Causality-based Verification of Multi-threaded Programs ⋆

Andrey Kupriyanov and Bernd Finkbeiner

Universität des Saarlandes, Saarbrücken, Germany

Abstract. We present a new model checking procedure for concurrent systems against safety properties such as data races or atomicity violations. Our analysis sidesteps the state space explosion problem by inferring causal dependencies for concurrent traces instead of searching over a space of reachable states, and can be understood as an interplay between local trace inference and termination analysis based on causal loops. Local trace inference introduces new actions anywhere in the trace if they causally follow from the context. Our procedure terminates if we either find a complete error trace or the whole space of potential error traces is covered by causal loops. The causality-based verification of multi-threaded programs can be dramatically faster than the standard state space traversal. In particular, we show that the complexity of verifying multi-threaded programs with locks reduces from exponential to polynomial.

1 Introduction

Causality, the relationship between two events where the first event is recognized as a necessary requirement for the occurrence of the second, is a key concept in our understanding of complex computer systems. Processes in a concurrent system proceed independently until they establish causal dependence through synchronization. Modeling formalisms like Petri nets [10] explicitly capture the causal dependence between transitions through the flow of tokens.

Not surprisingly, causality has also proven useful in the automatic verification of concurrent systems. Petri net unfoldings [4], for example, avoid a total temporal ordering of the events and instead unwind the causality relation. In partial order reduction [5], causally independent events are forced into a fixed temporal order. Traditionally, however, the role of causality in automatic verification has always been secondary compared to the state-based reachability analysis: in Petri net unfoldings, we unwind the causality relation forward until we are certain that no more reachable markings can be found; in partial order reduction, we avoid the exploration of computation paths that lead to the same states that have already been seen on some other path with a different ordering of the causally independent events.

⋆ This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, `www.avacs.org`).

In this paper, we upgrade the role of causality in automatic verification to that of a first-class citizen. The approach is based on the observation that reaching an error state often causally depends on a small number of certain key events, which, in a correct system, contradict each other. For example, a violation of mutual exclusion between two processes requires that previously both processes have entered the critical section and, during one of these events, the other process was already in the critical section. Intuitively, our verification procedure identifies such necessary steps on a trace from initial to error states and then either completes the partial trace into a full error trace or proves that no such completion exists.

In our algorithm, we capture the causal dependencies as Mazurkiewicz-style concurrent traces. Starting with a default initial trace, which only captures the initial and error states, we capture, step by step, more dependencies by applying special graph transformations, which we call *causal transitions*: the causal transitions include, for example, the *necessary action* transition, which uses Craig interpolation to find a necessary intermediate action. A full exploration of the causal dependencies leads to an in general infinite tree, which we call the *causal trace unwinding*. If all branches of the causal trace unwinding are either contradictory (meaning that the causal requirements of reaching the error cannot be satisfied) or infinite (meaning that no finite number of actions suffices to reach the error), we can conclude that the error is, in fact, unreachable. The verification algorithm builds finite prefixes of the causal trace unwinding and terminates as soon as the two conditions can be established for the full tree.

The causality-based verification of multi-threaded programs can be dramatically faster than the standard state space traversal. In the paper, we demonstrate this effect for multi-threaded programs with locks. It turns out that our algorithm verifies the most general class of these programs in polynomial time. This answers an open question originally posed by Alexander Malkis [7].

The remainder of the paper is structured as follows. After a brief discussion of related work, we consider a motivating example from the class of multi-threaded programs with locks in Section 2. We discuss the necessary preliminaries in Section 3 and define the central structure of our approach, causal trace unwinding, in Section 4. The causality-based verification algorithm, which allows us to explore only a finite prefix of the unwinding, is described in Section 5. Finally, in Section 6 we show how our verification algorithm settles the open question regarding the verification of multi-threaded programs with locks, reducing the complexity from exponential to polynomial.

Related work. Causality-based verification can be understood as a generalization of standard model checking [1], because the next-state relation usually explored in model checking captures the causal dependencies between successor states: a trace from some arbitrary non-error state to an error state can only exist if there is a trace from one of the state's successors. However, it is usually easy to obtain additional elements of the causality relation, for example based on a cheap analysis of the control flow graph. Such additional elements are exploited

by our procedure, but not by standard model checking based on a forward or backward traversal of the state space.

Our method is related to approaches based on partial orders, such as partial order reduction [5], Mazurkiewicz traces [9], and Petri net theory [10]. Similar to causality-based verification, these approaches exploit the independence that results from the combination of separate processes. Unlike these approaches, we do not require, however, that the system is given a-priori as a partial order. Our rules extract causal dependencies from the system description and use such dependencies to contradict the existence of an error trace. Finally, causality-based verification is a tableau-based decision procedure, related to the tableau-based approaches for modal and temporal logics [8].

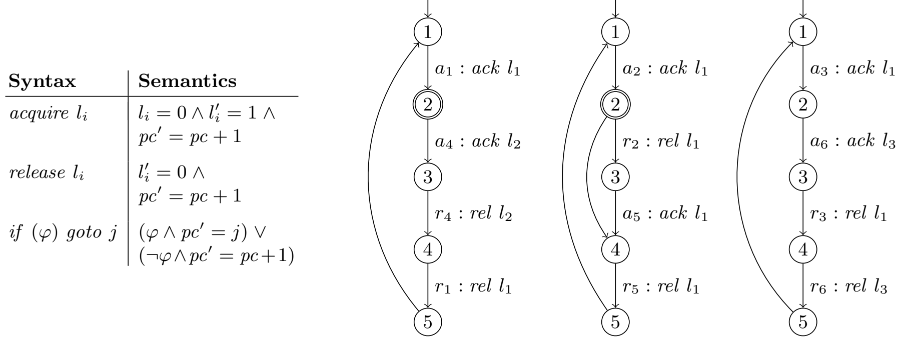
2 Motivating Example

As a motivating example we consider the class of multithreaded programs with critical sections protected by shared lock variables, which is described in [7]. A program in this class consists of n threads, executing in the interleaved fashion, and m shared boolean lock variables. Each thread contains some finite number of critical sections, protected by the "*acquire* lck_i" and "*release* lck_i" statements for one of the lock variables. Critical sections may be arbitrarily nested or intersected, and a thread may have an arbitrary control structure via the use of "*if* (φ) *goto* j" statements, where φ is any formula over the shared variables. The only restriction for a correct program is that the control flow may enter one of the critical sections for the lck_i variable only via the "*acquire* lck_i" statement, and may exit it either by jumping to another critical section for the same lock variable, or by executing the "*release* lck_i" statement. The syntax and semantics of such programs are shown in the left part of Figure 1; they can be used to analyze systems with built-in "test-and-set" primitive. In the following we consider the example program depicted on the right of Figure 1; we want to verify that threads 1 and 2 cannot be simultaneously at their critical sections 2, protected by lock l_1.

