

Transition Predicate Abstraction and Fair Termination

ANDREAS PODELSKI

Max-Planck-Institut für Informatik, Saarbrücken

and

ANDREY RYBALCHENKO

Ecole Polytechnique Fédérale de Lausanne

Max-Planck-Institut für Informatik, Saarbrücken

(SPECIAL POPL ISSUE) Predicate abstraction is the basis of many program verification tools. Until now, the only known way to overcome the inherent limitation of predicate abstraction to safety properties was to manually annotate the finite-state abstraction of a program. We extend predicate abstraction to *transition predicate* abstraction. Transition predicate abstraction goes beyond the idea of finite *abstract-state* programs (and checking the absence of loops). Instead, our abstraction algorithm transforms a program into a finite *abstract-transition* program. Then, a second algorithm checks fair termination. The two algorithms together yield an automated method for the verification of liveness properties under full fairness assumptions (impartiality, justice, and compassion). In summary, we exhibit principles that extend the applicability of predicate abstraction-based program verification to the full set of temporal properties.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Theory, Verification.

Additional Key Words and Phrases: Software model checking, transition predicate abstraction, fair termination, liveness.

1. INTRODUCTION

Since 1977, a high amount of research, both theoretical and applied, has been invested in honing the tools for abstract interpretation [Cousot and Cousot 1977] for verifying safety and invariance properties of programs. This effort has been a success. One promising approach is *predicate abstraction* on which a number of academic and industrial tools are based [Graf and Saïdi 1997; Ball et al. 2001; Yahav 2001; Henzinger et al. 2002; Chaki et al. 2003].

What has been left open is how to obtain the same kind of tools for the full set

This research was supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year, Pages 1–??.

of temporal properties. So far, there was no viable approach to the use of abstract interpretation for analogous tools establishing liveness properties (under fairness assumptions). This paper presents the first steps towards such an approach. We believe that our work may open the door to a series of activities for liveness, similar to the one mentioned above for safety and invariance.

One basic idea of abstraction is to transform the program to be checked into a more abstract one, one on which the property still holds. When we are interested in termination under fairness assumptions, we need to solve two problems: the abstract program needs to preserve (1) the termination property, and (2) the fairness assumptions. (Checking liveness can be reduced to fair termination, just as safety reduces to reachability.) In this paper, we show how to solve these two problems. We propose a transformation of a program into a node- and edge-labeled graph such that the termination property can be retrieved from the node labels and the fairness assumptions from the edge labels. (Note that it does not check the absence of loops in the graph. For proving termination our method only inspects the node labeling.) The transformation is based on *transition predicate abstraction*, an extension of predicate abstraction that we propose.

The different steps in our automated method for checking a liveness property under fairness assumptions are:

- the reduction of the liveness property to fair termination (this reduction is standard, see e.g. [Vardi 1991]);
- the transition predicate abstraction-based transformation of the program P into a node- and edge-labeled graph, the *abstract-transition program* $P^\#$;
- a number of termination checks that mark some nodes of $P^\#$ as ‘terminating’;
- an algorithm on the automaton underlying $P^\#$ that marks some nodes as ‘fair’;
- the method returns ‘property verified’ if each ‘fair’ node is marked ‘terminating’.

Our conceptual contribution lies in the use of transition predicates for automated liveness proofs. Our technical contributions are the algorithm to retrieve fairness in the abstract program $P^\#$, and the proof of the correctness of the overall method. We use three relevant kinds of fairness, which are impartiality, justice, and compassion (to model the assumption that a transition is eventually taken unconditionally, if it is continually or infinitely often enabled).

2. RELATED WORK

Our work is most closely related to the work on predicate abstraction; see e.g. [Graf and Saïdi 1997; Ball et al. 2001; Yahav 2001; Henzinger et al. 2002; Chaki et al. 2003]. The key idea of predicate abstraction is to partition the state space of the program into a finite set of equivalence classes using predicates over states. The equivalence classes are treated as the *abstract states* forming the nodes of a finite graph. A safety property can then be checked on the abstract system. Predicate abstraction can also provide a basis for the development of testing methods by guiding the test generation, e.g. [Ball 2005].

Unfortunately, predicate abstraction is inherently limited to safety properties. That is because every sufficiently long computation of the program (with the length greater than the number of abstract states) results in a computation of the abstract

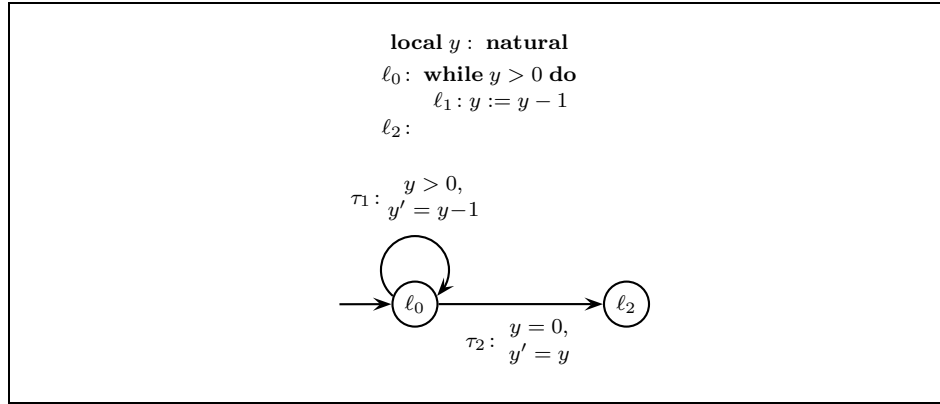


Fig. 1. Terminating program LOOP. This small program shows that state abstractions does not preserve termination property. For example, consider the abstract-state program in Figure 2. It contains a self-loop at the abstract state S_1 .

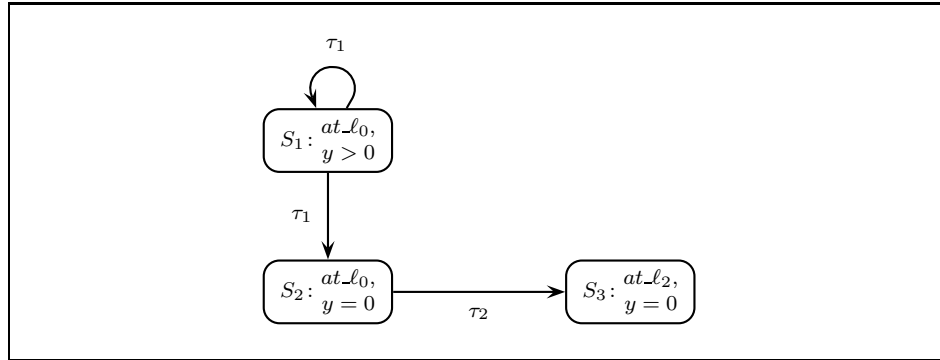


Fig. 2. Non-terminating abstract-state program for LOOP.

system that contains a loop. This means that termination (as well as more general liveness properties) cannot be preserved by predicate abstraction.

We illustrate the limitation of state abstraction on a very simple program LOOP [Kesten and Pnueli 2000], shown in Figure 1 together with the (slightly simplified) control-flow graph. The predicates $y = 0$ and $y > 0$ split the data domain of the variable y . The corresponding abstraction transforms the program LOOP into the finite abstract-state program shown in Figure 2. That program contains a self-loop, i.e. is not terminating. The abstract state S_1 corresponds to the conjunction $at_l_0 \wedge y > 0$ denoting the set of states where the program counter has the value l_0 and y is strictly positive. If we split the abstract state S_1 (by adding more predicates) then at least one of the resulting abstract states will have a self-loop, and so on.

In the *augmented abstraction* framework for proving liveness properties, the finite-state abstraction is annotated by progress monitors or the like [Merz 1997; Kesten and Pnueli 2000; Kesten et al. 2001; Pnueli et al. 2002; Yahav et al. 2003]. The

annotation involves the manual construction of ranking functions or other termination arguments. Until now, this has been the only known way to overcome the inherent limitation of predicate abstraction to safety properties. In contrast, the method that we propose does not require the manual construction of termination arguments.

In [Podelski and Rybalchenko 2004b] we presented a proof rule for termination and liveness based on *transition invariants*. In this paper, we make the first steps towards realizing its potential for automation.

We note a major difference in the notions of fairness used here and in [Podelski and Rybalchenko 2004b]. In [Podelski and Rybalchenko 2004b], we used an automata-theoretic notion of state-based fairness to formalize a uniform setting. Here we use impartiality, justice, and compassion, which are transition-based notions of fairness (see [Francez 1986] for a detailed treatment of fairness requirements). These are the notions of fairness that are relevant with concrete concurrent programs. It is widely accepted that one needs a direct treatment of fairness. As a consequence, the notion of transition invariant in [Podelski and Rybalchenko 2004b] is not applicable as such. For intuition, an abstract program $P^\#$ can be imagined as a new notion of transition invariant, one that encodes impartiality, justice, and compassion assumptions in a graph with labeled edges.

The abstract interpretation framework formalizes the conservative approximation of fixpoint expressions [Cousot and Cousot 1977]. For the verification of liveness properties denoted by fixpoint expressions, this approximation involves the under-approximation of least fixpoints or (equivalently) the over-approximation of greatest fixpoints. Although possible in principle, the automation of the corresponding extrapolation seems difficult, and practical techniques (analogous to the extrapolation by intervals, convex hulls, Cartesian products, etc.) are not in sight (cf. [Bourdoncle 1993; Sipma et al. 1996; Uribe 1999; Delzanno and Podelski 2001]).

One source of inspiration for the idea of abstracting relations is the work on higher-order abstract interpretation in [Cousot and Cousot 1994]. Its instantiation to transition predicate abstraction and its use for liveness with impartiality, justice and compassion is proper to this paper.

Ramsey’s theorem [Ramsey 1930] is a prominent tool for reasoning about infinite sequences. It plays an important role in the theory of Büchi automata [Büchi 1960], and has also found applications in termination analysis, e.g. [Sagiv 1991; Dershowitz et al. 2001; Lee et al. 2001; Codish et al. 2003; Podelski and Rybalchenko 2004b].

The restriction of our proof argument to termination compares to the proof argument used in the termination analysis of logic programs as e.g. in [Sagiv 1991; Dershowitz et al. 2001; Codish et al. 2003]. Our setting develops a more general form of abstraction of the composition of relation, in contrast with an abstract composition operator in [Dershowitz et al. 2001] which is required to be associative (such an operator is called “structured shadow” in the terminology of [Dershowitz et al. 2001]). Structured shadows map relations into query-mapping pairs [Sagiv 1991], which are graph-based relational assertions designed to abstract transition relations of logic programs, and, hence, account for instantiation of parameters, numerical arguments, term sizes, etc. The termination check is performed on the transitive closure of structured shadows of each procedure call. In our setting, the abstraction is performed on-the-fly. The resulting abstract-transition program pro-

vides a connection between abstract transitions and the corresponding sequences of program transitions, which allows us to distinguish between ‘fair’ and ‘unfair’ nodes in the abstract-transition programs.

The termination analysis of [Lee et al. 2001] for functional programs is based on the comparison of infinite paths in the control-flow graph and in ‘size-change graphs’; that comparison can be reduced to the language containment test of Büchi automata. Our work extends the termination principle in [Lee et al. 2001] to a setting with *parameterized* abstraction (parameterized by sets of transition predicates), liveness, and fairness.

