Causal Termination of Multi-threaded Programs*

Andrey Kupriyanov and Bernd Finkbeiner

Universität des Saarlandes, Saarbrücken, Germany

Abstract. We present a new model checking procedure for the termination analysis of multi-threaded programs. Current termination provers scale badly in the number of threads; our new approach easily handles 100 threads on multi-threaded benchmarks like Producer-Consumer. In our procedure, we characterize the existence of non-terminating executions as Mazurkiewicz-style concurrent traces and apply causality-based transformation rules to refine them until a contradiction can be shown. The termination proof is organized into a tableau, where the case splits represent a novel type of modular reasoning according to different causal explanations of a hypothetical error. We report on experimental results obtained with a tool implementation of the new procedure, called *Arctor*, on previously intractable multi-threaded benchmarks.

1 Introduction

One of the most exciting recent advances in computer-aided verification is the extension of CEGAR-based model checking to liveness properties. Counterexample-guided abstraction refinement (CEGAR) [5] has been very successful in the verification of safety properties, where model checkers like *Magic* [4], *ARMC* [23], and *SLAB* [10] can handle even complex multi-threaded programs. For quite some time, the common belief was that CEGAR is limited to safety — until a new generation of CEGAR-based model checkers, notably the termination checkers *Terminator* [6] and *T2* [2, 8], proved capable of verifying the termination of difficult recursive functions, such as McCarthy's 91 function [16], as well as of reasonably complicated industrial software, such as device drivers. Unlike the model checkers for safety, however, the termination provers have been targeted to sequential programs only, and experiments show that they indeed scale badly for multi-threaded programs.

In this paper, we present *Arctor* (*A*bstraction *R*efinement of *C*oncurrent *T*emporal *Or*derings), the first termination checker that scales to a large number of concurrent threads. On typical multi-threaded programs such as the Producer-Consumer benchmark shown in Fig. 1, where the CEGAR-based tools and, likewise, termination provers based on classic techniques for term rewrite systems,

* 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`).

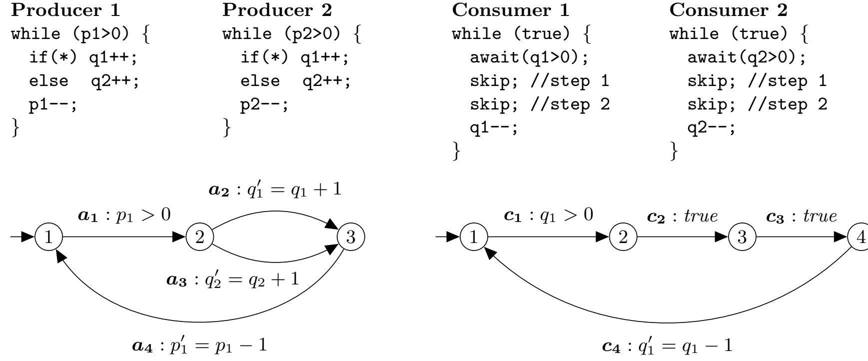


Fig. 1. The *Producer-Consumer* benchmark, shown here for 2 producers and 2 consumers (*Top*: pseudocode; *Bottom*: control flow graphs with labeled transitions for Producer 1 and Consumer 1). The producer threads draw tasks from individual pools and distribute them to nondeterministically chosen queues,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

such as *AProVE* [3, 13], can handle no more than two threads, Arctor proves termination for 100 threads in less than three minutes. Table 1 shows the experimental data for the Producer-Consumer benchmark, the full experimental evaluation is presented in Section 7.

The CEGAR-based termination provers *Terminator* and *T2* build on the *Ramsey*-based approach, introduced by Podelski and Rybalchenko [21], which searches for a termination argument in the form of a *disjunction* of wellfounded relations. If the transitive closure of the transition relation is contained in the union of these relations, we call the disjunction a *transition invariant*; Ramsey's theorem then implies that the transition relation is wellfounded as well. The approach is attractive, because it is quite easy to find individual relations: one can look at the available program statements and take any decreasing transitions as hints for new relations. In the Producer-Consumer example, the termination can be proved with the disjunction of the relations $p_1' < p_1$, $p_2' < p_2$, $q_1' < q_1$, and $q_2' < q_2$. The bottleneck of this approach is the containment check: with an increasing number of relations it becomes very expensive to check the inclusion of the transitive closure of the program transition relation in the transition invariant.

Similar to the Ramsey-based approach, Arctor works with multiple wellfounded relations that are individually quite simple and therefore easy to discover. The key difference is that we avoid disjunctive combinations, which would require us to analyze the transitive closure of the transition relation, and instead combine the relations only either *conjunctively* or based on a *case-split* analysis. Intuitively, our proof in the Producer-Consumer example makes a case distinction based on which thread might run forever. The case that Producer 1 runs

	Terminator		T2		AProVE		Arctor		
Threads	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Vertices
1	3.37	26	2.42	38	3.17	237	0.002	2.3	6
2	1397	1394	3.25	44	6.79	523	0.002	2.6	11
3	×	MO	U(29.2)	253	U(26.6)	1439	0.002	2.6	21
10	×	MO	Z3-TO	×	×	MO	0.027	3.0	135
20	×	MO	Z3-TO	×	×	MO	0.30	4.2	470
60	×	MO	Z3-TO	×	×	MO	20.8	35	3810
100	×	MO	Z3-TO	×	×	MO	172	231	10350

