Transition Predicate Abstraction and Fair Termination*

Andreas Podelski Andrey Rybalchenko

Max-Planck-Institut für Informatik
Saarbrücken, Germany

ABSTRACT

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 (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.

Keywords

Software model checking, transition predicate abstraction, fair termination, liveness.

*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.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

1. INTRODUCTION

Since 1977, a high amount of research, both theoretical and applied, has been invested in honing the tools for abstract interpretation [10] 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 [2, 6, 14, 15, 30].

What has been left open is how to obtain the same kind of tools for the full set 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-labeled edge-labeled graph such that the termination property can be retrieved from the node labels and the fairness assumptions from the edge labels. (To avoid the possibility of confusion, note that our method does not check the absence of loops in the graph.) 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.* [29]);
- the transition predicate abstraction-based transformation of the program P into a node-labeled 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 both relevant kinds of fairness, which are justice and compassion (to model the assumption that a transition is eventually taken if it is continually resp. infinitely often enabled).

2. RELATED WORK

Our work is most closely related to the work on predicate abstraction; see *e.g.* [2, 6, 14, 15, 30]. 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.* [1].

Unfortunately, predicate abstraction is inherently limited to safety properties. That is because every sufficiently long computation of the program (with the length greater then the number of abstract states) results in a computation of the abstract system that contains a loop. I.e., termination (as well as more general liveness properties) cannot be preserved by predicate abstraction.

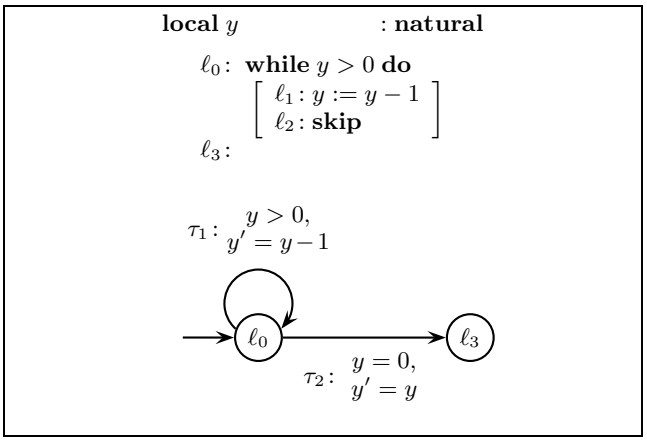

Figure 1: Terminating program LOOP.

We illustrate the limitation on a very simple program LOOP [16], shown on 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 into zero and pos. The corresponding abstraction transforms the program LOOP into the finite-state abstract program shown on Figure 2. That program contains a self-loop, *i.e.* is not terminating. The abstract state S_1 corresponds to the conjunction $at\_\ell_0 \wedge y > 0$ 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

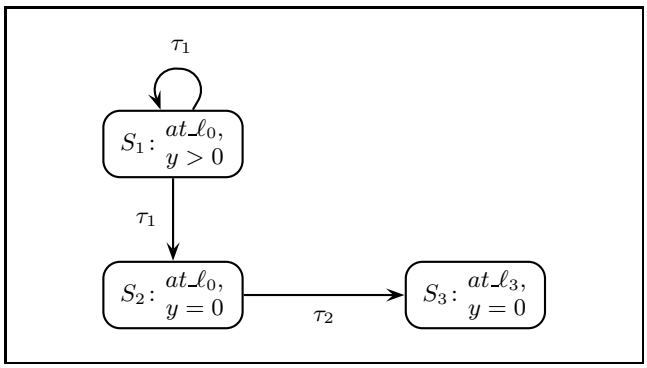

Figure 2: Non-terminating abstract-state program for LOOP.

progress monitors or the like [16, 17, 22, 31]. 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 inherit limitation of predicate abstraction to safety properties. In contrast, the method that we propose does not require the manual construction of termination arguments.

In [24] 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 [24]. In [24], we used an automata-theoretic notion of state-based fairness to formalize a uniform setting. Here we use justice and compassion, two transition-based notions of fairness. These are the two notions of fairness that are relevant with concrete concurrent programs. It is widely accepted that one needs a direct treatment of justice and compassion since the translation to the automata-theoretic notion is prohibitively expensive. As a consequence, the notion of transition invariant in [24] is not applicable as such. For intuition, an abstract program $P^{\#}$ can be imagined as a new notion of transition invariant, one that encodes justice and compassion assumptions in a graph with labeled edges.

The abstract interpretation framework formalizes the conservative approximation of fixed point expressions [10]. For the verification of liveness properties denoted by fixpoints expressions, this approximation involves the underapproximation of least fixpoints or (equivalently) the overapproximation 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. [4, 12, 26, 28]).

One source of inspiration for the idea of abstracting relations is the work on higher-order abstract interpretation in [11]. Its instantiation to transition predicate abstraction and its use for liveness with justice and compassion is proper to this paper.

The termination analysis of [19] for functional programs is based on the comparison of infinite paths in the control flow graph and in 'size-changing graphs'; that comparison can be reduced to the language containment test of Büchi automata. Our work extends the termination principle in [19] to a setting with parameterized abstraction, liveness, and fairness.

Verification diagrams are graphs that are useful to factorize deductive proofs of temporal properties including liveness [5]. Their nodes denote sets of states (and not pairs of states) and are hence close in spirit to abstract-state programs (and not to the abstract-transition programs). It may be interesting to consider verification diagrams with nodes denoting sets of *pairs* of states, and to come up with according proof rules.