Our algorithm operates on a causal trace unwinding, where vertices are labeled with abstract traces. In Figure 2 we depict the unwinding in the center, and the labels of its vertices on the left or on the right from a corresponding vertex (we do not draw some trace edges, which follow from transitivity).

Step 1: We start with an unwinding, containing a single vertex 1, and labeled with the abstract trace representing all concrete traces from initial state to error state (initial action i is labeled with $pc'_1 = 1 \wedge pc'_2 = 1 \wedge pc'_3 = 1$, and error action e is labeled with $pc_1 = 2 \wedge pc_2 = 2$).

Step 2: We check whether the abstract trace of vertex 1 is concretizable. It is not, for example because of the of the conflict $pc'_1 = 1 \not\downarrow pc_1 = 2$ between actions i and e. We conclude that in between of initial and error actions a *necessary action*, characterized by the transition predicate $pc_1 \neq 2 \wedge pc'_1 = 2$, should happen. There is only one system transition, a_1, satisfying this predicate, and we introduce new vertex 2 in the unwinding, labeled with the abstract trace



Syntax	Semantics
acquire l_i	$l_i = 0 \wedge l_i' = 1 \wedge$ $pc' = pc + 1$
release l_i	$l_i' = 0 \wedge$ $pc' = pc + 1$
if (φ) *goto* j	$(\varphi \wedge pc' = j) \vee$ $(\neg\varphi \wedge pc' = pc + 1)$

Fig. 1. General class of multithreaded programs with binary locks. *Left:* Syntax and semantics. *Right:* Example system consisting of 3 threads with critical sections over 3 lock variables. Initial state vector is $(1, 1, 1)$, and error state vector is $(2, 2, _)$.

where transition a_1 is inserted. We require that thread 1 does not leave location 2, and mark the edge $a_1 \to e$ with the predicate $pc_1 = 2$.

Step 3 is similar to step 2: there is a conflict $pc_2' = 1 \lightning pc_2 = 2$, and a single necessary action, a_2, satisfying the transition predicate $pc_2 \neq 2 \wedge pc_2' = 2$. We introduce new vertex 3, where actions a_1 and a_2 are concurrent: they are both necessary, but the order of their occurrence is unspecified.

Step 4: We try to linearize the abstract trace from vertex 3; it contains two concurrent actions, and we choose an arbitrary order between them, for example a_1 before a_2. The linear trace contains the conflict $l' = 1 \lightning l = 0$ (a_1 has the postcondition $l = 1$, while a_2 has the precondition $l = 0$). But, because the order between a_1 and a_2 is not dictated by the abstract trace, we make an *order split*, considering both alternatives (vertices 4 and 5).

Step 5: We consider only vertex 5 from the previous step; vertex 4 is analyzed analogously. The conflict $l' = 1 \lightning l = 0$ between a_1 and a_2 is now dictated by the trace; so a new necessary action, satisfying the predicate $l \neq 0 \wedge l' = 0$ should be inserted. We instantiate this abstract action with all concrete actions, satisfying the predicate, namely r_1, r_2, r_3, and r_5. Here, for the picture clarity, we show only vertices 6 (with r_1) and 7 (with r_2).

Step 6: Consider first the trace of vertex 6: it contains a contradiction, namely action r_1 (labeled with $pc_1 = 4 \wedge pc_1' = 5 \wedge l' = 0$) lies in the scope of the edge $a_1 \to e$ (labeled with $pc_1 = 2$), and their labels are unsatisfiable together. Thus, we close this branch as contradictory.

Step 7: For each leaf vertex we try to find whether a "similar" concurrent trace was already encountered before. For the case of vertex 7, its label is, indeed, similar to the label of vertex 1: we can find a mapping from all nodes and edges of the latter to the nodes and edges of the former. More precisely, we can map node i to node i, and node e to node r_2. Moreover, the label of node r_2, which equals to $pc_2 = 2 \wedge pc_2' = 3 \wedge l' = 0 \wedge pc_1 = 2$ due to the restriction from the edge $a_1 \to e$, implies the label of node e. Thus, the trace of vertex 7 is more

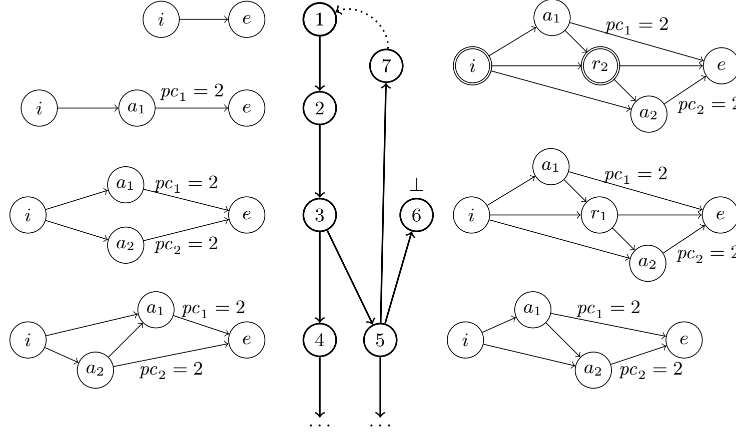


Fig. 2. First steps of the causal trace unwinding for the example multi-threaded program with binary locks.

restrictive than the trace of vertex 1, and we can *cover* vertex 7 by vertex 1. Also, there is node a_2 on the right of r_2, which the covering "forgets". The path $1 \to 2 \to 3 \to 5 \to 7 \to 1$ in the unwinding constitutes a *causal loop*: as long as we follow the loop, we keep introducing new and new actions a_2 on the right.

Continuing the process, we would find out that all leaf nodes in the trace unwinding are either contradictory, as in step 6, or are covered by such causal loops as in step 7. This implies that our system is correct, because any possible error trace would have infinite length. We will return to the verification of multi-threaded programs with locks in Section 6.