Table 1. Running times of the termination provers *Terminator*, *T2*, *AProVE*, and *Arctor* on the Producer-Consumer benchmark. MO stands for memout; the time spent until memout was in all cases more than 1 hour. U indicates that the termination prover returned "unknown"; Z3-TO indicates a timeout in the Z3 SMT solver.

forever is ruled out by the ranking function p_1. Analogously, Producer 2 cannot run forever because of the ranking function p_2. To rule out that one of the consumers, say Consumer 1, runs forever, we introduce the ranking function q_1, which allows an infinite execution of the `while` loop in Consumer 1 *only if* the `while` loop of Producer 1 or the `while` loop of Producer 2 also run forever, which we have already ruled out with the ranking functions p_1 and p_2. We discuss this example in more detail in Section 3; the informal reasoning should already make clear, however, that the case split has significantly simplified the proof: not only is the termination argument for the individual cases simpler than a direct argument for the full program, the cases also support each other in the sense that the termination argument from one case can be used to discharge the other cases.

Our termination checking algorithm is an extension of the *causality*-based proof technique for safety properties from our previous work [15]. To prove a safety property, we build a *tableau* of Mazurkiewicz-style *concurrent traces*, which capture causal dependencies in the system. The root of the tableau is labeled by a default initial trace, which expresses, by way of contradiction, the assumption that there exists a computation from the initial to the error configuration of the system. We then unwind the tableau by following proof rules that capture, step by step, more dependencies; for example, the *necessary action* rule uses Craig interpolation to find necessary intermediate transitions. We terminate as soon as all branches are found to be contradictory.

In Arctor, we show termination with a similar proof by contradiction that is also guided by the search for an erroneous computation. The difference to the safety case is that, instead of assuming the existence of a computation that leads to an error configuration, we start by assuming the existence of a non-terminating execution, and then pursue the causal consequences that follow from this assumption. In this way, we build a tableau of potentially non-terminating traces. The discovery of a ranking function for the currently considered trace may either close the branch, if the rank decreases along all transitions, or result in one or more new traces, if the rank remains equal or increases along some transitions: in this case, we conclude that the existence of an execution for the current trace implies the existence of an execution for some other trace, in which at least one of these transitions occurs infinitely often.

Related work. In addition to the approaches discussed above, there is a substantial body of techniques for automated termination proofs, in particular for term rewrite systems. Many of these techniques, including simplification orders, dependency pairs, and the size-change principle, are implemented in AProVE [3, 13]. Arctor is the first termination checker that is capable of handling multi-threaded programs with a substantial number of threads. Arctor is based on three key innovations: a novel notion of modular reasoning, a novel composition of ranking functions, and a novel tableau construction based on causality. In the following we point to related work in each of these areas. *Modular reasoning.* The case split in Arctor is a new type of modularity, where the verification task is split according to different causal explanations of a hypothetical error. Other termination provers apply different types of modular reasoning, such as the traditional split according to threads [7], or a split according to ranking functions, by eliminating, after each discovery of a new ranking function, those computations from the program that can now be classified as terminating [12, 2]. *Composition of ranking functions.* Similar to the lexicographic combination of ranking functions, constructed for example by T2 [2, 8], Arctor combines ranking functions within a branch of the tableau conjunctively. The key difference is that Arctor only imposes a partial order, not a linear order, on the individual ranking functions: the same ranking function may be combined independently with multiple other ranking functions from further splits or previously discharged cases. *Causality-based tableaux.* Concurrent traces and the causality-based tableaux are related to other partial-order methods, such as partial order reduction [14], Mazurkiewicz traces [19], and Petri net theory [24]. As explained above, the tableau construction in Arctor is based on our previous work on the causality-based verification of safety properties [15].

2 Concurrent Traces

We begin by introducing *concurrent traces*, which are the basic objects that our verification algorithm constructs and transforms. Concurrent traces capture the dependencies in a transition system.

2.1 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 $\mathcal{S} = \langle \mathcal{V}, T, \mathit{init} \rangle$ 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; $\mathit{init} \in \Phi(\mathcal{V})$ is a state predicate, characterizing the initial system states. A *fair* transition system is enriched with two sets of *just* and *compassionate* transitions $J, C \subseteq T$.

The requirement is that a just (compassionate) transition that is continuously (infinitely often) enabled, should be infinitely often taken.

A *state* of $\mathcal{S}$ is a valuation of system variables $\mathcal{V}$. We call an alternating sequence of states and transitions $s_0, t_1, s_1, t_2, \ldots$ a *run* of $\mathcal{S}$, if $init(s_0)$ holds, and for all $i \geq 1$, $t_i(s_{i-1}, s_i)$ holds. We say that $\mathcal{S}$ is *terminating* if there does not exist an infinite run; otherwise $\mathcal{S}$ is *non-terminating*. We denote the set of runs by $\mathcal{L}(\mathcal{S})$, and the set of non-terminating runs by $\mathcal{L}_n(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S})$.

Transition systems are well suited for the representation of multi-threaded programs with interleaving semantics: the set of transitions of the system consists of all the transitions of the individual threads.

2.2 Finite Concurrent Traces

Finite concurrent traces were introduced in our previous work on causality-based proofs of safety properties [15].

A *finite 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. The source and target functions $s, t : E \to N$ map each edge to its first and second component, respectively. We denote the set of finite concurrent traces by $\mathbb{A}$.