Our method can be seen as a weakening of the Ramsey’s theorem based approaches [Sagiv 1991; Dershowitz et al. 2001; Lee et al. 2001; Codish et al. 2003], which allows one to exclude non-fair infinite computations from consideration by checking only the well-foundedness of ‘fair’ nodes.

The stack assertions for proving fair termination account for fairness through the combination of ranking functions in a stack structure [Klarlund 1992]. The method described there requires the manual construction of ranking functions.

Verification diagrams are graphs that are useful to factorize deductive proofs of temporal properties including liveness [Browne et al. 1995]. In this regard, they are similar to abstract-transition programs. However, the nodes in verification diagrams denote sets of states (and not pairs of states); hence, they are close in spirit to abstract-state programs (and fundamentally different from abstract-transition programs). It may be interesting to consider verification diagrams with nodes denoting sets of *pairs* of states, and to come up with corresponding proof rules.

The predicate abstraction method of [Colón and Uribe 1998] constructs an abstract-state system for which it can automatically transfer some fairness requirements of the input program. The method applies to liveness properties that can be proven by considering only the transferred fairness, which crucially depends on the precision of the abstraction. Our method accounts for fairness requirements without abstracting them.

Our method is parameterized by a procedure for checking well-foundedness of abstract-transitions. We can apply many existing algorithms and tools for this task, e.g. from the domain of term rewriting systems [Arts and Giesl 2000] and numerical programs [Colón and Sipma 2001; Podelski and Rybalchenko 2004a; Bradley et al. 2005].

3. ABSTRACT-TRANSITION PROGRAMS

3.1 Informal Description

We propose to abstract *relations* instead of *sets of states*, and to use *transition predicate* abstraction instead of *predicate* abstraction. Transition predicates are binary relations over states (given e.g. by assertions over unprimed and primed program variables).

Transition predicate abstraction goes beyond the idea of abstracting a program by a finite *abstract-state* program (which does not preserve termination as well as more general liveness properties, as illustrated in Section 2.) Instead, we abstract a program by a finite *abstract-transition* program. An abstract transition is a binary relation represented by a conjunction of transition predicates. An abstract-

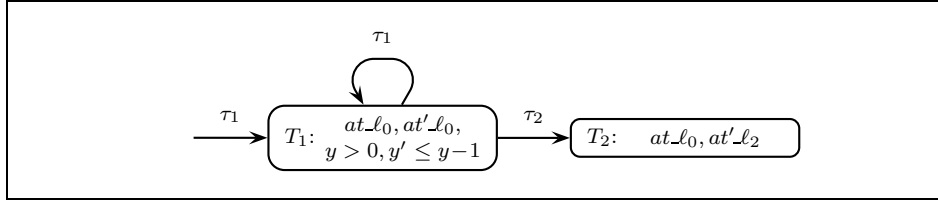


Fig. 3. Abstract-transition program LOOP#.

transition program is given by a finite directed graph whose nodes are labeled by abstract transitions, and whose edges are labeled by program statements, later formalized as transitions $\tau \in \mathcal{T}$.

We use the terminology ‘abstract-transition programs’ in order to stress the analogy with ‘abstract-state programs’ (and we use the term ‘programs’ in a broad sense, as specifications of computational behavior.)

In Figure 3, we see the abstract-transition program LOOP#. One node is labeled by the abstract transition T_1 . It corresponds to the conjunction of *transition predicates*

$$at_l_0 \wedge at'_l_0 \wedge y > 0 \wedge y' \leq y - 1$$

denoting the set of all pairs of states (s, s') , both at the program location l_0 . The value of y is strictly positive in the state s , and changes to a strictly smaller value in s' . The node labeled by T_2 refers to states s and s' at l_0 and at l_2 (with unspecified values for y), respectively.

The abstract-transition program LOOP# abstracts the program LOOP, shown in Figure 1. What does this mean?

We first recall the meaning of abstraction of a program by an abstract-state program. If a state s has a transition to s' under the execution of the program statement τ , then there is an edge labeled by τ between two corresponding abstract states S_1 and S_2 (i.e. $s \in S_1$ and $s' \in S_2$).

The meaning of abstraction of a program by an abstract-transition program is analogous. If a pair of states (s, s') can be ‘extended’ to the pair (s, s'') by the execution of the program statement τ (which is: s' goes to s'' under the execution of the statement τ), then there is an edge labeled by τ between two corresponding abstract transition T_1 and T_2 (which is: $(s, s') \in T_1$ and $(s, s'') \in T_2$).

Note that LOOP# only serves to demonstrate the concept of abstract-transition programs. To illustrate how our method works to verify termination and general liveness properties, we will use concurrent programs with nested loops. In fact, the program LOOP is an example of a *single while loop* program. Our method calls (as a subroutine) a termination check that exists for single while loop programs [Colón and Sipma 2001; Podelski and Rybalchenko 2004a; Tiwari 2004].

We now start the formal definitions.

3.2 Programs and Computations

Following [Manna and Pnueli 1995], we abstract away from the syntax of a concrete (concurrent) programming language and represent a program P by a *fair transition*

system

$$P = \langle \Sigma, \Theta, \mathcal{T}, \mathcal{I}, \mathcal{J}, \mathcal{C} \rangle$$

consisting of:

- Σ : the set of *states*,
- Θ : a set of *initial* states such that $\Theta \subseteq \Sigma$,
- \mathcal{T} : a finite set of *transitions* such that each transition $\tau \in \mathcal{T}$ is associated with a *transition relation* ρ_τ ,

$$\rho_\tau \subseteq \Sigma \times \Sigma$$

- \mathcal{I} : a set of *impartial* transitions such that $\mathcal{I} \subseteq \mathcal{T}$,
- \mathcal{J} : a set of *just* transitions such that $\mathcal{J} \subseteq \mathcal{T}$,
- \mathcal{C} : a set of *compassionate* transitions such that $\mathcal{C} \subseteq \mathcal{T}$.

A *computation* σ is a sequence of states s_1, s_2, \dots , which is either infinite or not more extendible, such that:

- s_1 is an *initial* state, i.e. $s_1 \in \Theta$,
- for each $i \geq 1$ there exists a transition $\tau \in \mathcal{T}$ such that s_i goes to s_{i+1} under ρ_τ , formally

$$(s_i, s_{i+1}) \in \rho_\tau.$$

We will define fairness requirements (impartiality, justice, and compassion) in Sections 7, 8, and 9, respectively.

We write example programs using the Simple Programming Language SPL of [Manna and Pnueli 1995]. The translation from SPL and other (concurrent) programming languages into fair transition systems is standard.

In SPL every program statement has a unique location ℓ which is reached by the control when execution of the statement has just finished. We define a predicate at_ℓ such that at_ℓ holds for a state s if the control resides at the location ℓ in the state s . See [Manna and Pnueli 1995] for a detailed discussion on program locations.

3.3 Transition Predicates

We now define the building blocks for abstract-transition programs.

Definition 1. (TRANSITION PREDICATE p). A *transition predicate* p is a binary relation over states:

$$p \subseteq \Sigma \times \Sigma.$$

We denote transition predicates by atomic assertions over unprimed and primed program variables.

We fix a transition predicate Id for the identity relation:

$$Id = \{(s, s) \mid s \in \Sigma\}.$$

From now on, the formal statements refer to a fixed *finite* set of transition predicates \mathcal{P} . We assume that for each program location ℓ the predicates at_ℓ and at'_ℓ are contained in \mathcal{P} .

Definition 2. (ABSTRACT TRANSITION T). An *abstract transition* T is a conjunction of transition predicates. We write $\mathcal{T}_{\mathcal{P}}^{\#}$ for the (finite) set of abstract transitions (where $|\mathcal{P}|$ denotes the cardinality of the set \mathcal{P}):

$$\mathcal{T}_{\mathcal{P}}^{\#} = \{p_1 \wedge \dots \wedge p_n \mid 0 \leq n \leq |\mathcal{P}| \text{ and } p_1, \dots, p_n \in \mathcal{P}\}.$$

An abstract-transition program uses abstract transitions for its node labels. Its definition is relative to a program P , and a set of abstract transitions $\mathcal{T}_{\mathcal{P}}^{\#}$.

Definition 3. (ABSTRACT-TRANSITION PROGRAM $P^{\#}$). An *abstract-transition program* $P^{\#}$ is a finite directed rooted node- and edge-labeled graph with the set of nodes V and the set of edges E . The root node is $v_0 \in V$. It is labeled by the identity relation Id . Every non-root node $v \in V \setminus \{v_0\}$ is labeled by an abstract transition from $\mathcal{T}_{\mathcal{P}}^{\#}$, which we denote by T_v . Every edge is labeled by a transition from \mathcal{T} .

We will often use the set V^- of all *non-root* nodes (in figures illustrating examples, we do not show the root node v_0):

$$V^- = V \setminus \{v_0\}.$$

We illustrate Definition 3 on the abstract-transition program LOOP $^{\#}$, shown in Figure 3. It has the following nodes and edges:

$$\begin{aligned} V &= \{v_0, v_1, v_2\}, \\ E &= \{(v_0, v_1), (v_1, v_1), (v_1, v_2)\}. \end{aligned}$$

The root node v_0 is labeled by the identity relation. The nodes v_1 and v_2 are labeled by the abstract transitions T_1 and T_2 , respectively:

$$\begin{aligned} T_1 &= at_l_0 \wedge at'_l_0 \wedge y > 0 \wedge y' \leq y - 1, \\ T_2 &= at_l_0 \wedge at'_l_2. \end{aligned}$$

The edges (v_0, v_1) and (v_1, v_1) are labeled by the transition τ_1 . The edge (v_1, v_2) is labeled by τ_2 .

We can now define the meaning of abstraction of a program P by an abstract-transition program $P^{\#}$. Later on, we present an algorithm that constructs an abstract-transition program $P^{\#}$ for a given program P and a set of transition predicates \mathcal{P} .

Definition 4. (ABSTRACTION $P \sqsubseteq P^{\#}$). An abstract-transition program $P^{\#}$ is an *abstraction* of the program P when for all nodes v_1 of $P^{\#}$ labeled by, say, the abstract transition T_1 , and for all transitions τ of the program P the following holds:

- if T_1 contains a pair of states (s, s') such that s' goes to some state s'' under the transition τ , then $P^{\#}$ contains:
 - a non-root node v_2 that is labeled by an abstract transition T_2 containing the pair (s, s'') , and
 - an edge from v_1 to v_2 labeled by τ .

Note that the target node v_2 in the definition above must be different from the root node v_0 . However, there may exist a target node v_2 labeled by Id .

In the rest of the paper, the notation $P^\#$ always refers to an abstract-transition program $P^\#$ that is an abstraction of the program P , i.e. $P \sqsubseteq P^\#$.

4. AUTOMATED ABSTRACTION $P \mapsto P^\#$

Given a finite set of transition predicates \mathcal{P} , the algorithm shown in Figure 4 takes a program P and returns a program $P^\#$ that abstracts it, i.e. $P \sqsubseteq P^\#$.

The algorithm constructs the nodes (and edges) of $P^\#$ in a breadth-first manner. The set of nodes whose successors have not been yet explored are kept in the queue Q .