The predicate abstraction method of [9] 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 transfered fairness, which strongly depends on the precision of the abstraction. Our method accounts for fairness requirements without abstracting them.

3. ABSTRACT-TRANSITION PROGRAMS

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. 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-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 $\tau \in \mathcal{T}$.

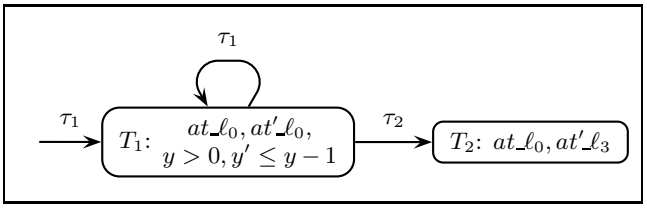

Figure 3: Abstract-transition program LOOP$^{\#}$.

On 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*

$$at\_\ell_0 \wedge at'\_\ell_0 \wedge y > 0 \wedge y' \leq y - 1$$

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_3 (with unspecified values for y), respectively.

The abstract-transition program LOOP$^{\#}$ abstracts the program LOOP. 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 [8, 23, 27].

We now start the formal definitions.

Programs and Computations. Following [20], we abstract away from the syntax of a concrete (concurrent) programming language and represent a program P by a *fair transition system*

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

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,
$$\rho_\tau \subseteq \Sigma \times \Sigma$$
- $\mathcal{J}$: 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 $s_1, s_2, \ldots$, which is either infinite or no more extendible, such that:

- s_1 is a *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 (justice and compassion) in Sections 6 and 7, respectively.

We write example programs using the Simple Programming Language SPL of [20]. The translation from SPL and other (concurrent) programming languages into fair transition system is standard.

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.*

Usually, transition predicates are given by atomic assertions over unprimed and primed program variables. We fix a transition predicate Id for the identity relation, formally

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

From now on, the formal statements refer to a fixed *finite* set of transition predicates $\mathcal{P}$.

The predicates at_ℓ and at'_ℓ are implicitly contained in $\mathcal{P}$, for all program locations ℓ.

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. Formally,*

$$\mathcal{T}_{\mathcal{P}}^{\#} \;=\; \{p_1 \wedge \ldots \wedge p_n \mid n \geq 0 \text{ and } p_1, \ldots, p_n \in \mathcal{P}\}.$$

Alternatively, we may define an abstract transition to be a conjunction in which every transition predicate appears either positively or negated. In this case, abstract transitions can be identified by bit-vectors. The difference is only relevant for implementation issues.

An abstract-transition program uses abstract transitions for its node labels.

DEFINITION 3 (ABSTRACT-TRANSITION PROGRAM $P^{\#}$).
An abstract-transition program $P^{\#}$ *is a finite directed rooted node-labeled edge-labeled graph*

$$P^{\#} \;=\; \langle V, E, v_0, L_V, L_E \rangle$$

where:

- V *and* E *are the set of nodes resp. edges,*
- $v_0 \in V$ *is the root node,*
- $L_V : V \to \mathcal{T}_{\mathcal{P}}^{\#}$ *and* $L_V(v_0) = Id$,
  i.e., *every node* v *is labeled by an abstract transition* $L_V(v)$ *which we also write* T_v,
  the root node is labeled Id,
- $L_E : E \to \mathcal{T}$,
  i.e., *every edge* (u, v) *is labeled by a transition* τ.

We illustrate the definition above on the abstract-transition program LOOP$^{\#}$, shown on Figure 3.

$$\begin{aligned} V &= \{v_0, T_1, T_2\}, \\ E &= \{(v_0, T_1), (T_1, T_1), (T_1, T_2)\}, \\ L_V(v_0) &= Id \\ L_V(T_1) &= at\_\ell_0 \wedge at'\_\ell_0 \wedge y > 0 \wedge y' \leq y - 1 \\ L_V(T_2) &= at\_\ell_0 \wedge at'\_\ell_3 \\ L_E(v_0, T_1) &= L_E(T_1, T_1) = \tau_1 \\ L_E(T_1, T_2) &= \tau_2 \end{aligned}$$

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

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

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 can now define the meaning of abstraction of a program P by an abstract-transition program $P^{\#}$. Later on, we present an algorithm for the transformation of a program P into an abstract-transition program $P^{\#}$.

DEFINITION 4 (ABSTRACTION $P \sqsubseteq P^{\#}$).
An abstract-transition program $P^{\#} = \langle V, E, v_0, L_V, L_E \rangle$ *is an* abstraction *of the program* $P = \langle \Sigma, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ *if for all nodes* v_1 *labeled by, say, the abstract transition* T_1, *and for all transitions* τ *of the program* P,

if T_1 *contains a pair of states* (s, s') *such that* s' *goes to some state* s'' *under the transition* τ, *then*

- *there exists a non-root node* v_2 *that is labeled by an abstract transition* T_2 *containing the pair* (s, s''), *and*
- *there exists an edge from* v_1 *to* v_2 *labeled by* τ.

Formally:
$v_1 \in V$, $L_V(v_1) = T_1$, $(s, s') \in T_1$, $(s', s'') \in \rho_\tau$ *implies the existence of* $v_2 \in V^-$ *and* $(v_1, v_2) \in E$ *such that* $L_E(v_1, v_2) = \tau$ *and, for* $L_V(v_2) = T_2$, $(s, s'') \in T_2$.

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 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 on Figure 4 takes a program P and returns a program $P^{\#}$ abstracting 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.

The set of transition predicates $\mathcal{P}$ defines a unique 'best-abstraction' function α for the abstract domain $\mathcal{T}_{\mathcal{P}}^{\#}$. It maps a binary relation T over states to the smallest abstract transition containing the relation T.