A concurrent trace describes a set of system runs. For a particular concurrent trace its actions specify which transitions should necessarily occur in a run, while its edges represent the (partial) ordering between such transitions and constrain the transitions that occur in-between.

Trace language. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init \rangle$, the *language* of a concurrent trace $A = \langle N, E, \nu, \eta \rangle$ is defined as a set $\mathcal{L}(A)$ of finite system runs such that for each run $s_0, t_1, s_1, t_2, \ldots, t_n, s_n \in \mathcal{L}(A)$ 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 any of its actions is labeled with an unsatisfiable predicate, i.e. if there exists $n \in N$ such that $\nu(n)$ implies $\bot$. Obviously, the language of such a trace is empty.

Given two concurrent traces $A = \langle N, E, \nu, \eta \rangle$ and $A' = \langle N', E', \nu', \eta' \rangle$, a *trace morphism* $f : A \to A'$ is a pair $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ of injective mappings for nodes and edges of one trace to those of another, preserving sources and targets: $f_N \circ t = t' \circ f_E$, and $f_N \circ s = s' \circ f_E$.

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 there exists a trace morphism $\lambda = \langle \lambda_N : N' \to N, \lambda_E : E' \to E \rangle$ such that for all $n' \in N' \,.\, \nu(\lambda_N(n')) \implies \nu'(n')$, and for all $e' \in E' \,.\, \eta(\lambda_E(e')) \implies \eta'(e')$.

We write $A \subseteq_\lambda A'$ if trace inclusion holds for a particular trace morphism λ.

Proposition 1 ([15]). *For $A, A' \in \mathbb{A}$, if $A \subseteq A'$ then $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.*

2.3 Infinite Concurrent Traces

In order to reason about potentially non-terminating computations, we need infinite traces. We define an *infinite concurrent trace* as a tuple $I = \langle A_s, A_c, \phi_s, \phi_c \rangle$ where A_s, A_c are two finite concurrent traces, which we call the *stem* and the *cycle*, and ϕ_s, ϕ_c are two transition predicates. They define the set of infinite system runs in the following way: the stem should occur once in the beginning of the run, while the cycle should occur infinitely often after the stem. Transition predicates ϕ_s and ϕ_c restrict the transitions that are allowed to appear in the stem and cycle part of the run, respectively. We denote the set of infinite concurrent traces by $\mathbb{I}$, and in the following call them simply (concurrent) traces.

Trace language. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init \rangle$, the *language* of an infinite concurrent trace $I = \langle A_s, A_c, \phi_s, \phi_c \rangle$ is defined as a set $\mathcal{L}(I)$ of infinite system runs such that for each run $s_0, t_1, s_1, t_2, \ldots \in \mathcal{L}(I)$ there exists an infinite sequence of indices $i_1, i_2, \ldots$ such that:

1. $s_0, t_1, \ldots t_{i_1}, s_{i_1} \in \mathcal{L}(A_s)$, and for all $0 < j \leq i_1$ the formula $\phi_s\big(s_{j-1}, s_j\big)$ holds.
2. for all $k \geq 2$ it holds that $s_{i_{k-1}}, t_{i_{k-1}+1}, \ldots t_{i_k}, s_{i_k} \in \mathcal{L}(A_c)$, and for all $i_{k-1} < j \leq i_k$ the formula $\phi_c\big(s_{j-1}, s_j\big)$ holds.

Trace inclusion. We lift the trace inclusion relation to infinite concurrent traces. For any two infinite concurrent traces $I = \langle A_s, A_c, \phi_s, \phi_c \rangle$ and $I' = \langle A'_s, A'_c, \phi'_s, \phi'_c \rangle$ we define the *trace inclusion* relation $\subseteq$ as follows: $I \subseteq I'$ iff there exists a pair of trace morphisms $\lambda = \langle \lambda_s, \lambda_c \rangle$, where $\lambda_s : A'_s \to A_s$ and $\lambda_c : A'_c \to A_c$, written also $\lambda : I' \to I$, such that $A_s \subseteq_{\lambda_s} A'_s$, $A_c \subseteq_{\lambda_c} A'_c$, $\phi_s \implies \phi'_s$, and $\phi_c \implies \phi'_c$. For a particular pair of trace morphisms λ we write also $I \subseteq_\lambda I'$.

Proposition 2. *For $I, I' \in \mathbb{I}$, if $I \subseteq I'$ then $\mathcal{L}(I) \subseteq \mathcal{L}(I')$.*

Graphical Notation. We show action identities in circles, and labeling formulas in squares. We omit any of these parts when it is not important or would create clutter in the current context. The cycle part of the trace is depicted in round brackets, superscripted with ω. The predicate ϕ_c is shown under the edge, connecting opening and closing brackets.

3 Motivating Example

In the introduction, we gave an informal sketch of the termination proof for the Producer-Consumer benchmark from Fig. 1. Using the concept of concurrent traces from the previous section, we can now explain the termination argument more formally.