3 Preliminaries

Transition Systems. We consider concurrent systems described in some first-order assertion language. For a set of variables $\mathcal{V}$, we denote by $\Phi(\mathcal{V})$ the set of first-order formulas over $\mathcal{V}$. For each variable $x \in \mathcal{V}$ we define a primed variable $x' \in \mathcal{V}'$, which denotes the value of x in the next state. We call formulas from the sets $\Phi(\mathcal{V})$ and $\Phi(\mathcal{V} \cup \mathcal{V}')$ *state predicates* and *transition predicates*, respectively.

A *transition system* is a tuple $\mathcal{S} = \langle \mathcal{V}, T, \mathit{init}, \mathit{error} \rangle$ where $\mathcal{V}$ is a finite set of system variables; $T \subseteq \Phi(\mathcal{V} \cup \mathcal{V}')$ is a finite set of system transitions; $\mathit{init} \in \Phi(\mathcal{V})$ and $\mathit{error} \in \Phi(\mathcal{V})$ are state predicates, characterizing initial and error states.

A *state* of $\mathcal{S}$ is a valuation of system variables $\mathcal{V}$. We call an alternating sequence of states and transitions $s_0, t_1, s_1, t_2, \ldots, t_n, s_n$ a *trace*, if $\mathit{init}(s_0)$ holds, and for all $1 \leq i \leq n$, $t_i(s_{i-1}, s_i)$ holds. We call a trace $s_0, t_1, s_1, t_2, \ldots, t_n, s_n$ an *error trace* if it ends in some error state, i.e. the predicate $\mathit{error}(s_n)$ holds. We say that the system is *safe* if there does not exist any error trace for that system; otherwise the system is *unsafe*. For a system $\mathcal{S}$ we denote the set of its traces by $\mathcal{L}(\mathcal{S})$, and the set of its error traces by $\mathcal{L}_e(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S})$.

Transition systems are well suited for the representation of multi-threaded programs with interleaving semantics: in this case the set of system transitions is simply a union of transitions of individual processes.

Graph Transformations. We follow [2, 3] and use the so-called *single-pushout (SPO)* and *double-pushout (DPO)* approaches to describe graph transformations. All graph transformations that we use are non-erasing and lie at the intersection of both approaches; the definitions below are adapted from [2].

A *graph* is a tuple $G = \langle N, E \rangle$, where N is a set of nodes, and $E \subseteq N \times N$ is a set of edges. The source and target functions $s, t : E \to N$ map each edge to its first and second component, respectively.

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

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

4 Causal Trace Unwindings

4.1 Concurrent Traces

We follow the theory of Mazurkiewicz traces, and define concurrent traces through their *dependence graphs*. A *concurrent trace* is a labeled, directed, acyclic graph $A = \langle N, E, \nu, \eta \rangle$, where $\langle N, E \rangle$ is a graph with nodes N, called *actions*, and edges E; $\nu : N \to \Phi(V \cup V')$, $\eta : E \to \Phi(V \cup V')$ are labelings of nodes and edges with transition predicates. We denote the set of concurrent traces by $\mathbb{A}$.

A concurrent trace describes a set of system traces. For a particular concurrent trace its actions specify which transitions should necessarily occur in a system trace, while its edges represent the (partial) ordering between such transitions and constraint the intermediate ones.

Trace language. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$, the *language* of a concurrent trace $A = \langle N, E, \nu, \eta \rangle$ is defined as a set $\mathcal{L}(A)$ of system traces such that for each trace $s_0, t_1, s_1, t_2, \ldots, t_n, s_n \in \mathcal{L}(A)$ there exists an injective mapping $\sigma : N \to \{t_1, \ldots, t_n\}$ such that:

1. for each action $a \in N$ and $t_i = \sigma(a)$ the formula $\nu(a)\big(s_{i-1}, s_i\big)$ holds.
2. for each edge $e = (a_1, a_2) \in E$, and $t_i = \sigma(a_1)$, $t_j = \sigma(a_2)$, we have that a) $i < j$, and b) for all $i < k < j$, the formula $\eta(e)\big(s_{k-1}, s_k\big)$ holds.

We call a concurrent trace $A = \langle N, E, \nu, \eta \rangle$ *contradictory* if some of its actions is labeled with an unsatisfiable predicate, i.e. if there exists $n \in N$ such that $\nu(n)$ implies $\perp$. Obviously, the language of such a trace is empty.

Trace inclusion. For any two concurrent traces $A = \langle N, E, \nu, \eta\rangle$ and $A' = \langle N', E', \nu', \eta'\rangle$ we define the *trace inclusion* relation $\subseteq$ as follows: $A \subseteq A'$ iff

1. there exists a graph morphism $\lambda = \langle \lambda_N : N' \to N, \lambda_E : E' \to E\rangle$.
2. for all $n' \in N'$. $\nu(\lambda_N(n')) \implies \nu'(n')$.
3. for all $e' \in E'$. $\eta(\lambda_E(e')) \implies \eta'(e')$.

Proposition 1. *if* $A \subseteq A'$ *then* $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.

We write $A \subseteq_\lambda A'$, if trace inclusion holds for a particular graph morphism λ. Let λ^N be the image of λ_N: $\lambda^N = \{n \in N \mid (n', n) \in \lambda_N\}$. We call the trace inclusion $A \subseteq_\lambda A'$ *left-forgetful* (resp. *right-forgetful*), if for all $n \in \lambda^N$ there exists $n_\times \in N \setminus \lambda^N$ such that $(n_\times, n) \in E$ (resp. $(n, n_\times) \in E$). We call the trace inclusion *forgetful* if it is either left- or right-forgetful. Intuitively, when $A \subseteq_\lambda A'$ is a forgetful trace inclusion, we "forget" some action on the left or on the right when moving from A to A'.

Our intention is to find causal consequences from the information about error traces, represented in the form of concurrent traces. For that purpose we start with a single concurrent trace, containing two actions: initial action i, marked with $init'$, and error action e, marked with *error*, connected with an unrestricted edge. The marking ensures that all possible error traces are preserved.

Initial Abstraction. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error\rangle$ we define $InitialAbstraction(\mathcal{S})$ as a concurrent trace $A = \langle N, E, \nu, \eta\rangle$, where $N = \{i, e\}$, $E = \{(i, e)\}$, $\nu = \{(i, init'), (e, error)\}$, $\eta = \{((i, e), true)\}$.