As usual, the symbol \circ denotes the *relational composition* operator:

$$R_1 \circ R_2 = \{(s, s'') \mid \text{exists } s' \text{ such that } (s, s') \in R_1 \text{ and } (s', s'') \in R_2\}.$$

We choose an abstraction function α that maps a binary relation T over states to the smallest abstract transition in $\mathcal{T}_\mathcal{P}^\#$ that contains the relation T . (We may obtain a more precise abstraction by choosing an abstract domain that contains disjunctions of transition predicates.) For example, if the set of transition predicates is

$$\mathcal{P} = \{x \geq 0, x' \leq x - 1, x' = x, x' \geq x + 1\},$$

the relation

$$T = x > 0 \wedge x' = x - 1$$

is abstracted to the abstract transition

$$\alpha(T) = x \geq 0 \wedge x' \leq x - 1.$$

The algorithm implements the abstraction function α using the following equality.

$$\alpha(T) = \bigwedge \{p \in \mathcal{P} \mid T \subseteq p\}$$

Here, the assertions p and T define *binary* instead of unary relations over states, and use unprimed and *primed* variables instead of just unprimed variables. Everything else is as in classical predicate abstraction. That is, a theorem prover is called for each entailment test “ $T \subseteq p$ ”. If n is the number of predicates, then for each newly created node and each transition τ we have n calls to the theorem prover. Thus, the theoretical worst-case number of calls to the theorem prover is the same as in classical predicate abstraction.

The algorithm terminates, since the number of abstract transitions induced by the finite set of transition predicates \mathcal{P} is finite. It constructs an abstract-transition program $P^\#$ that abstracts P , i.e. $P \sqsubseteq P^\#$, by adding new abstract transitions until the conditions in Definition 4 becomes valid.

5. OVERALL METHOD

Our overall method to check a liveness property of a program under fairness assumptions consists of the five steps given in the introduction.

We do not further elaborate the first step, which is the reduction of the verification problem for general temporal properties to the one for fair termination.

```

input
  P: program with finite set of transitions  $\mathcal{T}$ 
   $\mathcal{P}$ : finite set of transition predicates that defines abstraction function
       $\alpha(T) = \bigwedge \{p \in \mathcal{P} \mid T \subseteq p\}$ 
output
  abstract-transition program  $P^\#$  with:
    V: set of nodes labeled by abstract transitions
    E: set of edges labeled by transitions  $\tau$ 
begin
   $v_0 :=$  new node labeled by  $Id$ 
   $V := \{v_0\}$ 
   $E := \emptyset$ 
  Q := empty queue
  enqueue(Q,  $v_0$ )
  while Q not empty do
     $u :=$  dequeue(Q)
    foreach  $\tau \in \mathcal{T}$  do
       $T := \alpha(T_u \circ \rho_\tau)$ 
      if exists  $w \in V^-$  such that  $T = T_w$  then
         $v := w$ 
      else
         $v :=$  new node labeled by T
         $V := V \cup \{v\}$ 
        enqueue(Q, v)
      fi
       $(u, v) :=$  new edge labeled by  $\tau$ 
       $E := E \cup \{(u, v)\}$ 
    od
  od
end.

```

Fig. 4. Transition predicate abstraction $P \mapsto P^\#$.

This step is standard (cf. [Vardi 1991]), and is analogous to the reduction of safety properties to reachability properties.

We have just presented the second step, the transition predicate abstraction-based transformation of the program P into a node- and edge-labeled graph, the *abstract-transition program* $P^\#$. We now fix $P^\#$.

The third step checks, for each non-root node v of $P^\#$, whether its label, the abstract transition T_v , is well-founded (and then marks the node accordingly as ‘terminating’ or not). In fact, our method can be parameterized by the well-foundedness test we apply. Here, we assume that the transition predicates are linear arithmetic formulas (without disjunction). Then, we can apply one of the well-foundedness tests described in [Colón and Sipma 2001; Podelski and Rybalchenko 2004a; Tiwari 2004]. For intuition, the well-foundedness of a relation defined by a conjunctive formula in primed and unprimed variables is the termination of a corresponding program that consists of a single while loop. The loop body only contains a simultaneous (possibly non-deterministic) update statement. For example, the conjunction $x > 0 \wedge x' = x - 1$ corresponds to the program `while(x>0){x:=x-1}`. From our experience, checking well-foundedness of abstract transitions (termination of single while loops) can be done very efficiently. For example, our prototype imple-

ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year.

mentation of [Podelski and Rybalchenko 2004a] handles hundreds of single while loops in seconds.

The only missing link is the fourth step of our overall method: an algorithm on the automaton underlying $P^\#$ that marks nodes as either ‘fair’ or ‘unfair’. Before we give the formal definition of this algorithm for each kind of fairness (impartiality, justice, and compassion in Sections 7, 8, and 9, respectively), we outline the algorithm. The first part of the algorithm computes, for each node v , a set $\text{abc}(\mathcal{L}_v)$ of transitions, i.e. $\text{abc}(\mathcal{L}_v) \subseteq \mathcal{T}$. The second part checks a condition on $\text{abc}(\mathcal{L}_v)$. That condition is specific to the kind of fairness, and is given by Definitions 5, 6, and 7 in Sections 7, 8, and 9 respectively. The algorithm marks the node v according to the outcome of the check.

In its fifth, final step our method returns ‘property verified’ if each ‘fair’ node is marked ‘terminating’. Hence, the correctness of our overall method follows from Theorems 2, 3, and 4 in Sections 7, 8, and 9 respectively, depending on the kind of fairness.

5.1 Finite Automata

We observe that the graph of $P^\#$ without the node labels is the transition graph of a deterministic finite automaton over the alphabet \mathcal{T} . Each node $v \in V$ defines an automaton \mathcal{A}_v whose initial state is the root node v_0 , and whose only final state is the node v .

$$\mathcal{A}_v = \langle \mathcal{T}, V, \delta, v_0, \{v\} \rangle$$

Let E be the set of edges in the graph of $P^\#$. The transition relation δ is the following.

$$\delta = \{(u, \tau, v) \mid (u, v) \in E \text{ is an edge labeled by } \tau\}$$

Let \mathcal{L}_v be the language defined by the automaton \mathcal{A}_v . We next formalize the fact that the language \mathcal{L}_v covers all relevant compositions of transition relations.

LEMMA 1. *Every word $\tau_1 \dots \tau_n$ over transitions in \mathcal{T} is in the language \mathcal{L}_v for some non-root node v , unless the composition of the corresponding transition relations is empty. Formally,*

$$\rho_{\tau_1} \circ \dots \circ \rho_{\tau_n} \neq \emptyset \rightarrow \exists v \in V^- . \tau_1 \dots \tau_n \in \mathcal{L}_v.$$

PROOF. By induction over n . \square

The set $\text{abc}(\mathcal{L}_v)$ consists of all letters appearing in some word in \mathcal{L}_v , i.e. of all transitions $\tau \in \mathcal{T}$ labeling the edges that constitute a path from the root node v_0 to the node v .

$$\text{abc}(\mathcal{L}_v) = \bigcap \{M \subseteq \mathcal{T} \mid \mathcal{L}_v \subseteq M^*\}$$

We compute $\text{abc}(\mathcal{L}_v)$ by traversing backwards the graph of \mathcal{A}_v from the node v .

6. TERMINATION

Before considering fair termination, let us first look at termination. In this section we rephrase the sound and complete proof rule for termination given in [Podelski and Rybalchenko 2004b] in terms of abstract-transition programs. This formulation

is the basis of our correctness proofs. The correctness of the proof rule given in [Podelski and Rybalchenko 2004b] is based on the Ramsey’s argument and uses ideas similar to the proofs of algorithms for the termination analysis of logic and functional programs, e.g. [Sagiv 1991; Dershowitz et al. 2001; Lee et al. 2001; Codish et al. 2003]. We will extend this argument for dealing with impartiality, justice, and compassion in Sections 7, 8, and 9 respectively.

We reason about (plain) termination of a program P by examining the abstract transitions that label the nodes of the abstract-transition program $P^\#$. We do not take the edges of $P^\#$ into account. We say that an abstract transition T is well-founded, written $\text{well-founded}(T)$, if T denotes a well-founded relation, i.e., a relation that does not admit infinite chains.

THEOREM 1. (TERMINATION [PODELSKI AND RYBALCHENKO 2004B]).
The program P terminates if every non-root node in the abstract-transition program $P^\#$ is labeled by a well-founded abstract transition:

$$\forall v \in V^- . \text{well-founded}(T_v).$$

PROOF. Assume that the program P does not terminate. We show that there exists a non-root node v labeled by a non-well-founded abstract transition T_v .

Let $\sigma = s_1, s_2, \dots$ be an infinite computation induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \dots$, where for all $i \geq 1$ we have $(s_i, s_{i+1}) \in \rho_{\tau_i}$. Let $L = 1, 2, \dots$ be the infinite ordered set of positions in σ .

For the fixed sequences ξ and L , we choose a function f that maps a pair of positions (l_i, l_j) , where $i < j$, from L to one of the nodes of the abstract-transition program $P^\#$ in the following way. We define $f(l_i, l_j)$ to be the node v such that the word $\tau_{l_i} \dots \tau_{l_j-1}$, which is a segment of ξ , is in the language \mathcal{L}_v . Such a function f exists, by Lemma 1.

The function f induces an equivalence relation \sim on pairs of elements of L .

$$(l_i, l_j) \sim (l_m, l_n) \quad \text{if and only if} \quad f(l_i, l_j) = f(l_m, l_n)$$

Since the range of f is finite, the equivalence relation \sim has finite index.

By Ramsey’s theorem [Ramsey 1930], there exists an infinite ordered set of positions $K = k_1, k_2, \dots$, where $k_i \in L$ for all $i \geq 1$, that satisfies the following property. All pairs of elements in K belong to the same equivalence class. That is, there exists a non-root node v such that for all $k_i, k_j \in K$ where $i < j$ we have $f(k_i, k_j) = v$. We fix the node v .

Since $f(k_i, k_{i+1}) = v$ for all $i \geq 1$, the infinite sequence s_{k_1}, s_{k_2}, \dots is induced by the relation T_v .

$$(s_{k_i}, s_{k_{i+1}}) \in T_v \quad \text{for all } i \geq 1$$

We conclude that the abstract transition T_v is not well-founded. \square

6.1 Example CHOICE

We use the program CHOICE in Figure 5 to illustrate our method for termination. We compute the abstract-transition program $\text{CHOICE}^\#$, shown in Figure 6, by taking the following set of transition predicates.

$$\mathcal{P} = \{x' \leq x, x' \leq x - 1, x' \leq y - 2, \\ y' \leq y, y' \leq y - 1, y' \leq x + 1, y' \leq x\}$$

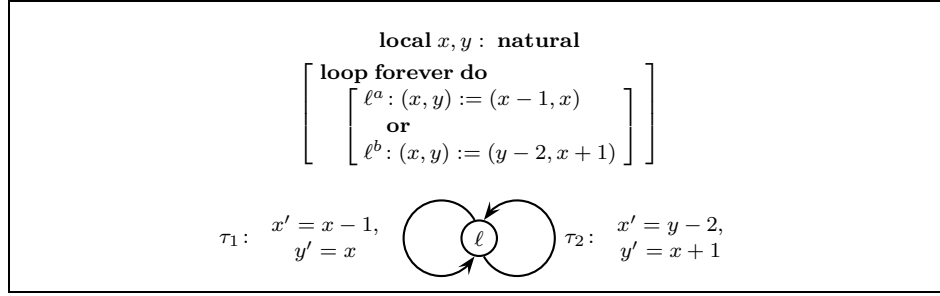


Fig. 5. Program CHOICE.