Our analysis starts with the assumption (by way of contradiction) that there exists some infinite run. The assumption is expressed as the concurrent trace at Position 1 in Fig. 2: infinitely often some transition should occur. The transition is so far unknown, and therefore characterized by the predicate $\top$. Our argument proceeds by instantiating this unknown action with the transitions of the

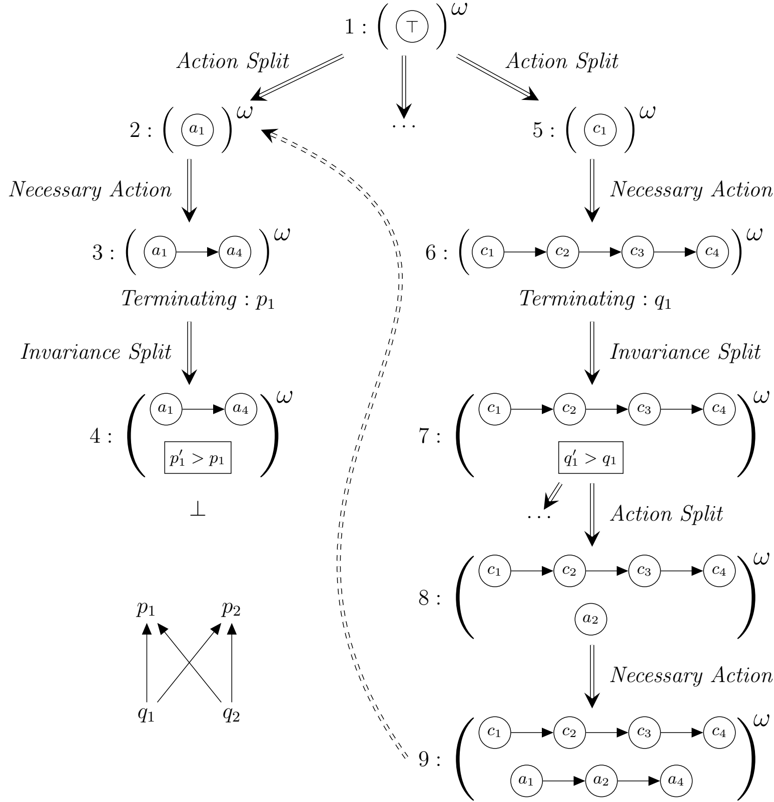


Fig. 2. Termination proof for the Producer-Consumer example. *Bottom left*: partially ordered ranking function discovered in the analysis.

transition system, resulting in one new trace per transition. The *Action Split* proof rule represents a case distinction, and we will need to discharge all cases.

For example, transition a_1 of Producer 1, gives us the trace shown in Position 2. A consequence of the decision that a_1 occurs infinitely often is that a_4 must also occur infinitely often: after the execution of a_1, the program counter of producer 1 equals 2, and the precondition for the execution of a_1 is that it is equal to 1. The only transition, that can achieve that goal, is a_4 (here we oversimplify to make the presentation clearer; in the algorithm we derive the necessity of action a_4 by an interpolation-based local safety analysis). The requirement that both a_1 and a_4 occur infinitely often is expressed as the trace in Position 3, obtained from the trace in Position 1 by the *Necessary Action* proof rule. The edge between a_1 and a_4 specifies an ordering between the two transitions; between them, there may be an arbitrary number of other transitions. The trace in Position 3 is terminating: p_1 is decreased infinitely often and is bounded from below; it is therefore a ranking function. The only remaining situation in which an infinite run might exist is if some transition increases p_1, i.e., that satisfies the predicate

$p_1' > p_1$, is executed infinitely often. This situation is expressed by the trace in Position 4, obtained by the application of the *Invariance Split* proof rule. Since there is no transition in the program transition relation that satisfies $p_1' > p_1$, we arrive at a contradiction.

Let us explore another instantiation of the unknown action in the trace at Position 1, this time with transition c_1 of Consumer 1: we obtain the trace of Position 5. Again, exploring causal consequences, local safety analysis gives us that actions c_2, c_3, and c_4 should also occur infinitely often in the trace: we insert them, and get the trace at Position 6. Termination analysis for that trace gives us the ranking function q_1: it is bounded from below by action c_1 and decreased by action c_4. Again, we conclude that the action increasing q_1 should occur infinitely often, and introduce it in the trace of Position 7. Next, we try all possible instantiations of the action characterized by the predicate $q_1' > q_1$: there are two transitions that satisfy the predicate, namely a_2 and b_2. We explore the instantiation with a_2 in the trace at Position 8; for b_2, the reasoning proceeds similarly. The local safety analysis allows us to conclude that, besides a_2, transitions a_1 and a_4 should occur infinitely often (Position 9). At this point, we realize that the trace at Position 9 contains as a subgraph the trace at Position 2, namely the transition a_1. We can conclude, without repeating the analysis done at Positions 2–4, that there is no infinite run corresponding to the trace at Position 9.

We call the graph of traces corresponding to this analysis the *causal trace tableau.* The tableau for the Producer-Consumer benchmark is (partially) shown in Fig. 2. The analysis can also be understood as the construction of a partially-ordered composition of ranking functions; the final ranking for the Producer-Consumer example is shown at the bottom left of Fig. 2.

We study causal trace tableaux in more detail in the following Section 4. The proof rules driving the analysis are presented in Section 5.

4 Causal Trace Tableaux

We prove termination by constructing a graph labeled by concurrent traces. We call such graphs *causal trace tableaux.*

4.1 Initial Abstraction

At the root of the tableau, we start with a single infinite concurrent trace, containing two actions: the initial action i in the stem part, marked with init', and the infinitely repeating action w in the cycle part, marked with $\top$. The marking ensures that all possible non-terminating system traces are preserved.