Proposition 2. $\mathcal{L}_e(\mathcal{S}) \subseteq \mathcal{L}(InitialAbstraction(\mathcal{S}))$.

Trace Productions. We lift graph morphisms to traces with the same meaning (mappings for nodes and edges of one trace to those of another), and call them *trace morphisms*. We generalize graph productions to concurrent traces: a *trace production* $\tau : (L \xrightarrow{r} R)$, where L, R are concurrent traces and r is a trace morphism, describes a transformation of trace L into trace R. The graphical part is transformed by the corresponding graph production, and labels are transformed by the operations of boolean algebra. Formally trace productions can be described as graph productions on *attributed graphs*; for details we refer the interested reader to [3], pp. 284-288. In the following we denote the set of trace productions by Π.

4.2 Causal Transitions

Starting from the initial abstraction, we find, step by step, further causal dependencies. For this purpose we introduce in the following special graph productions, which we call *causal transitions*:

Causal Transition. For a given transition system $\mathcal{S}$, a *causal transition* $\tau : \{\tau_1, \ldots, \tau_n\}$ is a set of trace productions $\tau_i : (L \xrightarrow{r_i} R_i)$, where all productions share the same left-hand side L; we will denote L by $\tau_\triangleleft$, and call transition *premise*. We say that causal transition τ is *sound* if the condition below holds:

$$\forall A \in \mathbb{A} \, . \, A \subseteq_m \tau_\triangleleft \implies \mathcal{L}(A) \subseteq \bigcup_{\tau_i \in \tau} \mathcal{L}\big(\tau_i^m(A)\big)$$

The above condition says that if the transition premise $\tau_{\lhd}$ can be matched to some concurrent trace A, then the application of the transition should preserve all possible concrete traces, contained in A. Please note, that causal transitions can be interpreted both operationally (as transformations of concurrent traces), and logically (as language inclusion); we employ both interpretations. In the following we denote the set of causal transitions by Δ.

Below we describe some examples of causal transitions, shown in Figure 3.

Order Split (Figure 3a). The *order split* causal transition considers alternative interleavings of two previously concurrent events.

Action Split (Figure 3b). The *action split* causal transition, given some action a in the trace, and a transition predicate ψ, considers two alternatives: either a satisfies ψ or not.

Transitivity (Figure 3c). The *transitivity* causal transition, given two sequential edges $a \to b$ and $b \to c$, allows to introduce edge $a \to c$, which follows from transitivity, and label it with the disjunction of the constraints in its scope.

Necessary Action (Figure 3d). The *necessary action* causal transition, given two ordered actions a and b in a concurrent trace, and a transition predicate ϕ, such that the label of a implies ϕ', and the label of b implies $\neg\phi$, i.e. there is a contradiction between these actions (a "ends" in the region ϕ, while b "starts" in the region $\neg\phi$), introduces a new "bridging" action x in between. The predicate ϕ may be obtained by Craig interpolation between the labels of a and b. The application condition for this causal transition ensures that there is no other action y in the trace already, that could play the role of x.

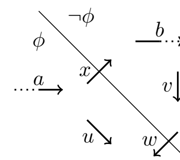


Figure 3d shows three causal transitions, which have the same left-hand-side L, but different right-hand sides R, R_{first}, and R_{last} (separated by vertical bars in the figure). In R we simply insert action x in the concurrent trace. In R_{first} (resp. R_{last}) we require, additionally, that action x is the first (resp. last) action, that crosses the boundary between ϕ and $\neg\phi$ (see picture on the right). This can be achieved by marking the corresponding edge with the predicate $\neg(\phi \wedge \neg\phi') = \neg\phi \vee \phi'$; but we can easily strengthen this requirement. Indeed, suppose we want action x to be the last in the sequence of possible necessary actions; the only actions allowed by the above predicate are of the type u ($\phi \wedge \phi'$), v ($\neg\phi \wedge \neg\phi'$), or w ($\neg\phi \wedge \phi'$). But, if an action of type u or w happens, then an action of type x becomes necessary again, and it is not allowed: a contradiction. Thus, we can safely allow only actions of type v to happen, and strengthen the above predicate to $\neg\phi \wedge \neg\phi'$.

Action/Edge Restriction (Figures 3e, 3f). The *action restriction* (resp. *edge restriction*) causal transition allows us to restrict the label of an action (resp. edge), if it happens to be in the scope of some edge (for example, as a result of *order split* application).

Forward/Backward Unrolling (Figures 3g, 3h). The *forward unrolling* (resp. *backward unrolling*) causal transition, given two actions in a concurrent trace, which cannot follow immediately one after another and do not have any other actions in between, unrolls the transition relation one step forward or