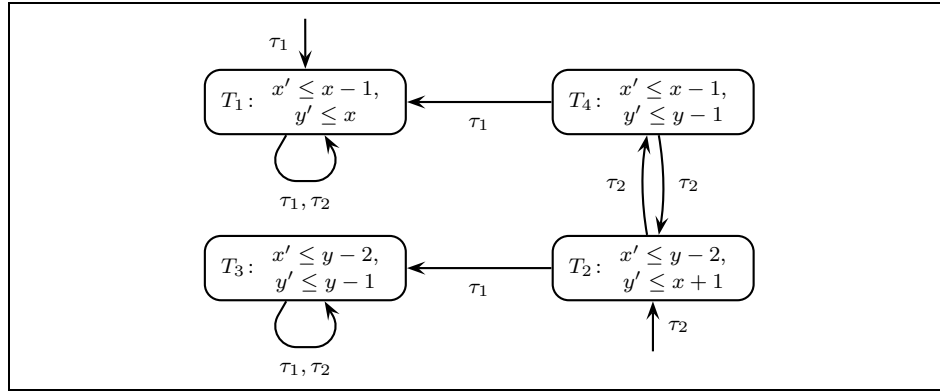


Fig. 6. Abstract-transition program CHOICE#.

The program CHOICE terminates, since every non-root node of CHOICE# is labeled by a well-founded abstract transition.

7. IMPARTIALITY

Impartiality is an unconditional fairness requirement [Lehmann et al. 1981]. Impartiality requirement is represented by a set \mathcal{I} of *impartial* transitions, $\mathcal{I} \subseteq \mathcal{T}$. Every impartial transition must be taken infinitely often.

Definition 5. (IMPARTIALLY FAIR NODES). A non-root node $v \in V^-$ is marked (impartially) ‘fair’ if all impartial transitions are contained in the set of transitions $\text{abc}(\mathcal{L}_v)$ that appear on paths into v :

$$\text{fair}_{\mathcal{I}}(v) = \mathcal{I} \subseteq \text{abc}(\mathcal{L}_v).$$

We say that a program *impartially terminates* if it does not have any infinite computations that satisfy the impartiality requirement.

THEOREM 2. (IMPARTIAL TERMINATION). *The program P impartially terminates if every non-root ‘fair’ marked node v of the abstract-transition program $P^\#$*

is labeled by a well-founded abstract transition T_v :

$$\forall v \in V^-. \text{fair}_{\mathcal{I}}(v) \rightarrow \text{well-founded}(T_v).$$

PROOF. Assume that the program P does not impartially terminate. We show that there exists a node $v \in V$ such that v is labeled by a non-well-founded abstract transition T_v , and the set of transitions $\text{abc}(\mathcal{L}_v)$ contains the set of impartial transitions \mathcal{I} .

Let $\sigma = s_1, s_2, \dots$ be an infinite computation that satisfies the impartiality requirement. Assume that σ is induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \dots$ such that for each $i \geq 1$ we have $(s_i, s_{i+1}) \in \rho_{\tau_i}$.

Let $L = l_1, l_2, \dots$ be an infinite ordered set of positions in the computation σ such that every transition from \mathcal{I} is taken on a state s_p lying between the positions l_i and l_{i+1} , formally

$$\forall \tau \in \mathcal{I} \forall i \geq 1 \exists l_i \leq p < l_{i+1}. \tau_p = \tau.$$

Such a set L exists, since σ satisfies the impartiality requirement \mathcal{I} .

We now apply the same argument as in the proof of Theorem 1. We define an equivalence relation on pairs from the set L and apply Ramsey's theorem. Then, we obtain an infinite ordered set $K = k_1, k_2, \dots$, where $K \subseteq L$, and a non-root node v with the following property. For each pair of elements k_i and k_j in K , where $i < j$, we have $(s_{k_i}, s_{k_j}) \in T_v$. Again, we observe that the abstract transition T_v is not well-founded.

We show that the set of impartial transition \mathcal{I} is contained in the set of transitions $\text{abc}(\mathcal{L}_v)$. By the choice of the set L and taking into consideration that the set K is a subset of L , we have

$$\mathcal{I} \subseteq \{\tau_{k_i}, \dots, \tau_{k_{i+1}-1}\} \quad \text{for all } i \geq 1.$$

Since the sequence of transitions $\tau_{k_i}, \dots, \tau_{k_{i+1}-1}$ labels a path through the graph of $P^\#$ that ends at the state v , we have $\{\tau_{k_i}, \dots, \tau_{k_{i+1}-1}\} \subseteq \text{abc}(\mathcal{L}_v)$. Hence, we conclude $\mathcal{I} \subseteq \text{abc}(\mathcal{L}_v)$. \square

7.1 Example ONLY-DOWN

We illustrate an application of Theorem 2 for proving impartial termination on the example program ONLY-DOWN shown in Figure 7. We obtain the control-flow graph shown in Figure 8 by taking the asynchronous parallel composition of the processes. We assume that the transition τ_1 is impartial:

$$\mathcal{I} = \{\tau_1\}.$$

We compute the abstract-transition program ONLY-DOWN[#], shown in Figure 9, by taking the following set of transition predicates.

$$\mathcal{P} = \{x \geq 0, x' = x, x' \leq x - 1\}$$

The abstract transition T_2 is not well-founded. We observe that τ_1 does not appear on any path that ends in T_2 , i.e., we have $\tau_1 \notin \text{abc}(\mathcal{L}_2)$. Hence, by Theorem 2, the program ONLY-DOWN impartially terminates.

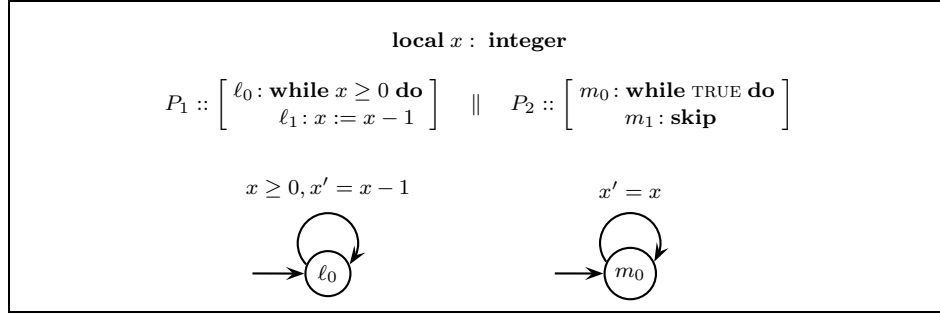


Fig. 7. Program ONLY-DOWN. The statement **skip** only advances the control-flow. It does not change the valuation of program variables.

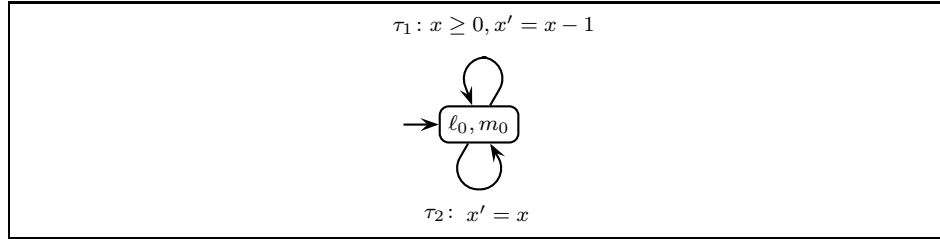


Fig. 8. Control-flow graph for the parallel composition of processes P_1 and P_2 in ONLY-DOWN.

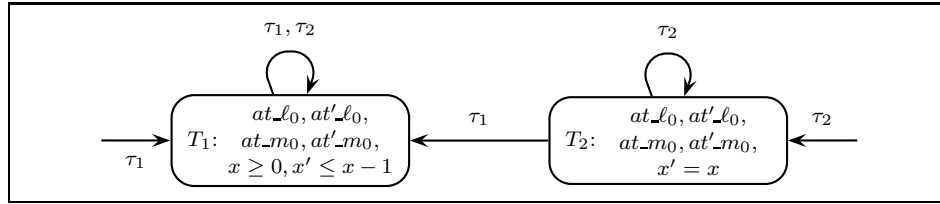


Fig. 9. Abstract-transition program ONLY-DOWN[#].

8. JUSTICE

Justice is a conditional fairness requirement [Manna and Pnueli 1995]. It is sensitive to the enabledness of transitions. A transition τ is *enabled* on the state s if there exists a state s' such that $(s, s') \in \rho_\tau$. We write $\text{En}(\tau)$ for the set of states on which the transition τ is enabled:

$$\text{En}(\tau) = \{s \mid \text{exists } s' \in \Sigma \text{ such that } (s, s') \in \rho_\tau\}.$$

Justice requirement is represented by a set \mathcal{J} of *just* transitions, $\mathcal{J} \subseteq \mathcal{T}$. Every just transition that is continually enabled beyond some point in the computation must be taken infinitely often.

We make the following assumption on the transition relations of the program P .

ASSUMPTION 1. (TRANSITION DISJOINTNESS FOR \mathcal{J}). *Transition relation of*

each just transition is disjoint from the transition relation of every other transition:

$$\forall \tau^j \in \mathcal{J} \forall \tau \in \mathcal{T}. \tau^j \neq \tau \implies \rho_{\tau^j} \cap \rho_{\tau} = \emptyset.$$

This assumption is not a proper restriction. In fact, it is automatically fulfilled by the transition relations of SPL programs. For every pair of transitions τ_ℓ and τ_m that belong to different processes, we have the following transition relations.

$$\begin{aligned} \rho_{\tau_\ell} &= at_l \wedge at'_l' \wedge at_m \wedge at'_m \wedge \dots \\ \rho_{\tau_m} &= at_l \wedge at'_l' \wedge at_m \wedge at'_m' \wedge \dots \end{aligned}$$

Clearly, the relations ρ_{τ_ℓ} and ρ_{τ_m} are disjoint. Transitions that belong to the same process are marked with different labels, so their transition relations are disjoint.

We make the following assumption on the enabledness sets of transition in the program P .

ASSUMPTION 2. (ENABLEDNESS FOR \mathcal{J}). *The enabledness set of each just transition is either disjoint or coincides with the enabledness set of every other transition:*

$$\begin{aligned} \forall \tau^j \in \mathcal{J} \forall \tau \in \mathcal{T}. \tau^j \neq \tau \implies \\ (\text{En}(\tau^j) \cap \text{En}(\tau) = \emptyset \vee \\ \text{En}(\tau^j) = \text{En}(\tau)). \end{aligned}$$

Assumption 2 is not a proper restriction either; for completeness, we give the corresponding syntactic transformation in the appendix.

We define an auxiliary predicate $\text{just}(v, \tau^j)$ as follows:

$$\begin{aligned} \text{just}(v, \tau^j) &= \tau^j \in \text{abc}(\mathcal{L}_v) \vee \\ &\quad \exists \tau \in \text{abc}(\mathcal{L}_v). \text{En}(\tau) \cap \text{En}(\tau^j) = \emptyset. \end{aligned}$$

Informally, $\text{just}(v, \tau^j)$ holds if the transition τ^j is either taken or not continually enabled on some path through the graph of $P^\#$ from the root to the node v . Such transitions contribute to the marking of v as ‘fair’.