Initial Abstraction. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, \mathit{init} \rangle$ we define $\mathit{InitialAbstraction}(\mathcal{S})$ as an infinite concurrent trace $I = \langle A_s, A_c, \phi_s, \phi_c \rangle$, where

- $A_s = \langle N_s, E_s, \nu_s, \eta_s \rangle$, and $N = \{i\}$, $E = \emptyset$, $\nu = \{(i, \mathit{init}')\}$, $\eta = \emptyset$.
- $A_c = \langle N_c, E_c, \nu_c, \eta_c \rangle$, and $N = \{w\}$, $E = \emptyset$, $\nu = \{(w, \top)\}$, $\eta = \emptyset$.
- $\phi_s = \phi_c = \top$.

Proposition 3. *$\mathcal{L}_n(\mathcal{S}) \subseteq \mathcal{L}(\mathit{InitialAbstraction}(\mathcal{S}))$.*

4.2 Causal Transitions

The children of a node in the tableau are labeled with traces that refine the trace of the parent node. We call the rules that construct the children traces from the parent trace *causal transitions*. Technically, causal transitions are special graph morphisms, as described below.

We follow [9, 11] 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.

Trace Productions. A *finite trace production* $p : (L \xrightarrow{r} R)$ is a trace morphism $r : L \to R$, where $L, R \in \mathbb{A}$ are finite concurrent traces. The traces L and R are called the *left-hand side* and the *right-hand side* of p, respectively. A given production $p : (L \xrightarrow{r} R)$ can be applied to a trace A if there is an occurrence of L in A, i.e. a trace morphism $\lambda : L \to A$, called a *match*. The resulting trace A' can be obtained from A by adding all elements of R with no pre-image in L.

An *infinite trace production* $p : (L \xrightarrow{r} R)$ where $L, R \in \mathbb{I}$ are infinite concurrent traces and $r = \langle r_s, r_c \rangle$ is a pair of trace morphisms, describes a transformation of trace L into trace R as a composition of two finite trace productions. In the following we denote the set of infinite trace productions by Π, and call them simply trace productions. We denote the result of the application of a trace production p to a trace I under a pair of morphisms $\lambda = \langle \lambda_s, \lambda_c \rangle$ by $p^\lambda(I)$.

Causal Transitions. For the purpose of system analysis we use special trace productions; we call them *causal transitions*. 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 a causal transition τ is *sound* if the condition below holds:

$$\forall I \in \mathbb{I} \,.\, I \subseteq_\lambda \tau_\triangleleft \implies \mathcal{L}(I) \subseteq \bigcup_{\tau_i \in \tau} \mathcal{L}\big(\tau_i^\lambda(I)\big)$$

4.3 Causal Trace Tableaux

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

- (V, F) is a directed forest with vertices V and edges F. Vertices are partitioned into internal vertices and leaves: $V = V_N \uplus V_L$, $V_N = \{v \in V \mid \exists (v, v') \in F\}$, $V_L = \{v \in V \mid \nexists (v, v') \in F\}$.
- $\gamma : V \to \mathbb{I}$ 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: $\delta(v) = \{\delta((v, v')) \mid (v, v') \in F\}$.
- $\rightsquigarrow : V_L \nrightarrow V_N$ is a partial *covering* function; for $(v, v') \in \rightsquigarrow$ we call v a *covered* vertex, and v' a *covering* vertex.

– λ is a labeling of internal or covered vertices with trace morphisms: $\forall v \in V_N \;.\; \lambda(v) : \delta(v)_{\lhd} \rightarrow \gamma(v)$; for all $(v, v') \in \rightsquigarrow\;.\; \lambda(v) : \gamma(v') \rightarrow \gamma(v)$.

We call a trace tableau $\Upsilon = \langle V, F, \gamma, \delta, \rightsquigarrow, \lambda \rangle$ *complete* if all its leaf vertices are either contradictory or covered. A trace tableau is said to be *correct* if:

1. $\exists v \in V \;.\; \mathit{InitialAbstraction}(\mathcal{S}) \subseteq \gamma(v)$.
2. for all $v \in V_N$ we have that a) $\delta(v)$ is sound, b) $\gamma(v) \subseteq_{\lambda(v)} \delta(v)_{\lhd}$, and c) for all $(v, v') \in F$ it holds that $\delta\big((v, v')\big)^{\lambda(v)}\big(\gamma(v)\big) \subseteq \gamma(v')$.
3. for all $(v, v') \in \rightsquigarrow$ we have $\gamma(v) \subseteq_{\lambda(v)} \gamma(v')$ and $(v', v) \notin (F \cup \rightsquigarrow)^*$.

A trace tableau is a forest, which can be seen as an unwinding of the system causality relation from some set of initial vertices. The label $\gamma(v)$ of the vertex v represents all possible infinite runs for that vertex. The first correctness condition requires that a tableau contains all non-terminating system runs. The second one guarantees the applicability of the causal transition $\delta(v)$ of a vertex v to its label $\gamma(v)$ and the full exploration of the causal transition consequences, thus preserving the set of system runs. Indeed, we have:

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

The last correctness condition ensures that we can apply at the covered vertex all the causal transitions in the subtree originating from the covering vertex; additionally, it guarantees that the resulting tableau is acyclic.