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 primed and unprimed 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.

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.

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

Figure 4: Transition predicate abstraction $P \mapsto P^{\#}$.

We do not further elaborate the first step, the reduction of the verification problem for general temporal properties to the one for fair termination. This step is standard (cf. [29]), analogous one for safety and reachability.

We have just presented the second step, the transition predicate abstraction-based transformation of the program P into a node-labeled 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 [8, 23, 27]. 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, $x > 0 \wedge x' = x - 1$ corresponds to `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. E.g. our prototype implementation of [23] handles over 500 single while loops in a couple of milliseconds.

The only missing link is the fourth step of our overall method: an algorithm on the automaton underlying $P^{\#}$ that marks nodes as 'fair' resp. 'unfair'. Before we give the formal definition of each kind of fairness, justice resp. compassion in Section 6 resp. Section 7, we outline the algorithm.

The first part of the algorithm computes, for each node v, a set $abc(\mathcal{L}_v)$ of transitions, *i.e.* $abc(\mathcal{L}_v) \subseteq \mathcal{T}$. The second part checks a condition on $abc(\mathcal{L}_v)$. That condition is specific to the kind of fairness, namely (1) in Section 6 resp. (2) in Section 7. 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 Theorem 1 in Section 6 resp. Theorem 2 in Section 7, depending on the kind of fairness.

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.

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

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 $\tau_1 \ldots \tau_n$ over transitions in $\mathcal{T}$ lies in the language $\mathcal{L}_v$ for a non-root node v, unless the composition of the corresponding transition relations is empty. Formally,

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

PROOF. By induction over n. □

The set $abc(\mathcal{L}_v)$ 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.

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

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

6. JUSTICE

Justice is a conditional fairness requirement [20]. It is sensitive to the enabledness of transitions. A transition τ is *enabled* on the state s if the set of states $\{s' \mid (s, s') \in \rho_\tau\}$ is not empty. We write $En(\tau)$ for the set of states on which the transition τ is enabled.

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

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 a certain point 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. Formally,*

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

The 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.

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

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 enabledness sets 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. Formally,*

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

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

We define an auxiliary predicate $just(v, \tau^j)$ as follows.

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

Informally, $just(v, \tau^j)$ holds if the transition τ^j is either taken or not continually enabled on some path from the root to the node v. Such transitions contribute to the marking of v as 'fair'.

A node $v \in V^-$ is marked (justly) 'fair' if the predicate $just(v, \tau^j)$ holds for every just transition.

$$fair_{\mathcal{J}}(v) = \forall \tau^j \in \mathcal{J}.\ just(v, \tau^j) \qquad (1)$$

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

Theorem 1 (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, formally*

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

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 $just(v, \tau^j)$ 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 justice requirement.

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. 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 a state lying between the positions l_i and l_{i+1}, formally

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

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

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

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

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

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

$$(k, l) \sim (k', l') \quad \text{if and only if} \quad f(k, l) = f(k', l')$$

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

By Ramsey's theorem [25], 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 $k, l \in K$ where $k < l$ we have $f(k, l) = v$. 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.

We show that each transition $\tau^j \in \mathcal{J}^t$ is contained in the set of transitions $abc(\mathcal{L}_v)$. By the choice of the set L and taking into consideration that the set K is a subset of L, for each $i \geq 1$ there exist positions a and b in L such that $a < b$, $l_a = k_i$, and $l_b = k_{i+1}$. Furthermore, we have

$$\tau^j \in \{\tau_{l_a}, \dots, \tau_{l_b - 1}\}.$$

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

We show that for every $\tau^d \in \mathcal{J}^d$ there exists a transition $\tau \in abc(\mathcal{L}_v)$ such that $En(\tau) \cap En(\tau^d) = \emptyset$. 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 $\tau \in abc(\mathcal{L}_v)$. By Assumption 2, the sets $En(\tau^d)$ and $En(\tau)$ are disjoint. □

We now illustrate an application of Theorem 1 for proving just termination of example programs.

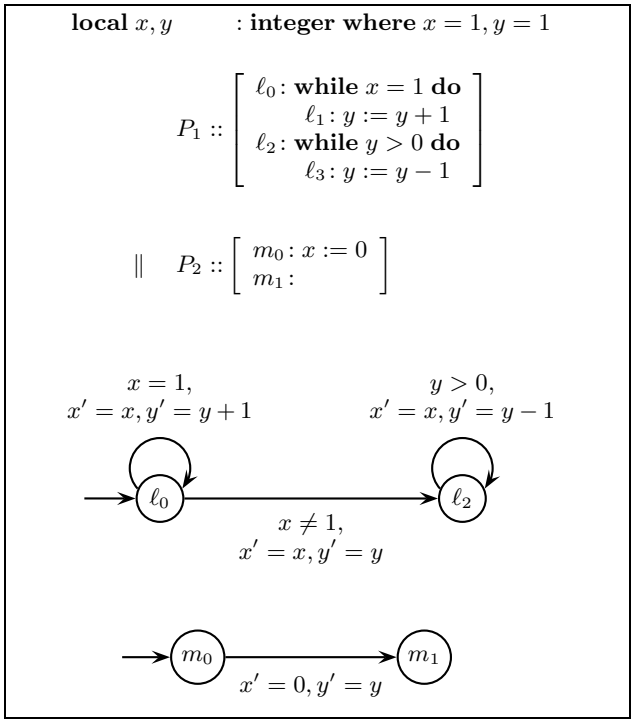

Figure 5: Program ANY-DOWN.