Definition 6. (JUSTLY FAIR NODES). A non-root node $v \in V^-$ is marked (justly) ‘fair’ if the predicate $\text{just}(v, \tau^j)$ holds for every just transition:

$$\text{fair}_{\mathcal{J}}(v) = \forall \tau^j \in \mathcal{J}. \text{just}(v, \tau^j).$$

We say that a program *justly terminates* if it does not have infinite computations that satisfy the justice requirement.

THEOREM 3. (JUST TERMINATION). *The program P justly terminates if every non-root ‘fair’ marked node v of the abstract-transition program $P^\#$ is labeled by a well-founded abstract transition T_v :*

$$\forall v \in V^-. \text{fair}_{\mathcal{J}}(v) \implies \text{well-founded}(T_v).$$

PROOF. Assume that the program P does not justly terminate. We show that there exists a non-root node v labeled by a non-well-founded abstract transition T_v , and that for every just transition τ^j the predicate $\text{just}(v, \tau^j)$ holds.

Let $\sigma = s_1, s_2, \dots$ be an infinite computation that satisfies the justice requirement. Assume that σ is induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \dots$, where for each $i \geq 1$ we have $(s_i, s_{i+1}) \in \rho_{\tau_i}$.

The computation σ partitions the set of just transitions \mathcal{J} into the sets $\mathcal{J}^{d(isabled)}$ and $\mathcal{J}^{t(aken)}$ as follows. A transition $\tau \in \mathcal{J}$ is in the set \mathcal{J}^d if it is not continually enabled after some position in σ . Otherwise, i.e., if τ is taken infinitely often, we have $\tau \in \mathcal{J}^t$.

Let $L = l_1, l_2, \dots$ be an infinite ordered set of positions in σ such that for all $i \geq 1$ we have:

—Every transition from \mathcal{J}^d is not enabled on some state s_p lying between the positions l_i and l_{i+1} , formally

$$\forall \tau \in \mathcal{J}^d \forall i \geq 1 \exists l_i \leq p < l_{i+1}. s_p \notin \text{En}(\tau).$$

—Every transition from \mathcal{J}^t is taken on some state s_p lying between the positions l_i and l_{i+1} , formally

$$\forall \tau \in \mathcal{J}^t \forall i \geq 1 \exists l_i \leq p < l_{i+1}. \tau_p = \tau.$$

Such a set L exists since σ satisfies the justice requirement.

We now apply the same argument as in the proof of Theorem 1. We define an equivalence relation on pairs from the set L and apply Ramsey's theorem. Then, we obtain an infinite ordered set $K = k_1, k_2, \dots$, where $K \subseteq L$, and a non-root node v with the following property. For every pair of elements k_i and k_j in K , where $i < j$, we have $(s_{k_i}, s_{k_j}) \in T_v$. Again, we observe that the abstract transition T_v is not well-founded.

We show that each transition $\tau^t \in \mathcal{J}^t$ is contained in the set of transitions $\text{abc}(\mathcal{L}_v)$. By the choice of the set L and taking into consideration that the set K is a subset of L , for each $i \geq 1$ we have

$$\tau^t \in \{\tau_{k_i}, \dots, \tau_{k_{i+1}-1}\}.$$

Since the word $\tau_{k_i} \dots \tau_{k_{i+1}-1}$ is in the language \mathcal{L}_v , we conclude $\tau^t \in \text{abc}(\mathcal{L}_v)$. Hence, $\text{just}(v, \tau^t)$ holds.

We show that for every $\tau^d \in \mathcal{J}^d$ there exists a transition $\tau \in \text{abc}(\mathcal{L}_v)$ such that $\text{En}(\tau) \cap \text{En}(\tau^d) = \emptyset$. By the choice of L , there exists a position p in σ between the positions k_i and k_{i+1} such that the transition τ^d is not enabled on the state s_p . Thus, the transition from the state s_p to its successor state is induced by a transition $\tau \neq \tau^d$. We have $\tau \in \text{abc}(\mathcal{L}_v)$. By Assumption 2, the sets $\text{En}(\tau^d)$ and $\text{En}(\tau)$ are disjoint. Hence, $\text{just}(v, \tau^d)$ holds. We conclude that $\text{fair}_{\mathcal{J}}(v)$ holds. \square

We now illustrate an application of Theorem 3 for proving just termination of example programs.

8.1 Example ANY-DOWN

We show the program ANY-DOWN in Figure 10. We obtain the control-flow graph shown in Figure 11 by taking the asynchronous parallel composition of the processes. We assume that every transition is just:

$$\mathcal{J} = \{\tau_1, \dots, \tau_4\}.$$

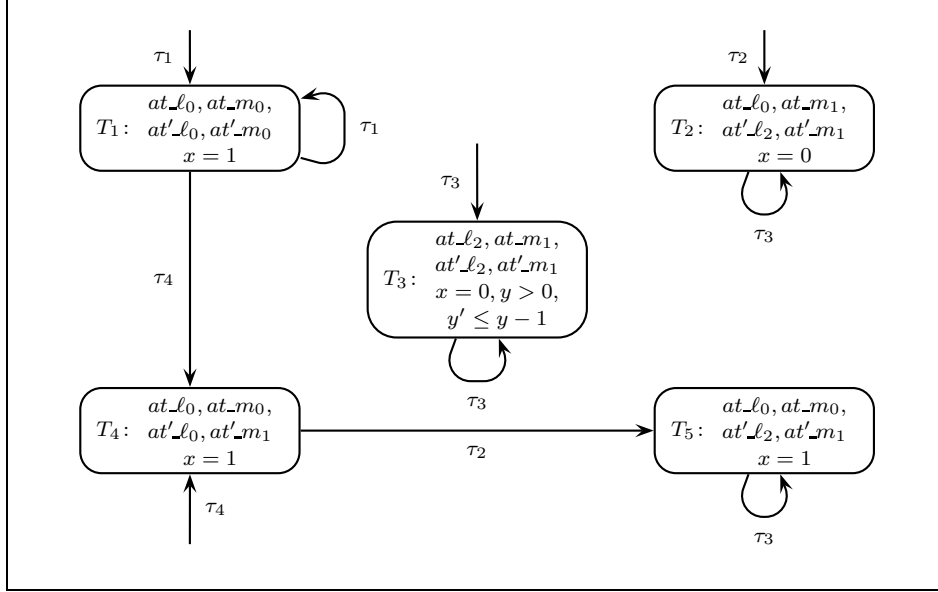


Fig. 12. Abstract-transition program ANY-DOWN#.

8.2 Example ANY-WHILE

We make the program ANY-DOWN more interesting by adding a loop in the second process. The resulting program ANY-WHILE and the control-flow graph for the parallel composition of its processes are shown in Figures 13 and 14 respectively. We assume that every transition is just:

$$\mathcal{J} = \{\tau_1, \dots, \tau_6\}.$$

For the set of transition predicates

$$\mathcal{P} = \{x = 0, x = 1, x' = x, x' = 0, \\ y > 0, y' = y, y' \leq y - 1\}$$

we compute the abstract-transition program ANY-WHILE#, shown in Figure 15.

We observe that the abstract transitions T_1 , T_5 , and T_6 are not well-founded. We read the following sets from the graph of ANY-WHILE#.

$$\text{abc}(\mathcal{L}_1) = \{\tau_1\}$$

$$\text{abc}(\mathcal{L}_5) = \{\tau_5\}$$

$$\text{abc}(\mathcal{L}_6) = \{\tau_6\}$$

Looking at the control-flow graph in Figure 14, we observe the following.

$$\text{En}(\tau_1) = \text{En}(\tau_4)$$

$$\text{En}(\tau_5) = \text{En}(\tau_2)$$

$$\text{En}(\tau_6) = \text{En}(\tau_3)$$

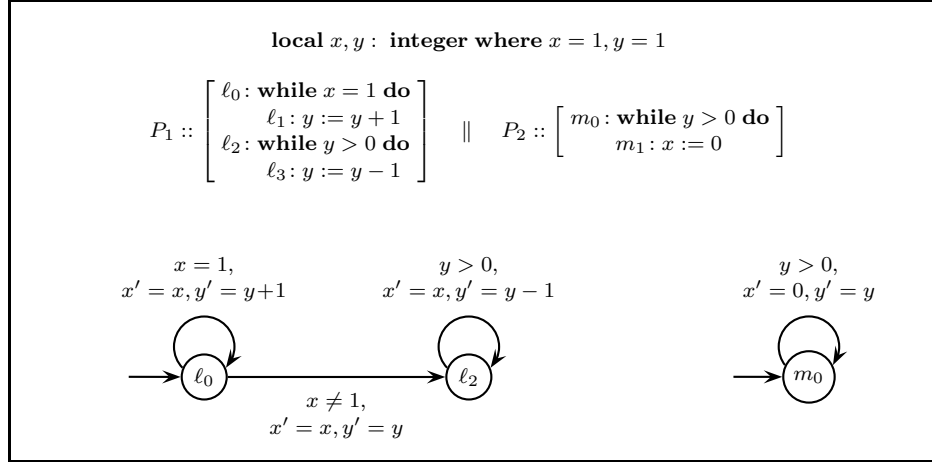
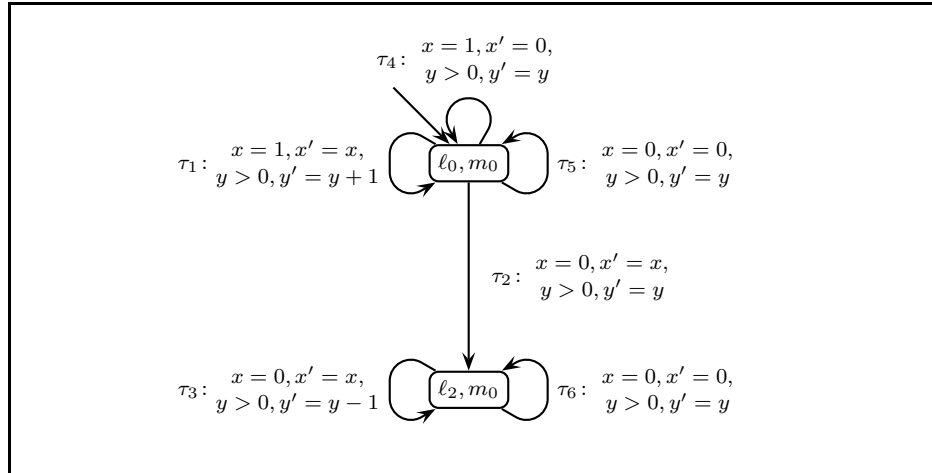


Fig. 13. Program ANY-WHILE.

Fig. 14. Control-flow graph for the parallel composition of the processes P_1 and P_2 in ANY-WHILE.

This means that the predicates $\text{just}(1, \tau_4)$, $\text{just}(5, \tau_2)$, and $\text{just}(6, \tau_3)$ do not hold. Hence, the well-foundedness of T_1, T_5 , and T_6 is not required for the just termination. We conclude that ANY-WHILE justly terminates.