5 Proof Rules for Termination

We now present the causal transitions needed for proving termination. We omit the proof rules for safety properties presented in [15], which can be applied to the stem part of a concurrent trace without any changes. The infinity-specific causal transitions are illustrated in Fig. 3; some have so called *application conditions*, showed to the right of the causal transition name.

Basic rules. These are the most important causal transitions, sufficient for the approach completeness in the case of an unconditionally terminating system.

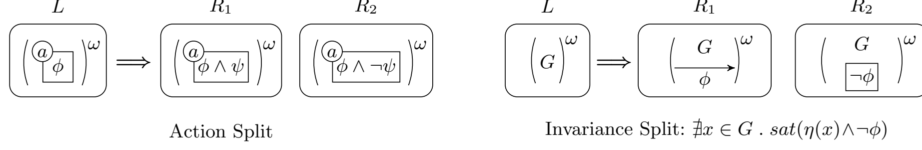
Action Split, given some action a in the cycle part of a trace, and a transition predicate ψ, considers two alternatives: either a satisfies ψ or $\neg\psi$ infinitely often.

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

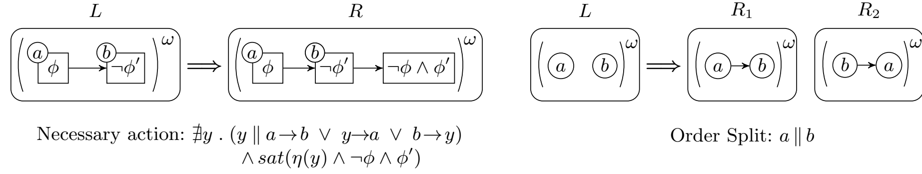
Local safety rules. The rules in this category make the approach efficient for the case we cannot find a perfect ranking function in one step.

Necessary Action is applied when there are two ordered actions a and b in the cycle part of a concurrent trace, and a transition predicate ϕ, such that the label

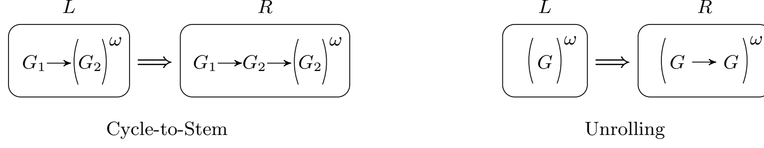
Basic Rules



Local Safety Rules



Unrolling Rules



Fairness Rules

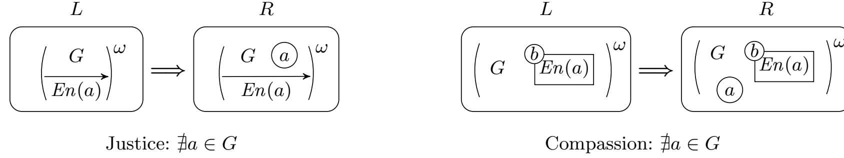


Fig. 3. Proof rules for termination. Action identities are arbitrary, and used to show the context preserved by the application of a causal transition. G stands for the rest of the trace besides the depicted parts.

of a implies ϕ, and the label of b implies $\neg\phi'$, i.e. there is a contradiction between these actions (b "ends" in the region $\neg\phi$, while a "starts" in the region ϕ). Given the repetitive character of the trace, we have that a should follow b again; the causal transition introduces a new "bridging" action x in between. Actions a and b can be the same single action, where the precondition contradicts the postcondition. The predicate ϕ may be obtained by Craig interpolation between the labels of b and a. 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.

Order Split considers alternative interleavings of two previously concurrent events. Either one or another ordering should happen infinitely often.

Unrolling rules use the infinite repetition of the cycle part of a trace.

Cycle-to-Stem causal transition allows us to shift the cycle part G_2 into the stem part G_1, thus going from the reasoning at infinity to the safety reasoning. This rule is need for the completeness of conditional termination.

Unrolling exploits the infinite repetition: we can unroll the complete graph G of the cycle part, and the unrolled version should still repeat infinitely often.

Fairness rules allow for the direct account of two well-known concurrent phenomena: weak and strong fairness (or justice and compassion) [17, 18].

Justice causal transition allows to introduce a just transition a in the cycle part of a trace in case it is continuously enabled and never taken.

Compassion causal transition states that a compassionate transition a which is infinitely often enabled should be also infinitely often taken.

Proposition 4. *The defined above causal transitions are sound.*

The following lemma is applied in the combination with the *invariance split* causal transition: in case a ranking function can be found for the cycle part of a concurrent trace, it allows to discard the left branch of the result.

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

6 The Termination Analysis Algorithm

Our termination analysis algorithm (see Algorithm 1) operates on the causal trace tableau defined above. We start with the single vertex in the tableau, labeled with *InitialAbstraction*$(\mathcal{S})$. At each iteration of the algorithm main loop we select some vertex v from the queue Q of unexplored tableau leaves, and analyze only the cycle part of its label. First, we try to cover v by some other vertex v': this can be done if the trace of v is included in the trace of v' (thus, all causal transitions at v' subtree can be repeated). Moreover, we require that the covering does not create any loops, and the resulting tableau is acyclic.

If the covering attempt was unsuccessful, we unroll and linearize the cyclic part of the v's label. The unrolling is necessary in order to detect possible conflicts between iterations of the cycle. If the linear trace L is unconcretizable - we apply the local safety refinement to the cyclic trace: this includes such causal transitions as *order split* and *necessary action.* The refinement procedure is essentially the same as in [15], so we do not repeat it here. After the refinement step, we put the newly created children of v into the queue and proceed.