ANY-DOWN. We show the program ANY-DOWN on Figure 5. We obtain the control-flow graph shown on Figure 6 by taking the asynchronous parallel composition of the processes. Every transition is just.

$$\mathcal{J} \;=\; \{\tau_1, \ldots, \tau_4\}$$

We compute the abstract-transition program ANY-DOWN$^{\#}$, shown on Figure 7, by taking the following set of transition predicates.

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

The abstract transition T_1 is the only one that is not well-founded. From the graph of ANY-DOWN$^{\#}$, we obtain the following set $abc(\mathcal{L}_1)$.

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

Since the enabledness condition of the transition τ_1 coincides with the enabledness condition of the transition τ_4, the predicate $just(1, \tau_4)$ does not hold. Hence, the non-well-foundedness of T_1 is not required for the just termination of ANY-DOWN. Since all other abstract transitions are well-founded, by Theorem 1, we conclude the ANY-DOWN justly terminates.

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 on Figures 8 and resp. 9. Every transition is just.

$$\mathcal{J} \;=\; \{\tau_1, \ldots, \tau_6\}$$

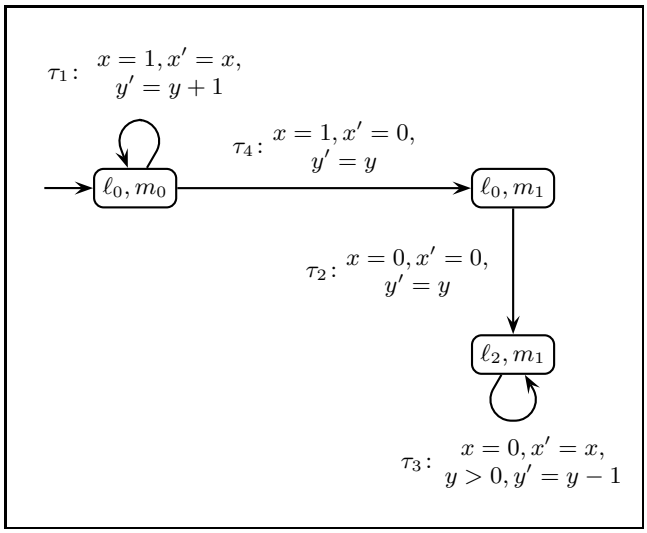

Figure 6: Control-flow graph for the parallel composition of processes P_1 and P_2 in ANY-DOWN.

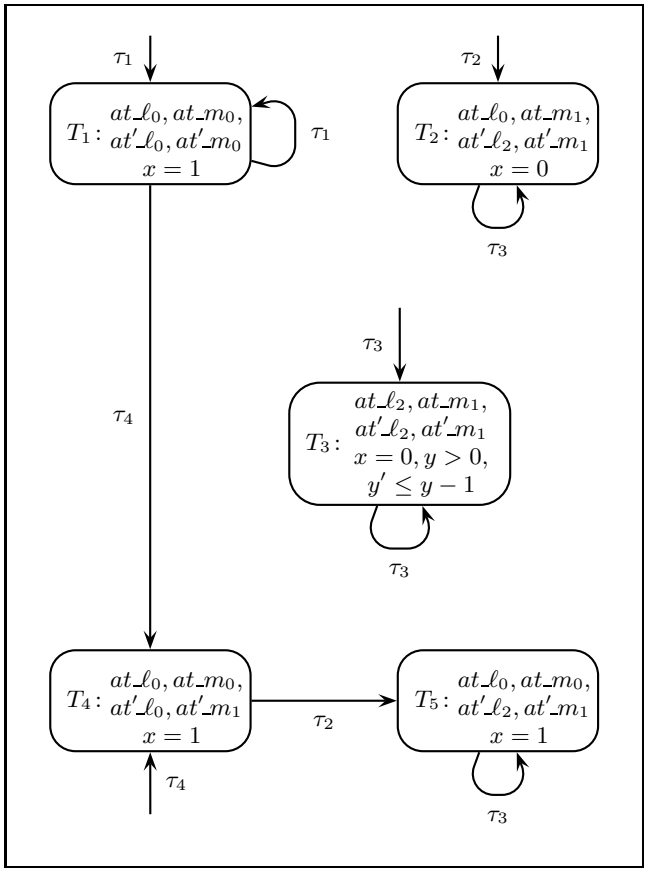

Figure 7: Abstract-transition program ANY-DOWN$^{\#}$.

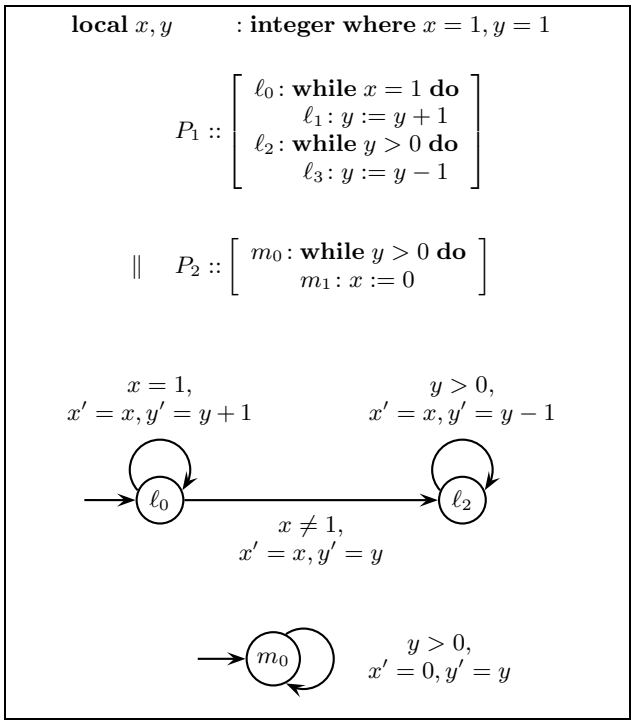

Figure 8: Program ANY-WHILE.