9. COMPASSION

Compassion is another conditional fairness requirement [Manna and Pnueli 1995]. Compared to justice, it is not sensitive to the interruption of transition enabledness infinitely many times. Compassion requirement is represented by a set \mathcal{C} of *compassionate* transitions, $\mathcal{C} \subseteq \mathcal{T}$. Every compassionate transition that is enabled infinitely often must be taken infinitely often.

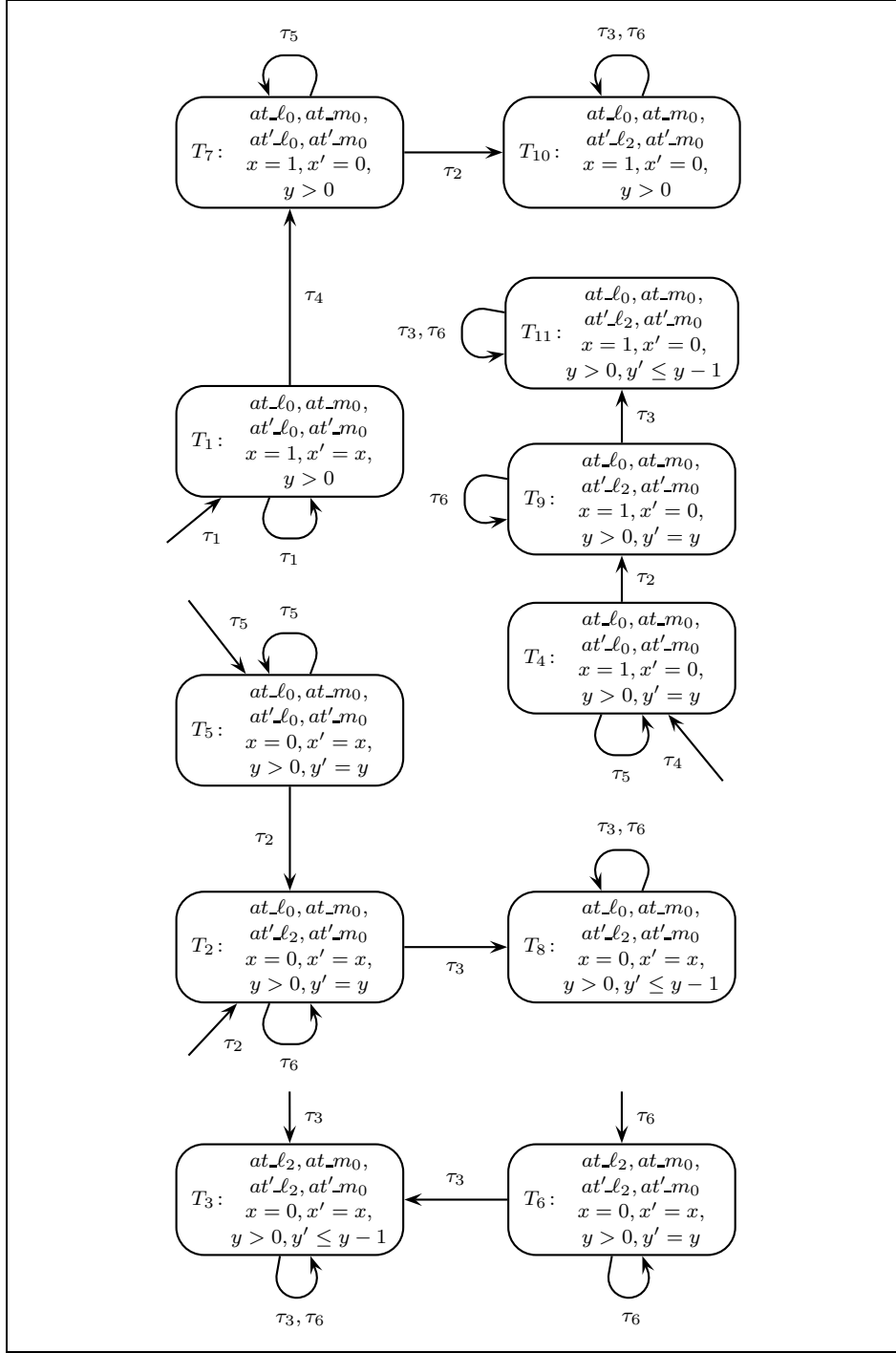


Fig. 15. Abstract-transition program ANY-WHILE#.

We extend Assumptions 1 and 2 to compassionate transitions. This extension is not a proper restriction (see the appendix for details).

For dealing with compassion, we are interested in the set of letters (transitions) $\text{abc}(\bigcap \mathcal{L}_v)$ that appear in every word of the language \mathcal{L}_v :

$$\text{abc}(\bigcap \mathcal{L}_v) = \{\tau \mid \mathcal{L}_v \cap (\mathcal{T} \setminus \{\tau\})^* = \emptyset\}.$$

We compute the set $\text{abc}(\bigcap \mathcal{L}_v)$ by a standard algorithm, which involves a backward graph traversal starting from v and computing intersections over all paths.

We define an auxiliary predicate $\text{comp}(v, \tau^c)$ as follows:

$$\begin{aligned} \text{comp}(v, \tau^c) &= \tau^c \in \text{abc}(\mathcal{L}_v) \vee \\ &\forall \tau \in \text{abc}(\bigcap \mathcal{L}_v). \text{En}(\tau) \cap \text{En}(\tau^c) = \emptyset. \end{aligned}$$

Informally, $\text{comp}(v, \tau^c)$ holds if the transition τ^j is either taken or possibly continually disabled on some path from the root to the node v .

Definition 7. (COMPASSIONATELY FAIR NODES). A non-root node $v \in V^-$ is marked (compassionately) ‘fair’ if the predicate $\text{comp}(v, \tau^c)$ holds for every compassionate transition:

$$\text{fair}_{\mathcal{C}}(v) = \forall \tau^c \in \mathcal{C}. \text{comp}(v, \tau^c).$$

We say that a program *compassionately terminates* if it does not have infinite computations that satisfy the compassion requirement.

THEOREM 4. (COMPASSIONATE TERMINATION). *The program P compassionately terminates if every non-root ‘fair’ marked node v of the abstract-transition program $P^\#$ is labeled by a well-founded abstract transition T_v :*

$$\forall v \in V^-. \text{fair}_{\mathcal{C}}(v) \implies \text{well-founded}(T_v).$$

PROOF. Assume that the program P does not compassionately terminate. We show that there exists a non-root node v labeled by a non-well-founded abstract transition T_v , and that for every compassionate transition τ^c the predicate $\text{comp}(v, \tau^c)$ holds.

Let $\sigma = s_1, s_2, \dots$ be an infinite computation induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \dots$, where for all $i \geq 1$ we have $(s_i, s_{i+1}) \in \rho_{\tau_i}$, that satisfies the compassion requirement.

The computation σ partitions the set of compassionate transitions \mathcal{C} into the sets $\mathcal{C}^{d(\text{isabled})}$ and $\mathcal{C}^{t(\text{aken})}$ as follows. A transition $\tau \in \mathcal{C}$ is in the set \mathcal{C}^d if it is not enabled infinitely often, i.e., is continually disabled after some position in σ . Otherwise, i.e., if τ is taken infinitely often, we have $\tau \in \mathcal{C}^t$.

Let $L = l_1, l_2, \dots$ be an infinite ordered set of positions in σ such that:

—Every transition $\tau \in \mathcal{C}^d$ is not enabled on states at positions after l_1 . Formally,

$$\forall \tau \in \mathcal{C}^d \forall p \geq l_1. s_p \notin \text{En}(\tau).$$

—Every transition $\tau \in \mathcal{C}^t$ is taken between the positions l_i and l_{i+1} for all $i \geq 1$. Formally,

$$\forall \tau \in \mathcal{C}^t \forall i \geq 1 \exists l_i \leq p < l_{i+1}. \tau_p = \tau.$$

We now apply the same argument as in the proof of Theorem 1. We define an equivalence relation on pairs from the set L and apply Ramsey's theorem. Then, we obtain an infinite ordered set $K = k_1, k_2, \dots$, where $K \subseteq L$, and a non-root node v with the following property. For every pair of elements k_i and k_j in K , where $i < j$, we have $(s_{k_i}, s_{k_j}) \in T_v$. Again, we observe that the abstract transition T_v is not well-founded.

Furthermore, since every transition τ^t from \mathcal{C}^t is taken on a state between the positions k_i and k_{i+1} for all $i \geq 1$, we conclude that \mathcal{C}^t is contained in the set of transitions $\text{abc}(\mathcal{L}_v)$. Hence, $\text{comp}(v, \tau^t)$ holds.

By the choice of L , a transition $\tau^d \in \mathcal{C}^d$ is not enabled on the state s_p for every position p in σ after the position k_1 . Since every transition $\tau \in \text{abc}(\bigcap \mathcal{L}_v)$ must appear between the positions k_i and k_{i+1} , we conclude that there exists a state s such that $s \in \text{En}(\tau)$ and $s \notin \text{En}(\tau^d)$. By Assumption 2 (which we extended to the compassionate transitions), the sets $\text{En}(\tau^d)$ and $\text{En}(\tau)$ are disjoint. Hence, $\text{comp}(v, \tau^d)$ holds. We conclude that $\text{fair}_{\mathcal{C}}(v)$ holds. \square

9.1 Example SUB-SKIP

We illustrate Theorem 4 on the program SUB-SKIP, shown in Figure 16. We assume the following set of compassionate transitions \mathcal{C} :

$$\mathcal{C} = \{\tau_2, \tau_3\}.$$

Every infinite computation of SUB-SKIP may take the transition τ_2 only finitely many times, although it is enabled infinitely often, thus, violating the compassion requirement \mathcal{C} .

We show the abstract transition program SUB-SKIP[#] in Figure 17. We compute SUB-SKIP[#] by applying the set of transition predicates below.

$$\mathcal{P} = \{y > 0, y' \leq y, y' \leq y - 1\}$$

The only non-well-founded abstract transitions are T_5 and T_7 . We show that according to Theorem 4, the well-foundedness of these two abstract transitions is not needed for proving compassionate termination. We show that the predicates $\text{comp}(5, \tau_2)$ and $\text{comp}(7, \tau_2)$ do not hold.

From Figure 17, we obtain the following sets of transitions.

$$\text{abc}(\mathcal{L}_5) = \text{abc}(\mathcal{L}_7) = \text{abc}\left(\bigcap \mathcal{L}_5\right) = \text{abc}\left(\bigcap \mathcal{L}_7\right) = \{\tau_1, \tau_3\}$$

Furthermore, we observe (in Figure 16) that $\text{En}(\tau_2) = \text{En}(\tau_3)$. Hence, the predicates $\text{comp}(5, \tau_2)$ and $\text{comp}(7, \tau_2)$ do not hold.

10. LEXICOGRAPHIC COMPLETENESS

Our main interest is in fair termination. But let us look again at termination. This allows us to compare the power of transition predicate abstraction with the classical means to construct termination arguments for programs with nested loops, which is the lexicographic combination of ranking functions (see e.g. [Manna and Pnueli 1996]). We show that if each lexicographic component of a ranking function for the program can be expressed by some conjunction of transition predicates in \mathcal{P} then transition predicate abstraction will construct a termination argument for the program (see Theorem 1).