On the contrary, if the unrolled cycle is concretizable, we check it for termination; any ranking function synthesis algorithm such as [1, 22] can be used for that purpose. If we have found a ranking function for the cycle - we apply the *invariance split* causal transition and Lemma 1. As a result, we introduce into

Algorithm 1: Causality-based Termination Analysis

Input : Transition system $\mathcal{S} = \langle \mathcal{V}, T, init \rangle$
Output: **terminating/termination unknown**
Data: Termination tableau $\Upsilon = \langle V, F, \gamma, \delta, \rightsquigarrow, \lambda \rangle$, queue $Q \subseteq V_L$,
Safety tableau $\Upsilon_s = \langle V_s, F_s, \gamma_s, \delta_s, \rightsquigarrow_s, \lambda_s \rangle$, queue $Q_s \subseteq V_{L\,s}$

begin
- set $V \longleftarrow \{v_0\}$, $\gamma(v_0) \longleftarrow \mathit{InitialAbstraction}(\mathcal{S})$, $Q \longleftarrow \{v_0\}$
- set all of $\{F, \delta, \rightsquigarrow, \lambda, V_s, F_s, \gamma_s, \delta_s, \rightsquigarrow_s, \lambda_s\} \longleftarrow \emptyset$
- **while** Q *not empty* **do**
  - take some v from Q
  - **if** $\exists\, v' \in V_N$ *and* $\lambda' : \gamma(v') \rightarrow \gamma(v)$. $\gamma(v) \subseteq_{\lambda'} \gamma(v')$ *and v is not reachable from v' by* $F \cup \rightsquigarrow$ **then**
    - add (v, v') to $\rightsquigarrow$
    - set $\lambda(v) \longleftarrow \lambda'$
  - **else**
    - set $L \longleftarrow \mathit{Linearize}(\mathit{Unroll}(\gamma(v)))$
    - **if** $\mathit{Unconcretizable}(L)$ **then**
      - $\mathit{LocalSafetyRefine}(v, L)$
    - **else if** $\mathit{Terminating}(L)$ **then**
      - $\mathit{InvarianceSplit}(v, L)$
    - **else**
      - put $\mathit{CycleToStem}(v)$ into V_s and Q_s
    - put children of v into Q
- **return** $\mathit{SafetyAnalysis}(\Upsilon_s, Q_s)$

the cycle all possible system transitions able to "repair" it: preserve its infinite repetition despite the existence of a ranking function.

Finally, if the cycle is both concretizable and no ranking function can be found for it, the termination part of our algorithm gives up and transfers the analyzed trace to the safety part of the algorithm. For that purpose we concatenate the stem and the cycle into the finite concurrent trace, and put the resulting vertex into the safety tableau for processing. Thus, the safety tableau is the forest that originates from the vertices of the termination tableau for which no termination argument can be found. We apply the method of [15] to analyze it. If all the leaves of the safety tableau are found to be unreachable, we mark the corresponding leaves of the termination tableau as contradictory, and report the program as terminating; otherwise we report a possibly non-terminating execution. The following theorems show that the proposed approach is sound and relatively complete for the program termination analysis.

Theorem 1 (Soundness). *If there exists a correct and complete causal trace tableau for a transition system $\mathcal{S}$, then $\mathcal{S}$ is terminating.*

Theorem 2 (Relative Completeness). *If a transition system $\mathcal{S}$ is terminating, then a correct and complete causal trace tableau for $\mathcal{S}$ can be constructed, provided that all necessary first-order formulas are given.*

	Terminator		T2		AProVE		Arctor		
Benchmark	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Time(s)	Mem.(MB)	Vertices
Chain 2	0.65	20	0.52	20	1.58	131	0.002	2.0	3
Chain 4	1.45	25	0.54	22	2.13	153	0.002	2.2	7
Chain 6	24.4	57	0.58	24	2.58	171	0.002	2.5	11
Chain 8	×	MO	0.63	26	3.48	210	0.002	2.5	15
Chain 20	×	MO	2.36	55	16.5	941	0.007	2.5	39
Chain 40	×	MO	40.5	288	536	1237	0.023	2.8	79
Chain 60	×	MO	Z3-TO	×	×	MO	0.063	3.0	119
Chain 100	×	MO	Z3-TO	×	×	MO	0.320	3.9	199
Phase 1	×	MO	U(4.53)	48	1.60	132	0.002	2.4	7
Phase 2	×	MO	U(4.53)	48	2.16	144	0.002	2.4	7
Phase 3	×	MO	U(30.6)	301	3.83	199	0.002	2.5	16
Phase 8	×	MO	×	MO	47.0	1506	0.003	2.6	61
Phase 10	×	MO	×	MO	×	MO	0.012	2.7	79
Phase 20	×	MO	×	MO	×	MO	0.061	3.3	169
Phase 60	×	MO	×	MO	×	MO	1.18	4.2	529
Phase 100	×	MO	×	MO	×	MO	7.38	6.1	889
Semaphore 1	3.05	26	2.81	46	3.22	230	0.002	2.6	8
Semaphore 2	622	691	U(20.7)	219	U(6.52)	465	0.002	2.6	16
Semaphore 3	×	MO	U(15.8)	239	U(10.42)	1138	0.003	2.6	24
Semaphore 10	×	MO	U(83.5)	470	U(246)	1287	0.023	2.8	80
Semaphore 20	×	MO	×	MO	×	MO	0.073	3.3	160
Semaphore 60	×	MO	×	MO	×	MO	0.58	4.0	480
Semaphore 100	×	MO	×	MO	×	MO	1.59	5.1	800

Table 2. Detailed experimental evaluation for the set of multi-threaded benchmarks. MO stands for memout; the time spent until memout was in all cases more than 1 hour. U indicates that the termination prover returned "unknown"; Z3-TO indicates a timeout in the Z3 SMT solver.