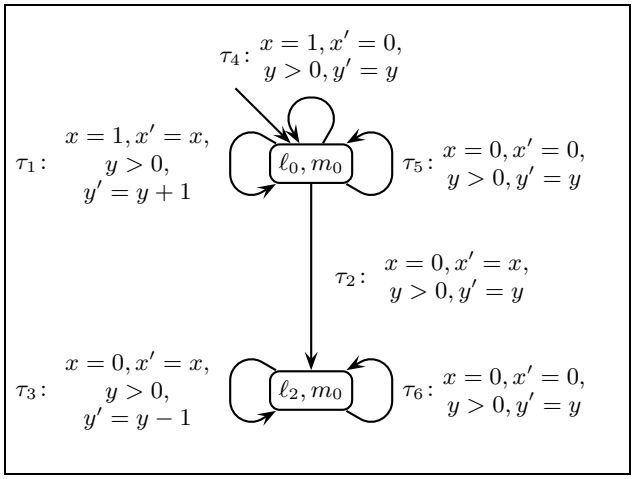

Figure 9: Control-flow graph for the parallel composition of the processes P_1 and P_2 in ANY-WHILE.

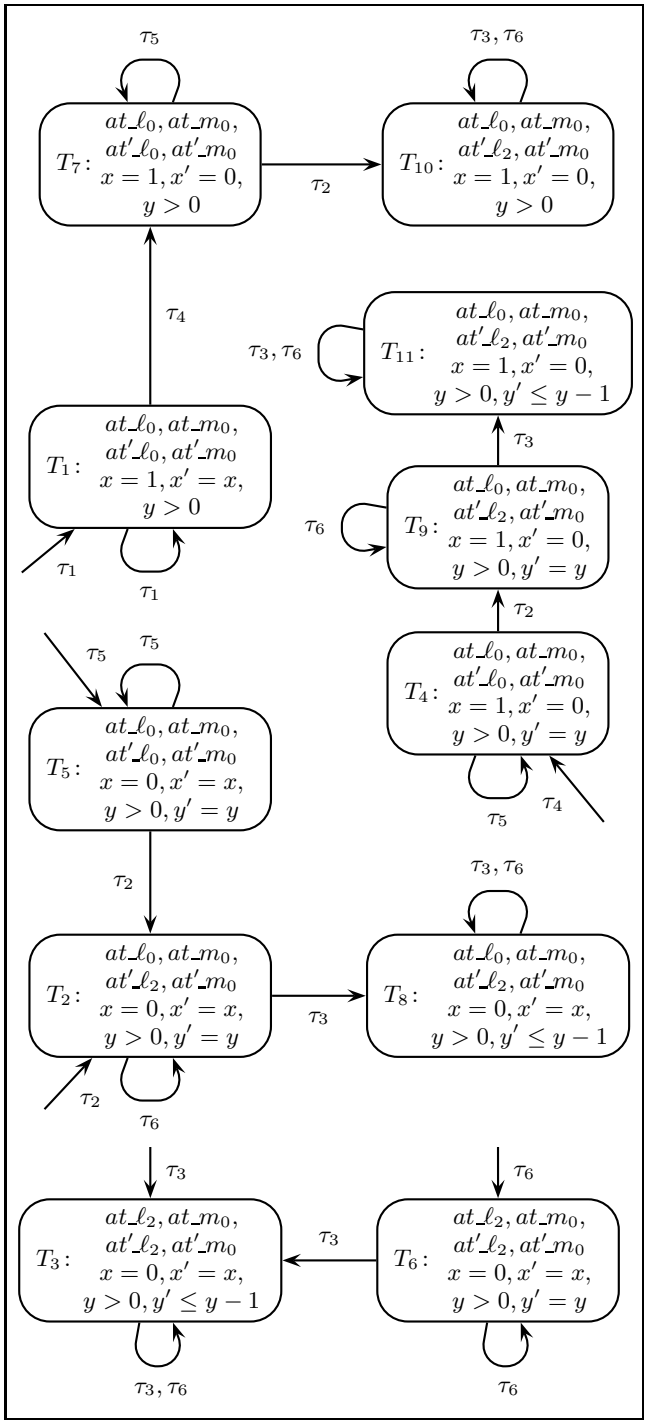

Figure 10: Abstract-transition program ANY-WHILE$^{\#}$.

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 on Figure 10.

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$^{\#}$.

$$\begin{aligned} abc(\mathcal{L}_1) &= \{\tau_1\} \\ abc(\mathcal{L}_5) &= \{\tau_5\} \\ abc(\mathcal{L}_6) &= \{\tau_6\} \end{aligned}$$

Looking at the control-flow graph on Figure 9, we observes the following.

$$\begin{aligned} En(\tau_1) &= En(\tau_4) \\ En(\tau_5) &= En(\tau_2) \\ En(\tau_6) &= En(\tau_3) \end{aligned}$$

This means that the predicates $just(1, \tau_4)$, $just(5, \tau_2)$, and $just(6, \tau_3)$ 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.

7. COMPASSION

Compassion is another conditional fairness requirement [20]. 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.