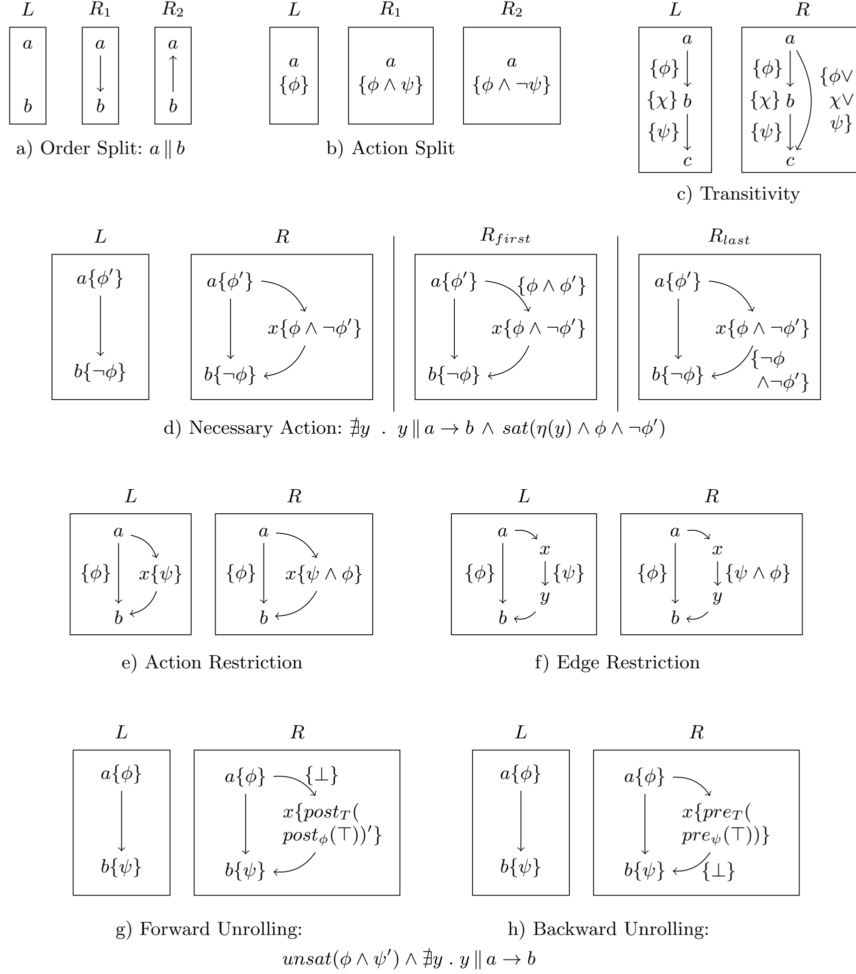


Fig. 3. Examples of causal transitions

backward. Let $\phi_{[\mathcal{V}/\mathcal{V}']}$ denote the substitution of $\mathcal{V}'$ variables in formula ϕ by corresponding variables $\mathcal{V}$. Then $post_\tau(\phi) = \big(\exists_{\mathcal{V}}\ \phi(\mathcal{V}) \wedge \tau(\mathcal{V} \cup \mathcal{V}')\big)_{[\mathcal{V}/\mathcal{V}']}$ is a post-image of ϕ with respect to transition relation τ, while $pre_\tau(\phi) = \exists_{\mathcal{V}'}\ \tau(\mathcal{V} \cup \mathcal{V}') \wedge \phi(\mathcal{V})_{[\mathcal{V}'/\mathcal{V}]}$ is a pre-image of ϕ with respect to τ. For the case of forward unrolling, to calculate the label of the newly introduced action x, we first compute the post-image of the preceding action a, and then the post-image of the result with respect to the whole transition relation T (treated here, by abuse of notation, as a disjunction of all transitions from T). The label $\bot$ on the edge $a \to x$ ensures that there are no other actions between a and x. Calculations for the case of backward unrolling are done similarly.

Proposition 3. *The defined above causal transitions are sound.*

4.3 Causal Trace Unwindings

Causal Trace Unwinding. For a transition system $\mathcal{S}$, we define a *(causal) trace unwinding* as a tuple $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$, where:

- (V, F) is a directed tree with vertices V, root vertex $v_0 \in V$, and edges F. Vertices are partitioned into internal vertices and leaves: $V = V_N \uplus V_L$, $V_N = \{v \in V \mid \exists (v, v') \in F\}$, $V_L = \{v \in V \mid \nexists (v, v') \in F\}$.
- $\gamma : V \to \mathbb{A}$ is a labeling of vertices with concurrent traces.
- $\delta : F \to \Pi$ is a labeling of edges with trace productions. We require that for all edges with the same source v, the labeling productions have the same left-hand side. Thus, we have an induced labeling of internal vertices $v \in V_N$ with causal transitions: $\delta(v) = \{\delta((v, v')) \mid (v, v') \in F\}$.
- λ is a labeling of internal vertices with trace morphisms: $\forall v \in V_N \,.\, \lambda(v) : \delta(v)_\triangleleft \to \gamma(v)$.

A trace unwinding is said to be *correct* if it satisfies the following criteria:

1. $\mathit{InitialAbstraction}(\mathcal{S}) \subseteq \gamma(v_0)$.
2. for all internal vertices $v \in V_N$ we have: a) $\delta(v)$ is sound, b) $\gamma(v) \subseteq_{\lambda(v)} \delta(v)_\triangleleft$, and c) for all $(v, v') \in F$ it holds that $\delta\big((v, v')\big)^{\lambda(v)}\big(\gamma(v)\big) \subseteq \gamma(v')$.

A trace unwinding is a tree, which can be seen as an unwinding of the trace causality relation. The label $\gamma(v)$ of the vertex v represents all possible error traces for that vertex; the first condition above ensures that the root vertex v_0 contains all error traces of the given system. The second condition guarantees the applicability of the causal transition $\delta(v)$ of a vertex v to its label $\gamma(v)$ and full exploration of the causal transition consequences, thus preserving the set of concrete traces. Indeed, we have:

$$\gamma(v) \subseteq_{\lambda(v)} \delta(v)_\triangleleft \implies \mathcal{L}\big(\gamma(v)\big) \subseteq \bigcup_{(v,v') \in F} \mathcal{L}\Big(\delta\big((v, v')\big)^{\lambda(v)}\big(\gamma(v)\big)\Big) \subseteq \bigcup_{(v,v') \in F} \mathcal{L}\big(\gamma(v')\big)$$

We call *causal path* a finite or infinite sequence $v_0, v_1, v_2, \ldots$ of vertices, starting from the root v_0, such that for all $i \geq 0$, $(v_i, v_{i+1}) \in F$. We call a causal path *contradictory* if it is finite and ends in a vertex labeled with a contradictory trace. We call a causal path *unbounded* if it is infinite and the number of actions in the labeling of its vertices increases beyond any bound: for any $n \in \mathbb{N}$ there exists $i \geq 0$ such that $|\gamma(v_i)| > n$.

Theorem 1 (Soundness of Trace Unwinding). *If there exists a correct causal trace unwinding for a transition system $\mathcal{S}$, where every causal path is either contradictory or unbounded, then $\mathcal{S}$ is safe.*

5 Causality-based Verification Algorithm

The causal trace unwinding, described in the preceding section, is easy to construct, but in most cases it will be infinite. In this section we provide an algorithm that explores only a finite prefix of an infinite unwinding and, based on that prefix, establishes the desired properties for the whole unwinding.