As the termination problem is, in general, undecidable, our approach has its limitations. In particular, it heavily depends on the power of termination proving techniques for simple loops such as [1, 22], and on the methods for reachability analysis. For the latter we apply our previous work [15], which is limited to theories, supporting Craig interpolation.

7 Experimental Evaluation

We have implemented the termination analysis algorithm in a model checker called *Arctor*[1]. The implementation consists of approximately 1500 lines of Haskell code and can currently handle multi-threaded programs with arbitrary control flow, finite data variables, and unbounded counters.

Table 2 shows experimental results obtained with the termination provers Terminator, T2, AProVE, and Arctor on the benchmarks described below, except for the results on the Producer-Consumer benchmark, which were discussed already in the introduction (see Table 1). All experiments were performed on an Intel Core i7 CPU running at 2.7 GHz.

Producer-Consumer. The *Producer-Consumer* benchmark from the introduction (see Fig. 1) is a simplified model of the *Map-Reduce* architecture from distributed processing: producers model the mapping step for separate

[1] available at http://www.react.uni-saarland.de/tools/arctor/

data sources, consumers model the reducing step for different types of input data. The natural requirement for such an architecture is that the distributed processing terminates for any finite amount of input data.

Chain. The *Chain* benchmark consists of a chain of n threads, where each thread decreases its own counter x_i, but the next thread in the chain can counteract, and increase the counter of the previous thread. Only the last thread in the chain terminates unconditionally.

Phase. The *Phase* benchmark is similar to the Chain benchmark, except that now each thread can either increase or decrease its counter x_i. Each such *phase change* is, however, guarded by the next thread in the chain, which limits the number of times the phase change can occur.

Semaphore. The *Semaphore* benchmark represents a model of a concurrent system where access to a critical resource is guarded by semaphores. We verify *individual accessibility* for a particular thread (i.e., the system without the thread should terminate) under the assumption of a *fair scheduler*. Since other tools do not support fairness, we have eliminated the fairness assumption for all tools including Arctor using the transformation from [20], which enriches each `wait` statement with a decreasing and bounded counter.

Arctor verifies all benchmarks efficiently, requiring little time and memory to handle even 100 threads. We have also tried out our approach on more complicated examples (available from the tool homepage), which represent models of publicly available industrial programs. These include parallel executions of GNU `make`, parallel computations in CUDA programs for GPUs, and Google's implementation of the Map-Reduce architecture for its *App Engine* platform. While other tools are not able to handle even two concurrent threads in these programs, Arctor verifies programs with dozens of threads within several minutes.

8 Conclusion

We have presented a new model checking procedure for the termination analysis of multi-threaded programs. The procedure has been implemented in *Arctor*, the first termination prover that scales to a large number of concurrent threads. Our approach is based on three key innovations: a novel notion of modular reasoning, a novel composition of ranking functions, and a novel tableau construction based on causality. With respect to the modular reasoning, the case split in Arctor is a new, and very effective, type of modularity, where the verification task is split according to different causal explanations of a hypothetical error. With respect to the composition of ranking functions, Arctor combines ranking functions within a branch of the tableau conjunctively, similar to the lexicographic combination in T2, but Arctor only imposes a partial order, not a linear order, on the individual ranking functions: the same ranking function may be combined independently with multiple other ranking functions from further splits or previously discharged cases. Finally, Arctor explores causal dependencies in a tableau of Mazurkiewicz-style concurrent traces in order to systematically discover case splits and ranking functions.

References

1. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2005.
2. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 413–429. Springer, 2013.
3. Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, Lecture Notes in Computer Science. Springer, 2014. To appear.
4. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.
5. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.
6. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41(6):415–426, June 2006.
7. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. *ACM SIGPLAN Notices*, 42(6):320, June 2007.
8. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.
9. Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In Rozenberg [25], pages 163–246.
10. Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2010.
11. Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In Rozenberg [25], pages 247–312.
12. Pierre Ganty and Samir Genaim. Proving Termination Starting from the End. *Computer Aided Verification*, (10):397–412, 2013.
13. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.
14. Patrice 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.
15. Andrey Kupriyanov and Bernd Finkbeiner. Causality-based verification of multi-threaded programs. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013.

16. Zohar Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974.
17. Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
18. Zohar Manna and Amir Pnueli. Temporal verification of reactive systems: Response. In Zohar Manna and Doron Peled, editors, *Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 279–361. Springer, 2010.
19. Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB 78, Aarhus University, 1977.
20. Ernst-Rüdiger Olderog and Krzysztof R. Apt. Fairness in parallel programs: The transformational approach. *ACM Trans. Program. Lang. Syst.*, 10(3):420–455, 1988.
21. A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, July 2004.
22. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.
23. Andreas Podelski and Andrey Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin Heidelberg, 2007.
24. W. Reisig. *Petri Nets – An Introduction*. Springer, 1985.
25. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.