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) $abc(\bigcap \mathcal{L}_v)$ that appear in every word of the language $\mathcal{L}_v$.

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

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

We define an auxiliary predicate $comp(v, \tau^c)$ as follows.

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

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

A node $v \in V^-$ is marked (compassionately) 'fair' if the predicate $comp(v, \tau^c)$ holds for every compassionate transition.

$$fair_{\mathcal{C}}(v) \;=\; \forall \tau^c \in \mathcal{C}.\; comp(v, \tau^c) \qquad (2)$$

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

THEOREM 2 (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, formally*

$$\forall v \in V^-.\; fair_{\mathcal{C}}(c) \implies \textit{well-founded}(T_v).$$

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 $comp(v, \tau^c)$ holds.

Let $\sigma = s_1, s_2, \ldots$ be an infinite computation induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \ldots$, 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 $\mathcal{C}^{d(isabled)}$ and $\mathcal{C}^{t(aken)}$ as follows. A transition $\tau \in \mathcal{C}$ is in the set $\mathcal{C}^d$ if it is not enabled infinitely often. Otherwise, *i.e.*, if τ is taken infinitely often, we have $\tau \in \mathcal{C}^t$.

Let $L = l_1, l_2, \ldots$ 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
$$\forall \tau \in \mathcal{C}^d\; \forall p \geq l_1.\; s_p \notin En(\tau).$$
- Every transition $\tau \in \mathcal{C}^t$ is taken on a state lying between the positions l_i and l_{i+1} for all $i \geq 1$, formally
$$\forall \tau \in \mathcal{C}^t\; \forall i \geq 1\; \exists\, l_i < p < l_{i+1}.\; \tau_p = \tau.$$

By defining an equivalence relation on pair from the set L and applying Ramsey's theorem along the lines of the proof of Theorem 1, we obtain an infinite ordered set $K \subseteq L$ and a non-root node v with the following property. For every pair of elements (k, l) in K we have $f(k, l) = v$. Again, we observe that the abstract transition T_v is not well-founded. Furthermore, since every transition 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 $abc(\mathcal{L}_v)$.

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 $\tau \in abc(\bigcap \mathcal{L}_v)$ must appear between the positions k_i and k_{i+1}, we conclude that there exists a state s such that $s \in En(\tau)$ and $s \notin En(\tau^d)$. By Assumption 2 (which we extended to the compassionate transitions), the sets $En(\tau^d)$ and $En(\tau)$ are disjoint. □

SUB-SKIP. We illustrate Theorem 2 on the program SUB-SKIP, shown on Figure 11. The set of compassionate transitions $\mathcal{C}$ is the following.

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

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$^{\#}$ on Figure 12. 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 2, the well-foundedness of these two abstract transitions is not needed

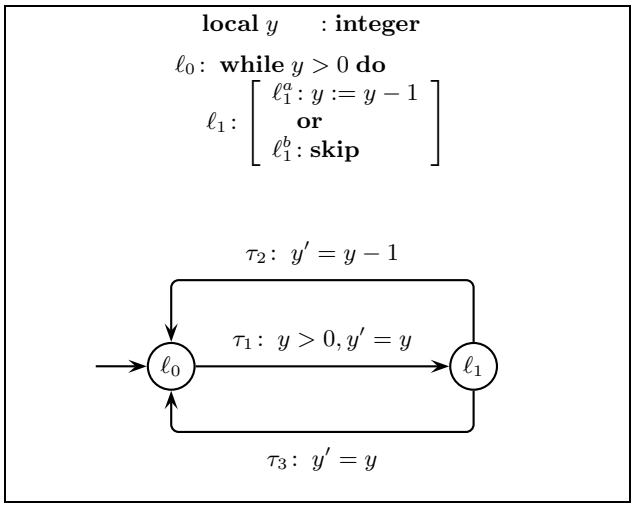

Figure 11: Program SUB-SKIP.

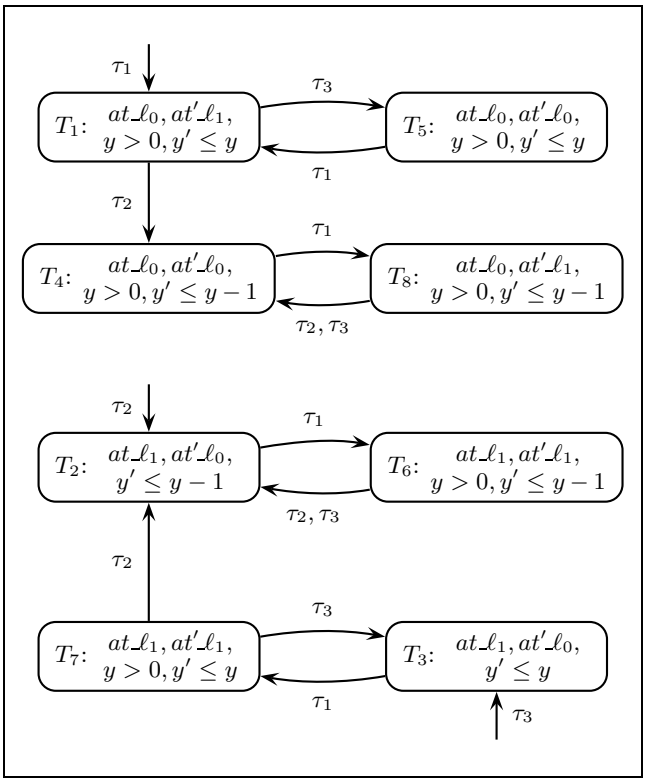

Figure 12: Abstract-transition program SUB-SKIP$^{\#}$.

for proving compassionate termination. We show that the predicates $comp(5, \tau_2)$ and $comp(7, \tau_2)$ do not hold.