The idea behind the finite unwinding prefix is simple: as soon as we encounter a new vertex, labeled with some concurrent trace we have seen before, we would like to cut the unwinding at that vertex and loop back. There are two problems with this simple-minded approach. First, the exact match of one trace to another, like in step 7 of the motivating example, is rarely achievable. We solve this problem by tracking for each vertex the most general trace, sufficient to repeat all causal transitions in the subtree of that vertex; in that way we significantly relax the matching requirement. Second, we should ensure that every infinite path in the unwinding is unbounded; we achieve that by requiring that a forgetful trace inclusion holds between the trace of the leaf vertex and the most general trace of the vertex where the back loop leads to. We call such a finite unwinding prefix *causal trace tableau*, and such back loops *covering*.

Causal Trace Tableau. A *(causal) trace tableau* for a transition system $\mathcal{S}$ is a tuple $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$ where:

- $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$ is a causal trace unwinding.
- $\rightsquigarrow : V_L \nrightarrow V_N$ is a partial *covering* function; for $(v, v') \in \rightsquigarrow$ we call v a *covered* vertex, and v' a *covering* vertex.
- $\alpha : V \to \mathbb{A}$ is a labeling of vertices with (abstract) concurrent traces.
- μ and σ are labelings of vertices with trace morphisms: $\mu(v) : \delta(v)_{\triangleleft} \to \alpha(v)$ and $\sigma(v) : \alpha(v) \to \gamma(v)$ such that $\sigma(v) \circ \mu(v) = \lambda(v)$.

We call a trace tableau $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$ *complete* if all its leaf vertices are either contradictory or covered. We call it *correct* if Υ is correct and, additionally:

1. for all vertices $v \in V$ we have $\gamma(v) \subseteq_{\sigma(v)} \alpha(v) \subseteq_{\mu(v)} \delta(v)_{\triangleleft}$.
2. for all $(v, v') \in F$ we have $\delta\big((v, v')\big)^{\mu(v)}\big(\alpha(v)\big) \subseteq \alpha(v')$.
3. for all $(v, v') \in \rightsquigarrow$ we have that $\alpha(v) \subseteq_{\mu(v)} \alpha(v')$ is a forgetful trace inclusion.
4. for all $v \in V_L$ such that $\gamma(v)$ is contradictory, $\alpha(v)$ is also contradictory.

Theorem 2 (Soundness of Trace Tableau). *If there exists a correct and complete causal trace tableau for a transition system $\mathcal{S}$, then $\mathcal{S}$ is safe.*

Theorem 3 (Completeness of Trace Tableau). *If a transition system $\mathcal{S}$ with finite-state quotient is safe, then there exists a correct and complete causal trace tableau for $\mathcal{S}$.*

Our causality-based verification algorithm (see Algorithm 1), operates on the trace tableau defined above. Each vertex v in the tableau is labeled with two concurrent traces: a *concrete* trace $\gamma(v)$, and an *abstract* trace $\alpha(v)$; we always have that $\gamma(v) \subseteq_{\sigma(v)} \alpha(v)$. Initially the unwinding contains only the root v_0, labeled with the concrete trace *InitialAbstraction*$(\mathcal{S})$. Concrete label $\gamma(v)$ is obtained as a result of a chain of applications of causal transitions on the path from v_0 to v. Abstract label $\alpha(v)$, on the other hand, represents conditions, sufficient to repeat all unwinding steps in the subtree, originating at v; it is obtained by propagating up the premises of causal transitions, applied at the subtree vertices.

At each iteration of the algorithm main loop we select some vertex v from the queue Q of unexplored tableau leaves. First, we try to cover v by some

Algorithm 1: Causality-based Verification

Input : Transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$
Output: **safe/unsafe**
Data: Trace tableau $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$, where $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$, queue $Q \subseteq V_L$, premise of last causal transition $\tau_\triangleleft \in \mathbb{A}$, trace morphism $\xi : \tau_\triangleleft \to \mathbb{A}$

begin
- set $V \longleftarrow \{v_0\}$, $\gamma(v_0) \longleftarrow \mathit{InitialAbstraction}(\mathcal{S})$
- set $Q \longleftarrow \{v_0\}$, all of $\{F, \rightsquigarrow, \alpha, \mu, \sigma, \delta, \lambda\} \longleftarrow \emptyset$
- **while** Q *not empty* **do**
  - take some v from Q
  - **if** $\exists\, v' \in V_N,\ \sigma' : \alpha(v') \to \gamma(v)\ .\ \gamma(v) \subseteq_{\sigma'} \alpha(v')$ *is forgetful* **then**
    - add (v, v') to $\rightsquigarrow$
    - set $\delta(v)_\triangleleft \longleftarrow \alpha(v')$, $\tau_\triangleleft \longleftarrow \alpha(v')$, $\xi \longleftarrow \sigma'$
  - **else**
    - set $L \longleftarrow \mathit{Linearize}(\gamma(v))$
    - **if** $\mathit{Concretizable}(L)$ **then**
      - **return** ***unsafe***
    - **else**
      - set $\langle \tau_\triangleleft, \xi \rangle \longleftarrow \mathit{Refine}(v, L)$
  - put children of v into Q
  - $\mathit{PropagateUp}(v, \tau_\triangleleft, \xi)$
- **return** ***safe***

In: vertex v, linear trace $L = \langle N, E, \nu, \eta \rangle$
Out: $\langle$premise $\tau_\triangleleft$, trace morphism $\xi\rangle$

begin
- $\langle N' \subseteq N, E' \subseteq E \rangle \longleftarrow \mathit{ExtractConflict}(L)$
- **if** $\exists\, o_1, o_2 \in N' \cup E'\ .\ o_1 \| o_2$ **then**
  - $\mathit{OrderSplit}(o_1, o_2)$
- **else**
  - **switch** $|N'|$ **do**
    - **case** *1*
      - $\mathit{Contradiction}(v, n_1)$
    - **case** *2*
      - $\phi = \mathit{Interpolate}(\eta(n_1)\ ;\ \eta(n_2)')$
      - $\mathit{NecessaryAction}(v, n_1, n_2, \phi)$
    - **otherwise**
      - $\phi = \mathit{Interpolate}(\eta(n_1) \wedge \ldots \ldots \wedge \eta(n_{k-1})^{k-1}\ ;\ \eta(n_k)^k)$
      - $\mathit{ActionSplit}(v, n_{k-1}, \phi)$
- **return** $\langle$*premise* $\tau_\triangleleft$ *of used causal transition, trace morphism* $\xi : \tau_\triangleleft \to L\rangle$

