcal{C}, \{(a, b)\}, \lambda_\mathcal{E}, \lambda_\mathcal{C} \rangle$

Event Restriction

The $\mathit{EventRestriction}(a, b, c, \phi, \psi)$ restricts an event b, which is in scope of the causal link (a, c). b has the labeling ψ, the label of the link is ϕ. Formally:

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

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

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

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

Link Restriction

The $\mathit{LinkRestriction}(a, b, c, \phi, \psi)$ trace transformer, given a causal link (a, b) labeled with ψ, which is in scope of causal link (a, c) labeled with ϕ restricts (a, b) with ϕ. Formally we have

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

- $\mathcal{E} = \{a, b, c\}$
- $\mathcal{C} = \{(a, b), (b, c), (a, c)\}$
- $\not\downarrow = \emptyset$

- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \top, c \to \top\}$
- $\lambda_{\mathcal{C}} = \{(a,b) \to \psi, (b,c) \to \top, (a,c) \to \phi\}$

$post(EventRestriction(a,b,c,\phi,\psi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, with

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

Causal Transitivity

The $CausalTransitivity(a,b,c,\phi_1,\psi,\phi_2)$ trace transformer, given two causal links (a,b) and (b,c) labeled with ϕ_1 and ϕ_2 respectively, and an event b labeled with ψ, introduces a new causal link (a,c) with labeling $\phi_1 \vee \psi \vee \phi_2$. Formally:

$pre(CausalTransitivity(a,b,c,\phi_1,\psi,\phi_2)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a,b,c\}$
- $\mathcal{C} = \{(a,b),(b,c)\}$
- $\lightning = \emptyset$
- $\lambda_{\mathcal{E}} = \{a \to \top, b \to \psi, c \to \top\}$
- $\lambda_{\mathcal{C}} = \{(a,b) \to \phi_1, (b,c) \to \phi_2\}$

$post(CausalTransitivity(a,b,c,\phi_1,\psi,\phi_2)) = \langle \mathcal{E}, \mathcal{C}', \lightning, \lambda_{\mathcal{E}}, \lambda'_{\mathcal{C}} \rangle$, with

- $\mathcal{C}' = \{(a,c),(b,c),(a,c)\}$
- $\lambda'_{\mathcal{C}} = \{(a,b) \to \phi_1, (b,c) \to \phi_2, (a,c) \to \phi_1 \vee \psi \vee \phi_2\}$

Conflict Transitivity

The $ConflictTransitivity(a,b,c)$ trace transformer, given a conflict $a \lightning b$ and causal links $(a,b),(b,c)$, derives a new conflict $a \lightning c$. Formally we have

$pre(ConflictTransitivity(a,b,c)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

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

$post(ConflictTransitivity(a,b,c)) = \langle \mathcal{E}, \mathcal{C}, \lightning \cup \{(a,c)\}, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$

Instantiate

The $\mathit{Instantiate}(a, \phi, \psi)$ trace transformer, given some event a in a trace labeled with predicate ϕ, instantiates it with all system transitions that satisfy ϕ. The predicate ψ lets to further restrict the potentially large set of system transitions to the ones satisfying $\phi \wedge \psi$. Formally:

$\mathit{pre}(\mathit{Instantiate}(a, \phi, \psi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$

- $\mathcal{E} = \{a\}$
- $\lambda_{\mathcal{E}} = \{a \to \phi\}$
- $\mathcal{C} = \lightning = \lambda_{\mathcal{C}} = \emptyset$

$\mathit{post}(\mathit{Instantiate}(a, \phi, \psi)) = \{R_0, R_1, ..., R_k\}$, where

- $R_0 = \langle \mathcal{E}, \mathcal{C}, \lightning, \{a \to \phi \wedge \neg\psi\}, \lambda_{\mathcal{C}} \rangle$
- let $\{t_1, ..., t_k\} = \{t \in T \mid \mathit{sat}(t \wedge \phi \wedge \psi)\}$,
  then $R_i = \langle \mathcal{E}, \mathcal{C}, \lightning, \{a \to t_i \wedge \phi \wedge \psi\}, \lambda_{\mathcal{C}} \rangle$

Forward Unrolling

The $\mathit{ForwardUnrolling}(a, b, \phi)$ trace transformer, given events a and b in a trace that cannot follow immediately after another (similar to $\mathit{NecessaryEvent}(a, b, \phi)$, explores all system transitions that can follow a. Formally, given a transition system $\mathcal{S} = \langle \mathcal{V}, \mathcal{T}, \Omega \rangle$, let $\{t_1, ..., t_k\} = \{t \in \mathcal{T} \mid \mathit{sat}(\phi \wedge t)\}$ be the set of transitions that can follow immediately after a. Then:

$\mathit{pre}(\mathit{ForwardUnrolling}(a, b, \phi)) = \langle \mathcal{E}, \mathcal{C}, \lightning, \lambda_{\mathcal{E}}, \lambda_{\mathcal{C}} \rangle$, where:

- $\mathcal{E} = \{a, b\}$
- $\mathcal{C} = \{(a, b)\}$
- $\lightning = \{(a, b)\}$
- $\lambda_{\mathcal{E}} = \{a \to \phi', b \to \neg\phi\}$
- $\lambda_{\mathcal{C}} = \emptyset$

$\mathit{post}(\mathit{ForwardUnrolling}(a, b, \phi)) = \{R_1, ..., R_k\}$, and:

- $R_i = \langle \mathcal{E}', \mathcal{C}', \lightning', \lambda'_{\mathcal{E}_i}, \lambda'_{\mathcal{C}} \rangle$, where:
  - $\mathcal{E}' = \mathcal{E} \cup \{c\}$
  - $\mathcal{C}' = \mathcal{C} \cup \{(a, c), (c, b)\}$
  - $\lightning' = \lightning \cup \{(a, c), (c, b)\}$
  - $\lambda'_{\mathcal{E}_i} = \lambda_{\mathcal{E}} \cup \{c \to t_i\}$
  - $\lambda'_{\mathcal{C}} = \lambda_{\mathcal{C}} \cup \{(a, c) \to \bot, (c, b) \to \top\}$

Backward Unrolling

Similar to $\mathsf{ForwardUnrolling}(a, b, \phi)$, $\mathsf{BackwardUnrolling}(a, b, \phi)$ explores all system transitions that can precede b. Formally, given a transition system $\mathcal{S} = \langle \mathcal{V}, \mathcal{T}, \Omega \rangle$, let $\{t_1, ..., t_k\} = \{t \in \mathcal{T} \mid \mathsf{sat}(t \wedge \neg \phi')\}$ be the set of transitions that can immediately precede b. Then:

$\mathsf{pre}(\mathsf{BackwardUnrolling}(a, b, \phi)) = \mathsf{pre}(\mathsf{ForwardUnrolling}(a, b, \phi))$

$\mathsf{post}(\mathsf{BackwardUnrolling}(a, b, \phi)) = \{R_1, ..., R_k\}$, and:

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

BIBLIOGRAPHY

[1] Rajeev Alur. "Timed Automata." In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 8–22.

[2] Rajeev Alur and David L. Dill. "A Theory of Timed Automata." In: *Theor. Comput. Sci.* 126.2 (Apr. 1994), pp. 183–235.

[3] Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. "Lower and Upper Bounds in Zone-based Abstractions of Timed Automata." In: *Int. J. Softw. Tools Technol. Transf.* 8.3 (June 2006), pp. 204–215.

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems." In: *Hybrid Systems*. 1995.

[5] Patricia Bouyer. "From Qualitative to Quantitative Analysis of Timed Systems." Mémoire d'habilitation. Université Paris 7, Paris, France, Jan. 2009.

[6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. "Handbook of Graph Grammars and Computing by Graph Transformation." In: ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997. Chap. Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach, pp. 163–245.

[7] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. "Handbook of Graph Grammars and Computing by Graph Transformation." In: ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997. Chap. Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach, pp. 247–312.

[8] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic Model Checking for Real-Time Systems." In: *Information and Computation* 111.2 (1994), pp. 193 –244.

[9] Tobias Isenberg and Heike Wehrheim. "Timed Automata Verification via IC3 with Zones." In: *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*. Lecture Notes in Computer Science. 2014, 203–218.

[10] Andrey Kupriyanov. "Causality-based Verification." PhD thesis. Universität des Saarlandes, Saarbrücken, Germany, 2016.

[11] Andrey Kupriyanov and Bernd Finkbeiner. "Causality-Based Verification of Multi-threaded Programs." In: vol. 8052. Aug. 2013, pp. 257–272.

[12] Andrey Kupriyanov and Bernd Finkbeiner. "Causal Termination of Multi-threaded Programs." In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 814–830.

[13] Philip M. Merlin and David Farber. "Recoverability of Communication Protocols–Implications of a Theoretical Study." In: *Communications, IEEE Transactions on* 24 (Oct. 1976), pp. 1036 – 1043.

[14] Marius Minea. "Partial Order Reduction for Model Checking of Timed Automata." In: *Proceedings of the 10th International Conference on Concurrency Theory*. CONCUR '99. London, UK, UK: Springer-Verlag, 1999, pp. 431–446.

[15] Joël Ouaknine and James Worrell. "Some Recent Results in Metric Temporal Logic." In: *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*. FORMATS '08. Saint Malo, France: Springer-Verlag, 2008, pp. 1–13.

[16] Tomohiro Yoneda and Bernd-Holger Schlingloff. "Efficient Verification of Parallel Real-Time Systems." In: *Form. Methods Syst. Des.* 11.2 (Aug. 1997), pp. 187–215.