From Figure 12, we obtain the following sets of transitions.

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

Furthermore, we observe (on Figure 11) that $En(\tau_2) = En(\tau_3)$. Hence, the predicates $comp(5, \tau_2)$ and $comp(7, \tau_2)$ do not hold.

8. LEXICOGRAPHIC COMPLETENESS

Our main interest is in fair termination. But let us look also 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.* [21]). 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.

The characterization of (plain) termination of a program P (namely, by the well-foundedness of the abstract transitions labeling the nodes of the abstract-transition program $P^{\#}$) is the instance of the characterization of fair termination where the set of fair transitions to be empty. The program P terminates if every non-root node in the abstract-transition program $P^{\#}$ is labeled by well-founded abstract transitions, formally

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

Let $(f_1, \ldots, f_n)$ be a tuple of functions from the set of states Σ into the domains $(\mathcal{W}_1, \succ_1), \ldots, (\mathcal{W}_n, \succ_n)$ such that $\succ_i$ is an ordering relation, *i.e.* transitive and irreflexive, for each $1 \le i \le n$.

The tuple $(f_1, \ldots, f_n)$ 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 $j \in \{1, \ldots, n\}$ such that the auxiliary predicate $lex(\rho_\tau, j)$, defined as follows, holds.

$$lex(R, j) \ = \ \forall (s, s') \in R.\ f_j(s) \succ_j f_j(s') \wedge \forall 1 \le i < j.\ f_i(s) \succeq_i f_i(s')$$

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.

$$f_i \succ_i f_i' \ = \ \{(s, s') \mid f_i(s) \succ_i f_i(s')\}$$
$$f_i \succeq_i f_i' \ = \ \{(s, s') \mid f_i(s) \succeq_i f_i(s')\}$$

Obviously, the transition predicate $f_i \succ_i f_i'$ is well-founded.

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 $x + y \ge x' + y'$.

THEOREM 3 (LEXICOGRAPHIC COMPLETENESS). *If the set $\mathcal{T}_{\mathcal{P}}^{\#}$ 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 $(f_1, \ldots, f_n)$ 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 $(f_1, \ldots, f_n)$ 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 $\mathcal{T}_{\mathcal{P}}^{\#}$ 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 $j \in \{1, \ldots, n\}$ such that the predicate $lex(T_v, j)$ 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 $lex(\rho_\tau, j)$ holds for some $j \in \{1, \ldots, n\}$, we have

$$\rho_\tau \subseteq f_j \succ_j f_j' \ \in \mathcal{T}_{\mathcal{P}}^{\#},$$
$$\forall 1 \leq i < j.\ \rho_\tau \subseteq f_i \succeq_i f_i' \ \in \mathcal{T}_{\mathcal{P}}^{\#}.$$

Since α is the 'best-abstraction' function, we have

$$\alpha(\rho_\tau) \subseteq f_j \succ_j f_j',$$
$$\forall 1 \leq i < j.\ \alpha(\rho_\tau) \subseteq f_i \succeq_i f_i'.$$

Hence, we conclude $lex(T_v, j)$ 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 $lex(T_u, j)$ hold for some index $j \in \{1, \ldots, n\}$. For a transition τ that labels the edge (u, v) there exists an index $l \in \{1, \ldots, n\}$ such that $lex(\rho_\tau, l)$ holds. Let $m = \min(j, l)$. We show that $lex(\alpha(T_v), m)$ holds.

By the induction hypothesis, we have

$$T_u \subseteq f_j \succ_j f_j' \quad \text{and} \quad \forall 1 \leq i < j.\ T_u \subseteq f_i \succeq_i f_i'.$$

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

$$\rho_\tau \subseteq f_l \succ_l f_l' \quad \text{and} \quad \forall 1 \leq k < l.\ \rho_\tau \subseteq f_k \succeq_k f_k'.$$

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

$$T_u \circ \rho_\tau \subseteq f_m \succ_m f_m',$$
$$\forall 1 \leq i < m.\ T_u \circ \rho_\tau \subseteq f_i \succeq_i f_i'.$$

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

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 $\mathcal{T}_{\mathcal{P}}^{\#}$.

CHOICE. We consider the program CHOICE shown on Figure 13. This program terminates. 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.

We compute the abstract-transition program CHOICE$^{\#}$, shown on Figure 14, by taking the following set of transition predicates.

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

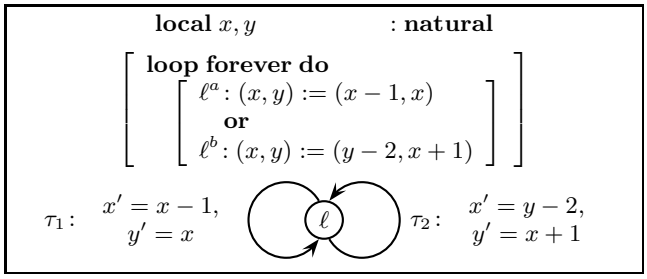

Figure 13: Program CHOICE.