Function Refine

In: vertex v, premise $\tau_\triangleleft$, trace morphism $\xi : \tau_\triangleleft \to \gamma(v)$

begin
- **if** $\nexists \chi = \langle \chi_N, \chi_E \rangle : \tau_\triangleleft \to \alpha(v)\ .\ \xi = \sigma \circ \chi$ **then**
  - **foreach** $o \in \gamma(v)\ .\ \exists\, o' \in \tau_\triangleleft\ .\ o = \xi(o') \wedge \nexists\, o'' \in \alpha(v)\ .\ o = \sigma(o'')$ **do**
    - add o' to $\alpha(v)$, and (o', o) to $\sigma(v)$
- let $\chi = \langle \chi_N, \chi_E \rangle : \tau_\triangleleft \to \alpha(v)\ .\ \xi = \sigma \circ \chi$
- **if** $\alpha(v) \not\subseteq_\chi \tau_\triangleleft$ **then**
  - **foreach** $n\ .\ \eta(\chi_N(n)) \not\Rightarrow \eta(n)$ **do**
    - set $\eta(\chi_N(n)) \longleftarrow \eta(\chi_N(n)) \wedge \eta(n)$
  - **foreach** $e\ .\ \nu(\chi_E(e)) \not\Rightarrow \nu(e)$ **do**
    - set $\nu(\chi_E(e)) \longleftarrow \nu(\chi_E(e)) \wedge \nu(e)$
- **foreach** $(v_c, v) \in \rightsquigarrow$ **do**
  - **if** $\alpha(v_c) \subseteq_{\mu(v_c)} \alpha(v)$ *not forgetful* **then** remove (v_c, v) from $\rightsquigarrow$; put v_c into Q
- **if** $\exists$ *parent* $v'\ .\ (v', v) \in F$ **then**
  - let $\delta' : \gamma(v') \to \gamma(v)$
  - set $\langle \tau_\triangleleft, \xi \rangle \longleftarrow \mathit{Pullback}(\delta', \xi)$
  - $\mathit{PropagateUp}(v', \tau_\triangleleft, \xi)$

Procedure PropagateUp

other vertex v': this can be done if the concrete trace of v is included in the abstract trace of v' (thus, all causal transitions at v' subtree can be repeated), and, moreover, the inclusion is forgetful, i.e. it "forgets" some action on the left or on the right from the trace.

If the covering attempt was unsuccessful, we linearize the concrete trace of v. If the linear trace L is concretizable - we have found a concrete error trace, and the algorithm terminates with **unsafe**; otherwise there is a conflict in L, and we proceed to the refinement phase, where some causal transition is applied to v.

At the end of each iteration we put into queue Q all children of v, added during the refinement phase, and propagate the premise of the applied causal transition up the unwinding tree. We finish the main loop of the algorithm as soon as queue Q becomes empty: in that case the only uncovered leaf vertices left are contradictory, and the tableau is complete; the algorithm returns **safe**.

The refinement function *Refine* is the main point of application of a wide spectrum of optimizations and specializations possible for causalty-based verification. Here we show one possible instantiation for *Refine*, which operates on a non-linearizable concurrent trace. First, it extracts a conflict from the trace (for example, by computing an unsatisfiable core): a minimal subtrace, which is still non-linearizable. If the subtrace contains some unordered actions or edges, an *order split* is applied, which forces a particular order. Otherwise we consider different cases with respect to the number of actions in the non-linearizable subtrace. If there is only one action, the trace is surely contradictory. If there are two actions, we apply the *necessary action*, which tries to repair the conflict by introducing new action in the middle. For that purpose we compute the Craig interpolant between contradictory actions. Finally, if there are more than two actions in the subtrace we shorten the subtrace by splitting the last but one action with the Craig interpolant between the last action and the rest of them. Each of the causal transitions applied returns its premise and the mapping of it into the concrete trace of the vertex: they are used later for propagation.

The propagation of premises is done in procedure *PropagateUp*, and explained graphically in the left part of Figure 4. Given as input the premise $\tau_\triangleleft$ and the mapping ξ of it to the concrete label $\gamma(v)$, the procedure adds missing components to the abstract label $\alpha(v)$. This is done in two stages: first, the objects (actions or edges), which are present in $\tau_\triangleleft$, but missing in $\alpha(v)$ are inserted into $\alpha(v)$, producing such $\alpha(v)'$ that there is a mapping $\chi : \tau_\triangleleft \to \alpha(v)'$; second, their labels in $\alpha(v)'$ are adjusted in such a way, that $\gamma(v) \subseteq_\chi \alpha(v)''$ holds. Because the abstract label $\alpha(v)$ becomes more concrete, previous covering by that vertex may stop to hold; they are checked and uncovered as needed. Finally, the premise is propagated up in the tableau to vertex v', by constructing the premise $\tau'_\triangleleft$ and the mapping $\xi' : \tau'_\triangleleft \to \gamma(v')$ as a pullback object and arrow of two arrows ξ and δ', where δ' is a direct transformation of $\gamma(v')$ to $\gamma(v)$. Then *PropagateUp* is called recursively with this new premise and mapping; the propagation process terminates either when the root vertex is reached, or when the abstract label $\alpha(v)$ already contains all objects used in the premise, i.e. when the inclusion $\alpha(v) \subseteq \tau_\triangleleft$ holds.

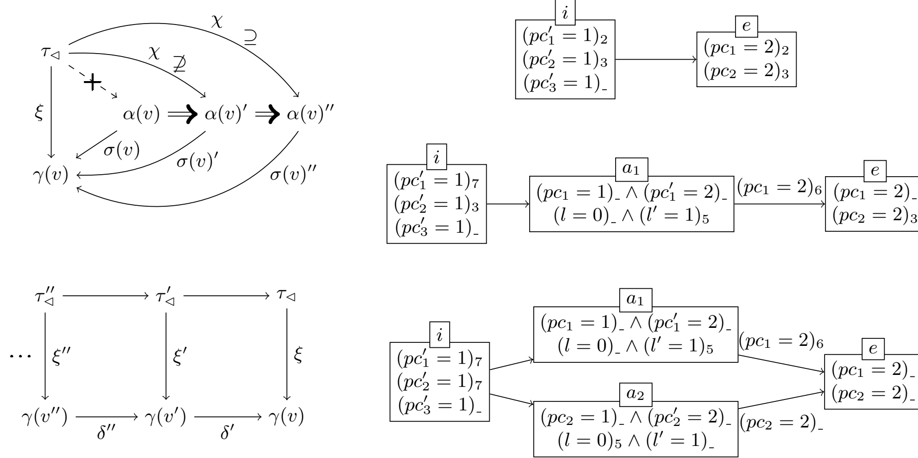


Fig. 4. Upward propagation of premises in trace tableau. *Top left*: calculation of abstract label $\alpha(v)$ in procedure *PropagateUp*. *Bottom left*: pullback construction, propagation of premise $\tau_\triangleleft$ to parent vertices. *Right*: abstract labels for the first three vertices of the tableau from motivating example.