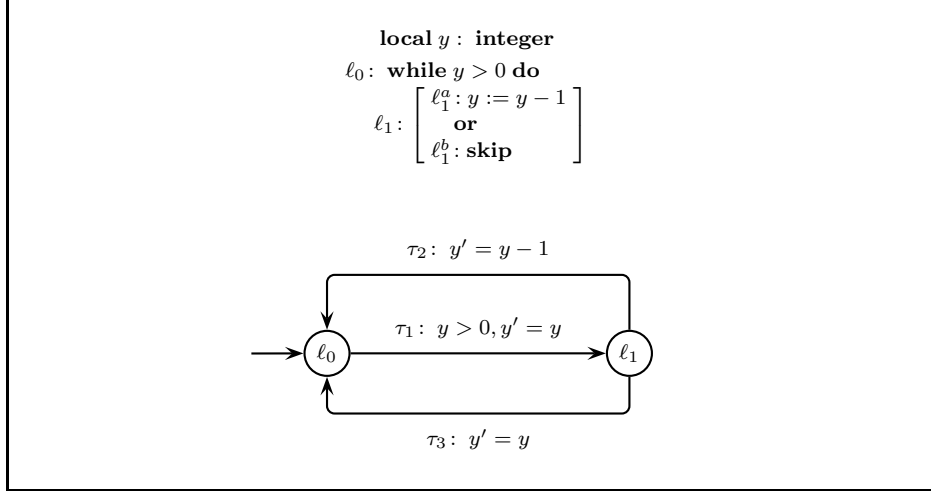


Fig. 16. Program SUB-SKIP.

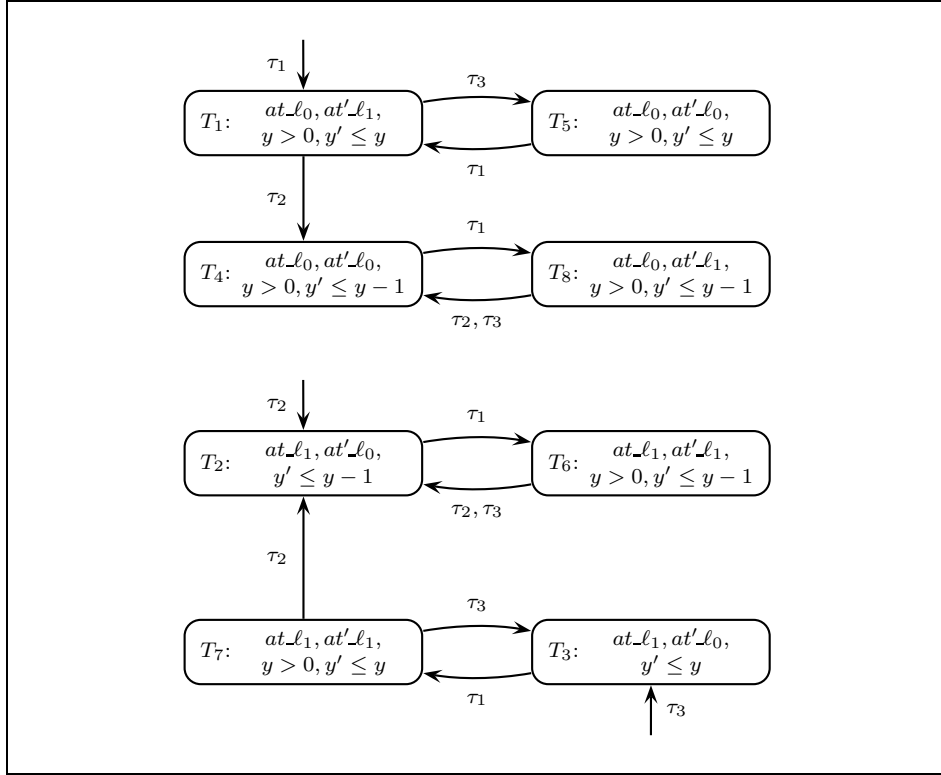


Fig. 17. Abstract-transition program SUB-SKIP#.

Let (f_1, \dots, f_n) be a tuple of functions from the set of states Σ into the domains $(\mathcal{W}_1, \succ_1), \dots, (\mathcal{W}_n, \succ_n)$ such that \succ_i is an ordering relation, i.e. transitive and irreflexive, for each $1 \leq i \leq n$. We define an auxiliary predicate $\text{lex}(R, j)$ on relations R and indices $j \in \{1, \dots, n\}$:

$$\begin{aligned} \text{lex}(R, j) = \forall (s, s') \in R. f_j(s) \succ_j f_j(s') \wedge \\ \forall 1 \leq i < j. f_i(s) \succeq_i f_i(s'). \end{aligned}$$

For each function f_i we define a pair $f_i \succ_i f'_i$ and $f_i \succeq_i f'_i$ of transition predicates.

$$\begin{aligned} f_i \succ_i f'_i &= \{(s, s') \mid f_i(s) \succ_i f_i(s')\} \\ f_i \succeq_i f'_i &= \{(s, s') \mid f_i(s) \succeq_i f_i(s')\} \end{aligned}$$

Following [Baader and Nipkow 1998], the tuple (f_1, \dots, f_n) is a *lexicographic ranking function* for the program P if each ordering \succ_i is well-founded and for every transition τ there exists an index $j \in \{1, \dots, n\}$ such that $\text{lex}(\rho_\tau, j)$.

For example, the function $f(x, y) = x + y$, where the variables x and y range over integers, into the set of natural numbers defines the transition predicates $x + y > x' + y'$ and $x + y \geq x' + y'$.

THEOREM 5. (LEXICOGRAPHIC COMPLETENESS). *If the set $\mathcal{T}_P^\#$ generated by the set of transition predicates \mathcal{P} contains the relation $f_i \succ_i f'_i$ and the relation $f_i \succeq_i f'_i$ for every component f_i of the lexicographic ranking function (f_1, \dots, f_n) for the program P , then every non-root node of the abstract program $P^\#$ obtained by transition predicate abstraction algorithm is labeled by a well-founded abstract transition.*

PROOF. Let the tuple (f_1, \dots, f_n) be a lexicographic ranking function for the program P such that the transition predicates $f_i \succ_i f'_i$ and $f_i \succeq_i f'_i$ are contained in the set of abstract transitions $\mathcal{T}_P^\#$ for each component f_i of the tuple.

We prove for each non-root node v , by induction over the length of a shortest path from the root node v_0 to the node v , that there exists an index $j \in \{1, \dots, n\}$ such that the predicate $\text{lex}(T_v, j)$ holds. The well-foundedness of T_v follows directly.

For the base case, let τ be the transition that labels the edge from the node v_0 to the node v . Since $\text{lex}(\rho_\tau, j)$ holds for some $j \in \{1, \dots, n\}$, we have

$$\begin{aligned} \rho_\tau \subseteq f_j \succ_j f'_j \in \mathcal{T}_P^\#, \\ \forall 1 \leq i < j. \rho_\tau \subseteq f_i \succeq_i f'_i \in \mathcal{T}_P^\#. \end{aligned}$$

Since α is the ‘best-abstraction’ function, we have

$$\begin{aligned} \alpha(\rho_\tau) \subseteq f_j \succ_j f'_j, \\ \forall 1 \leq i < j. \alpha(\rho_\tau) \subseteq f_i \succeq_i f'_i. \end{aligned}$$

Hence, we conclude $\text{lex}(T_v, j)$ where $T_v = \alpha(\rho_\tau)$.

For the induction step, let u be a predecessor node of a non-root node v such that u is on a shortest path from v_0 to v . Let the predicate $\text{lex}(T_u, j)$ hold for some index $j \in \{1, \dots, n\}$. For a transition τ that labels the edge (u, v) there exists an index $l \in \{1, \dots, n\}$ such that $\text{lex}(\rho_\tau, l)$ holds. Let $m = \min(j, l)$. We show that $\text{lex}(T_v, m)$ holds.

By the induction hypothesis, we have

$$\begin{aligned} T_u &\subseteq f_j \succ_j f'_j, \\ \forall 1 \leq i < j. T_u &\subseteq f_i \succeq_i f'_i. \end{aligned}$$

From $\text{lex}(\rho_\tau, l)$ we have

$$\begin{aligned} \rho_\tau &\subseteq f_l \succ_l f'_l, \\ \forall 1 \leq k < l. \rho_\tau &\subseteq f_k \succeq_k f'_k. \end{aligned}$$

By the transitivity of \succ_i for $1 \leq i \leq n$, we have

$$\begin{aligned} T_u \circ \rho_\tau &\subseteq f_m \succ_m f'_m, \\ \forall 1 \leq i < m. T_u \circ \rho_\tau &\subseteq f_i \succeq_i f'_i. \end{aligned}$$

Analogously to the base case, we conclude $\text{lex}(T_v, m)$, where $T_v = \alpha(T_u \circ \rho_\tau)$. \square

The following example illustrates that transition predicate abstraction may apply to programs whose termination cannot be proven by lexicographic ranking functions whose components are contained in $\mathcal{T}_P^\#$.

10.1 Example CHOICE[#] and Lexicographic Rankings

We again consider the terminating program CHOICE, discussed in Section 6. As one can easily see, no lexicographic combination of the functions

$$f_1(x, y) = x, \quad f_2(x, y) = y, \quad f_3(x, y) = x + y$$

is a ranking function for CHOICE. Executing the transition τ_1 may strictly increase the value of y and $x + y$, and executing the transition τ_2 the value of x or y may increase.

The abstract-transition program CHOICE[#], shown in Figure 6, was computed by taking the following set of transition predicates.

$$\begin{aligned} \mathcal{P} = \{ &x' \leq x, x' \leq x - 1, x' \leq y - 2, \\ &y' \leq y, y' \leq y - 1, y' \leq x + 1, y' \leq x \} \end{aligned}$$

Note that the set of abstract transition $\mathcal{T}_P^\#$ induced by the transition predicates above contains the transition predicates $f_i \succ_i f'_i$ and $f_i \succeq_i f'_i$ for each $i \in \{1, 2, 3\}$ (and no other ranking functions.)

11. CONCLUSION

In this paper, we have proposed the extension of predicate abstraction to transition predicate abstraction as a way to overcome the inherent limitation of predicate abstraction to safety properties. Previously, the only known way to overcome this limitation was to annotate the finite-state abstraction of a program in a process that involved the manual construction of ranking functions. We have gone beyond the idea of abstracting a program to a finite-state program and checking the absence of loops in its finite graph. Instead, we have given the transformation of a program into a finite *abstract-transition* program. We have given algorithms to check fair termination on the abstract-transition program. The two algorithms together yield an automated method for the verification of liveness properties under full fairness

assumptions (impartiality, justice, and compassion). In conclusion, we have exhibited principles that extend the applicability of predicate abstraction-based program verification to the full set of temporal properties.

We believe that our work may trigger a series of activities to develop tools for checking liveness, similar to the series of activities that have led to the success of tools for safety and invariance properties [Graf and Saïdi 1997; Ball et al. 2001; Yahav 2001; Henzinger et al. 2002; Chaki et al. 2003].

In a heuristics-based approach for finding an appropriate set of transition predicates, one would first examine guards (which yields the special case of transition predicates that are assertions without primed variables such as $x > 0$) and update statements $x := e$ (which yields transition predicates of the form $x' \leq e$ and $x' \geq e$). Although this heuristics may be useful in some experiments for the proof of principle, automated counterexample-driven abstraction refinement will be required at some point in the development of tools for automated liveness proofs. We have come up with a very general algorithmic scheme for verification of termination (not general liveness) with automated counterexample-driven abstraction refinement in [Cook et al. 2005].