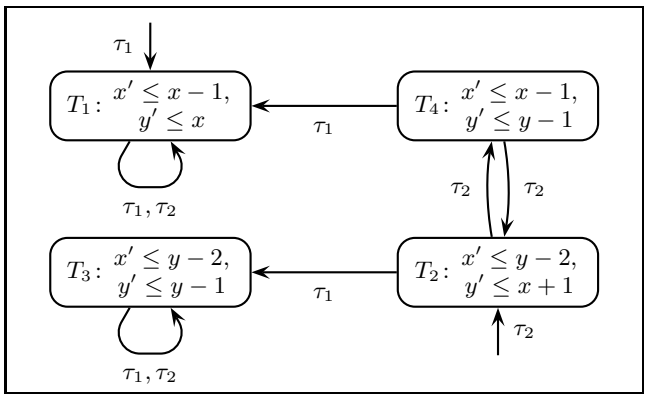

Figure 14: Abstract-transition program CHOICE$^{\#}$.

Note that the set of abstract transition $\mathcal{T}_{\mathcal{P}}^{\#}$ 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.)

We observe that every non-root node in CHOICE$^{\#}$ is labeled by a well-founded abstract transition, *i.e.*, the program CHOICE terminates.

9. 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 (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 lead to the success of tools for safety and invariance properties [2, 6, 14, 15, 30].

The logical next step is to investigate counterexample-driven abstraction refinement [2, 7, 15]. We extracted transition predicates from guards (which yields the special case of assertions such as $x > 0$, *i.e.* without primed variables) and from update statements `x:=e` (which yields transition predicates of the form $x' \leq e$ and $x' \geq e$). Although this was sufficient for our experiments so far, an automated counterexample-driven abstraction refinement will be desirable at some point. A counterexample will here be a relation $\tau_1 \circ \ldots \circ \tau_n$ corresponding to a path in the graph of an abstract-transition program, a path that leads to a 'fair' 'non-terminating' node.

Another direction for future work is to investigate whether the existing techniques to speed up predicate abstraction, *e.g.* [3, 13, 18], 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.

Acknowledgment.
We thank Bruno Blanchet, Bernd Finkbeiner, and Alexander Malkis for comments and suggestions.

10. REFERENCES

[1] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO'2004: Symp. on Formal Methods for Components and Objects*, LNCS. Springer, 2004. To appear.

[2] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'2001: Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[3] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.

[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI'1993: Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.

[5] I. Browne, Z. Manna, and H. Sipma. Generalized verification diagrams. In *FSTTCS'1995: Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer, 1995.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'2003: Int. Conf. on Software Engineering*, pages 385–395, 2003.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'2000: Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[8] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[9] M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV'1998: Computer Aided Verification*, volume 1427 of *LNCS*, pages 293–304. Springer, 1998.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'1977: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[11] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *ICCL'1994: Int. Conf. on Computer Languages*, pages 95–112. IEEE, 1994.

[12] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[13] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL'2002: Principles of Programming Languages*, pages 191–202. ACM Press, 2002.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'1997: Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[15] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'2002: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[16] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation, a special issue on Compositionality*, 163(1):203–243, 2000.

[17] Y. Kesten, A. Pnueli, and M. Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences*, 62(4):668–690, 2001.

[18] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV'2003: Computer Aided Verification*, volume 2725 of *LNCS*, pages 141–153. Springer, 2003.

[19] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'2001: Principles of Programming Languages*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

[20] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[21] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Progress. Draft, 1996.

[22] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.

[23] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In

VMCAI'2004: Verification, Model Checking, and Abstract Interpretation, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[24] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.

[25] F. P. Ramsey. On a problem of formal logic. In *Proc. London Math. Soc.*, volume 30, pages 264–285, 1930.

[26] H. Sipma, T. Uribe, and Z. Manna. Deductive model checking. In *CAV'1996: Computer Aided Verification*, volume 1102 of *LNCS*, pages 208–219. Springer, 1996.

[27] A. Tiwari. Termination of linear programs. In *CAV'2004: Computer Aided Verification*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.

[28] T. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1999.

[29] M. Y. Vardi. Verification of concurrent programs — the automata-theoretic framework. *Annals of Pure and Applied Logic*, 51:79–98, 1991.

[30] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL'2001: Principles of Programming Languages*, pages 27–40. ACM Press, 2001.

[31] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'2003: European Symp. on Programming*, volume 2618 of *LNCS*, pages 204–222. Springer, 2003.

APPENDIX

A. ENABLEDNESS ASSUMPTIONS

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

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 $En(\tau)$ 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 $\{\tau_1, \ldots, \tau_n\}$.

$$En(\tau) \;=\; En(\tau_1) \uplus \cdots \uplus En(\tau_n) \tag{3a}$$

$$\rho_\tau \;=\; \rho_{\tau_1} \uplus \cdots \uplus \rho_{\tau_n} \tag{3b}$$

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

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 Equation* (3).

PROOF. Assume that there exists an infinite computation $\sigma = s_1, s_2, \ldots$ of the original program that satisfies the justice requirement $\mathcal{J}$. Since partitioning does not make the transition relation of the program smaller, see Equation (3b), σ is a computation of the modified program.

We show that for every $\tau \in \mathcal{J}$ replaced by the set of transitions $\{\tau_1, \ldots, \tau_n\}$, 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 $En(\tau_j)$ for some $1 \leq j \leq n$ such that σ eventually does not leave the set $En(\tau_j)$, formally,

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

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. □

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 Equation* (3).

PROOF. Assume that there exists an infinite computation $\sigma = s_1, s_2, \ldots$ of the original program that satisfies the compassion requirement $\mathcal{C}$. Since partitioning does not make the transition relation of the program smaller, see Equation (3b), σ is a computation of the modified program.

We show that for each $\tau \in \mathcal{C}$ replaced by the set of transitions $\{\tau_1, \ldots, \tau_n\}$, 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 $En(\tau_j)$ 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. □