The right part of Figure 4 depicts the abstract labels of the first three vertices of the tableau, obtained after the execution of seven steps from the motivating example. Each conjunct of action or edge labels is marked with the number, showing at which step of the execution this conjunct was included into the abstract label; the undescore sign means that this conjunct is not present in the abstract label after all seven steps. For example, for the top right trace in Figure 4, the conjuncts $(pc'_1 = 1)$ and $(pc_1 = 2)$ were included in the abstract label in the second step, the conjuncts $(pc'_2 = 1)$ and $(pc_2 = 2)$ in the third step, and the conjunct $(pc'_3 = 1)$ was never used for all seven execution steps.

6 Polynomial-Time Verification for Programs with Locks

To demonstrate the advantages of causality-based verification, let us return again to the class of multi-threaded programs with locks, which we considered as a motivating example in Section 2. Standard model checking approaches require exponential, with respect to the number of threads, time and space to prove safety of such programs. In [7], a counterexample-guided refinement algorithm based on cartesian abstraction with exception sets is developed, which is capable to solve in polynomial time the safety problem for the restricted class of programs with locks. The restricted class allows only one lock variable, prohibits nesting/intersection of critical sections, and disallows control flow transfers. Alexander Malkis posed the following:

> **Open Problem ([7], p.65).** Is the most general locks class polynomially verifiable for a fixed number of locks?

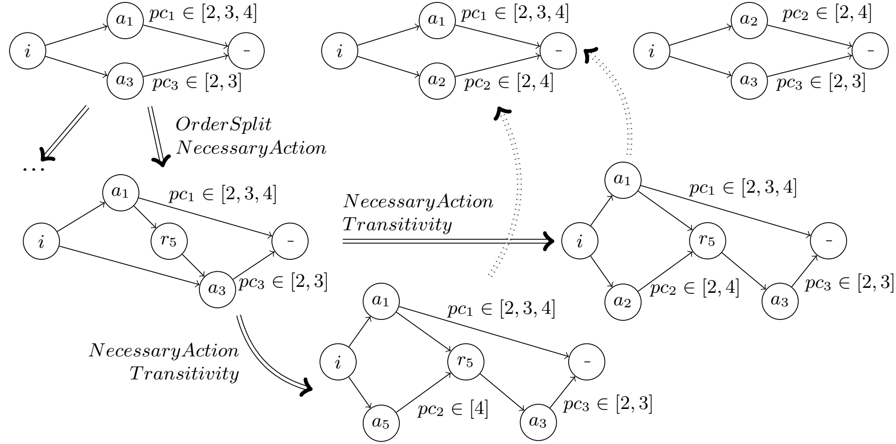


Fig. 5. Part of the causal trace tableau for the example multi-threaded program with locks.

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

Our algorithm starts its computation as shown in Section 2. After several initial steps, the causal tableau starts looping in the repetitions of the same concurrent scenario, shown in Figure 5: two threads enter (actions a_1, a_3) and stay (restrictions $pc_1 \in [2,3,4]$ and $pc_3 \in [2,3]$) in their critical sections for the same lock variable. Because all acquire actions satisfy the constrain $l = 0 \wedge l' = 1$, any ordering of them produces the conflict $l' = 1 \lightning l = 0$. The algorithm applies the *necessary action* causal transition, and inserts a release action, which can be instantiated to transitions r_1, r_2, r_3, and r_5; in the example we consider only transition r_5. Now the trace contains the conflict $pc_2' = 1 \lightning pc_2 = 4$ between actions i and r_5. There are two possible paths between locations 2 and 4 of the second trace; by inserting necessary actions on both alternative paths, we finally introduce actions a_2 and a_5 respectively. Both are acquire actions and we again repeat the concurrent scenario with two threads trying to enter critical sections; thus, the new tableau vertices are covered. Moreover, there is an action (a_3 in this case), which the covering forgets; thus, the covering is forgetful.

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

Theorem 4. *Causality-based verification algorithm proves the safety of the most general class of multi-threaded programs with binary locks in deterministic polynomial time and space with respect to the number of threads and locks.*

7 Conclusion

We have presented a new verification procedure for concurrent systems, which analyzes causal chains in the system behavior. In our procedure, we capture the causal dependencies as Mazurkiewicz-style concurrent traces, and explore the unwinding tree of the causally related traces. Our procedure terminates as soon as all the paths in the unwinding tree are either contradictory, or are covered by other tree vertices, where the same concurrent situation was already examined.

The key ingredient that distinguishes our approach from techniques based on state space exploration or Petri net unfoldings, is that we do not restrict ourselves to only forward or backward analysis of all the transitions available at the current analysis stage. Instead, we try to build a minimal concurrent error trace, which contains only the necessary transitions on the way from initial to error states. We have demonstrated that in some cases, such as multi-threaded programs with locks, our approach reduces the verification complexity from exponential to polynomial.

The full version of the present paper with all proofs is available in [6].

Acknowledgements. The authors thank the anonymous reviewers for their valuable comments and suggestions.

References

1. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 2001.
2. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In Rozenberg [11], pages 163–246.
3. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In Rozenberg [11], pages 247–312.
4. J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking.* EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 2008.
5. P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer-Verlag Inc., New York, NY, USA, 1996.
6. A. Kupriyanov and B. Finkbeiner. Causality-based verification of multi-threaded programs. Reports of SFB/TR 14 AVACS 92, SFB/TR 14 AVACS, 2013. ISSN: 1860-9821, `http://www.avacs.org`.
7. A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2010.
8. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, New York, 1995.
9. A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB 78, Aarhus University, 1977.
10. W. Reisig. *Petri Nets – An Introduction.* Springer, 1985.
11. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.