We also have developed a method that allows one to directly utilize a predicate abstraction-based software model checker for the computation of abstract-transition programs [Cook et al. 2006]. The first experimental results are promising. Our implementation can analyze the termination of dispatch routines of devices drivers written in the C programming language. It handles program fragments of more than 20,000 lines of code. A possible direction for future work is to investigate whether the existing techniques to speed up predicate abstraction, e.g. [Ball et al. 2001; Flanagan and Qadeer 2002; Lahiri et al. 2003], are applicable for transition predicate abstraction.

Our algorithm suggests a verification methodology where the input to the algorithm is a liveness property without fairness assumptions. One then takes the computed abstract-transition program and its node labeling (‘terminating’ or not) to derive what fairness assumptions are required for the liveness property to hold. It should be possible to automate this derivation step.

APPENDIX

For completeness, we give the syntactic transformation for Assumptions 1 and 2 (which we extended to the compassionate transitions in Section 9).

We replace every fair transition $\tau \in \mathcal{J} \cup \mathcal{C}$ by a set of transitions obtained as follows. For each bit-vector over the enabledness sets of transitions $\mathcal{T} \setminus \{\tau\}$ we create a new transition with the transition relation obtained from ρ_τ by intersecting its enabledness set $\text{En}(\tau)$ with the set defined by the bit-vector. The following conditions hold for the transition relations and the enabledness sets obtained by splitting the transition τ into the set of transitions $\{\tau_1, \dots, \tau_n\}$.

$$\text{En}(\tau) = \text{En}(\tau_1) \uplus \dots \uplus \text{En}(\tau_n) \quad (1a)$$

$$\rho_\tau = \rho_{\tau_1} \uplus \dots \uplus \rho_{\tau_n} \quad (1b)$$

The set of just (compassionate) transitions \mathcal{J} (\mathcal{C}) of the program is modified by replacing τ by the set $\{\tau_1, \dots, \tau_n\}$.

We show that the above modification preserves the fair termination property.

LEMMA 2. *The program P with the set of just transitions \mathcal{J} justly terminates if it justly terminates after replacing each just transition by the set of transitions satisfying (1).*

PROOF. Assume that there exists an infinite computation $\sigma = s_1, s_2, \dots$ of the original program that satisfies the justice requirement \mathcal{J} . Since partitioning does not make the transition relation of the program smaller, see (1b), σ is a computation of the modified program.

We show that for every $\tau \in \mathcal{J}$ replaced by the set of transitions $\{\tau_1, \dots, \tau_n\}$, the computation σ satisfies the justice requirement for each τ_i , where $1 \leq i \leq n$.

If τ is disabled infinitely often then each of τ_i , for $1 \leq i \leq n$, is disabled infinitely often. If τ is continually enabled, and, hence, infinitely often taken, we consider the following two cases.

We assume that there exists an enabledness set $\text{En}(\tau_j)$ for some $1 \leq j \leq n$ such that σ eventually does not leave the set $\text{En}(\tau_j)$, formally,

$$\exists 1 \leq j \leq n \exists k \geq 1 \forall l \geq k. s_l \in \text{En}(\tau_j).$$

Every transition τ_i , where $1 \leq i \neq j \leq n$, is not continually enabled, by Assumption 2. The transition τ_j is taken infinitely often, by Assumption 1.

If the assumption above does not hold, then none of the transitions τ_i , for $1 \leq i \leq n$, is continually enabled. \square

LEMMA 3. *The program P with the set of compassionate transitions \mathcal{C} compassionately terminates if it compassionately terminates after replacing each compassionate transition by the set of transitions satisfying (1).*

PROOF. Assume that there exists an infinite computation $\sigma = s_1, s_2, \dots$ of the original program that satisfies the compassion requirement \mathcal{C} . Since partitioning does not make the transition relation of the program smaller, see (1b), σ is a computation of the modified program.

We show that for each $\tau \in \mathcal{C}$ replaced by the set of transitions $\{\tau_1, \dots, \tau_n\}$, the computation σ satisfies the compassion requirement for each τ_i , where $1 \leq i \leq n$.

If τ is not enabled infinitely often then each of τ_i , for $1 \leq i \leq n$, is not enabled infinitely often. If τ is enabled often, and, hence, infinitely often taken, we consider the following two cases.

For each $1 \leq j \leq n$ such that $\text{En}(\tau_j)$ is visited infinitely often, by Assumptions 1 and 2 (extended to compassionate transitions), the transition τ_j is taken infinitely often. All other transitions are not enabled infinitely often. \square

ACKNOWLEDGMENTS

We thank Bruno Blanchet, Bernd Finkbeiner, Leslie Lamport, Alexander Malkis, and anonymous referees for comments and suggestions.

REFERENCES

- ARTS, T. AND GIESL, J. 2000. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.* 236, 1–2, 133–178.
- BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That*. Cambridge University Press.
- ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year.

- BALL, T. 2005. A theory of predicate-complete test coverage and generation. In *FMCO'2004: Symp. on Formal Methods for Components and Objects*. LNCS, vol. 3657. Springer, 1–22.
- BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. 2001. Automatic predicate abstraction of C programs. In *PLDI'2001: Programming Language Design and Implementation*. ACM SIGPLAN Notices, vol. 36. ACM Press, 203–213.
- BALL, T., PODELSKI, A., AND RAJAMANI, S. K. 2001. Boolean and cartesian abstractions for model checking C programs. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, vol. 2031. Springer, 268–283.
- BOURDONCLE, F. 1993. Abstract debugging of higher-order imperative languages. In *PLDI'1993: Programming Language Design and Implementation*. ACM Press, 46–55.
- BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. 2005. Linear ranking with reachability. In *CAV'2005: Computer Aided Verification*. LNCS, vol. 3576. Springer, 491–504.
- BROWNE, I., MANNA, Z., AND SIPMA, H. 1995. Generalized verification diagrams. In *FSTTCS'1995: Foundations of Software Technology and Theoretical Computer Science*. LNCS, vol. 1026. Springer, 484–498.
- BÜCHI, J. R. 1960. On a decision method in restricted second order arithmetic. In *Int. Congress on Logic, Methodology and Philosophy of Science*. Stanford University Press, 1–11.
- CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in C. In *ICSE'2003: Int. Conf. on Software Engineering*. IEEE Computer Society, 385–395.
- CODISH, M., GENAIM, S., BRUYNNOGHE, M., GALLAGHER, J., AND VANHOOF, W. 2003. One loop at a time. In *6th International Workshop on Termination (WST 2003)*. Universidad Politécnic de Valencia.
- COLÓN, M. AND SIPMA, H. 2001. Synthesis of linear ranking functions. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, vol. 2031. Springer, 67–81.
- COLÓN, M. AND URIBE, T. 1998. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV'1998: Computer Aided Verification*. LNCS, vol. 1427. Springer, 293–304.
- COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2005. Abstraction refinement for termination. In *SAS'2005: Static Analysis Symposium*. LNCS, vol. 3672. Springer, 87–101.
- COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation*. ACM.
- COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'1977: Principles of Programming Languages*. ACM Press, 238–252.
- COUSOT, P. AND COUSOT, R. 1994. Higher-order abstract interpretation (and application to compartment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *ICCL'1994: Int. Conf. on Computer Languages*. IEEE, 95–112.
- DELZANNO, G. AND PODELSKI, A. 2001. Constraint-based deductive model checking. *Int. Journal on Software Tools for Technology Transfer (STTT)* 3, 3, 250–270.
- DERSHOWITZ, N., LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing* 12, 117–156.
- FLANAGAN, C. AND QADEER, S. 2002. Predicate abstraction for software verification. In *POPL'2002: Principles of Programming Languages*. ACM Press, 191–202.
- FRANCEZ, N. 1986. *Fairness*. Springer.
- GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *CAV'1997: Computer Aided Verification*. LNCS, vol. 1254. Springer, 72–83.
- HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *POPL'2002: Principles of Programming Languages*. ACM Press, 58–70.
- KESTEN, Y. AND PNUELI, A. 2000. Verification by augmented finitary abstraction. *Information and Computation, a special issue on Compositionality* 163, 1, 203–243.
- KESTEN, Y., PNUELI, A., AND VARDI, M. Y. 2001. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences* 62, 4, 668–690.

- KLARLUND, N. 1992. Progress measures and stack assertions for fair termination. In *PODC'1992: Principles of Distributed Computing*. ACM Press, 229–240.
- LAHIRI, S. K., BRYANT, R. E., AND COOK, B. 2003. A symbolic approach to predicate abstraction. In *CAV'2003: Computer Aided Verification*. LNCS, vol. 2725. Springer, 141–153.
- LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. 2001. The size-change principle for program termination. In *POPL'2001: Principles of Programming Languages*. ACM SIGPLAN Notices, vol. 36, 3. ACM Press, 81–92.
- LEHMANN, D., PNUELI, A., AND STAVI, J. 1981. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP'1981: Int. Colloq. on Automata, Languages and Programming*. LNCS, vol. 115. Springer, 264–277.
- MANNA, Z. AND PNUELI, A. 1995. *Temporal verification of reactive systems: Safety*. Springer.
- MANNA, Z. AND PNUELI, A. 1996. Temporal verification of reactive systems: Progress. Draft.
- MERZ, S. 1997. Rules for abstraction. In *ASIAN'1997: Advances in Computer Science*. LNCS, vol. 1345. Springer, 32–45.
- PNUELI, A., XU, J., AND ZUCK, L. 2002. Liveness with $(0, 1, \infty)$ -counter abstraction. In *CAV'2002: Computer Aided Verification*. LNCS, vol. 2404. Springer, 107–122.
- PODELSKI, A. AND RYBALCHENKO, A. 2004a. A complete method for the synthesis of linear ranking functions. In *VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*. LNCS, vol. 2937. Springer, 239–251.
- PODELSKI, A. AND RYBALCHENKO, A. 2004b. Transition invariants. In *LICS'2004: Logic in Computer Science*. IEEE, 32–41.
- RAMSEY, F. P. 1930. On a problem of formal logic. In *Proc. London Math. Soc.* Vol. 30. 264–285.
- SAGIV, Y. 1991. A termination test for logic programs. In *ISLP'1991: Int. Logic Programming Symp.* 518–532.
- SIPMA, H., URIBE, T., AND MANNA, Z. 1996. Deductive model checking. In *CAV'1996: Computer Aided Verification*. LNCS, vol. 1102. Springer, 208–219.
- TIWARI, A. 2004. Termination of linear programs. In *CAV'2004: Computer Aided Verification*. LNCS, vol. 3114. Springer, 70–82.
- URIBE, T. 1999. Abstraction-based deductive-algorithmic verification of reactive systems. Ph.D. thesis, Stanford University.
- VARDI, M. Y. 1991. Verification of concurrent programs — the automata-theoretic framework. *Annals of Pure and Applied Logic* 51, 79–98.
- YAHAV, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL'2001: Principles of Programming Languages*. ACM Press, 27–40.
- YAHAV, E., REPS, T., SAGIV, M., AND WILHELM, R. 2003. Verifying temporal heap properties specified via evolution logic. In *ESOP'2003: European Symp. on Programming*. LNCS, vol. 2618. Springer, 204–222.