I N F O R M A T I K

Software Model Checking of Liveness Properties via Transition Invariants

Andreas Podelski   Andrey Rybalchenko

MPI–I–2003–2–004   December 2003

FORSCHUNGSBERICHT   RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85   66123 Saarbrücken   Germany

Authors' Addresses

Andreas Podelski
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
rybal@mpi-sb.mpg.de

Abstract

Model checking is an automated method to prove safety and liveness properties for finite systems. Software model checking uses *predicate abstraction* to compute *invariants* and thus prove safety properties for infinite-state programs. We address the limitation of current software model checking methods to safety properties. Our results are a characterization of the validity of a liveness property by the existence of transition invariants, and a method that uses *transition predicate abstraction* to compute *transition invariants* and thus prove liveness properties for infinite-state programs.

Keywords

Transition Invariants, Software Model Checking, Liveness, Predicate Abstraction, Termination

1 Introduction

Software model checking is an approach for extending the applicability of finite-state model checking to software systems with infinite state spaces (see [1, 14, 15, 16, 18, 22, 27, 29, 35]). The extension works via an abstraction step, which is essentially the construction of a finite-state system.[1] The finite-state abstraction step restricts the resulting method to *safety* properties. This is because it does in general not preserve *liveness* properties. For intuition, we take termination as an example of a liveness property; a finite system is terminating only if its execution traces do not contain loops; but then, it can not simulate execution traces of unbounded length (say, of the program `while (i>0) { i:=i-1; }`; see also [19, 24].) This paper addresses the limitation of current software model checking methods to safety properties.

The terminology safety vs. liveness is standard to distinguish two kinds of program properties in the scope of model checking. An example of a safety property is (from the interface specification of an operating system kernel [1]): *each time a lock is acquired, it will get released before the end of the function call.* An example of a liveness property is: *each time a lock is acquired, it will get released.* That is, a liveness property expresses a guarantee, without fixing a time bound. Termination is the standard example of a liveness property; its proof is required in the context of program correctness proofs with interactive theorem provers. Formally, the difference signifies whether the negation of the property can be reduced to reachability (of a 'bad' state) or to the existence of an infinite trace (without a 'good' state). Thus, the difference also signifies whether the property could in principle be checked at runtime, or not.

In this paper, we give a characterization of the validity of a liveness property via the existence of *transition invariants.* This leads to a deductive proof schema, where a given transition invariant is checked for inductiveness (i.e. closure under an operator that we introduce). Roughly, in its restriction to termination, the schema replaces the well-foundedness argument for a ranking relation by a weaker argument for the transition invariant. We show that the schema is suitable for automatization. For this purpose, we introduce *transition predicate abstraction.* This technique generalizes predicate abstraction, the basic abstract interpretation technique of the existing software model checking methods for safety properties. We use transition predicate abstraction as the parameter in a general method to compute tran-

[1] The abstraction step is formalized as the definition of an over-approximating fixed point operator over finitely many (in general infinite) sets of states in [10].

sition invariants, which again we can use to prove the liveness property of the infinite-state program.

As with every automated method for an undecidable problem, the best we can hope for is a semi-test (for safety but not for liveness, a semi-algorithm is another option). That is, if the abstraction is too coarse, the computed transition invariant is not 'strong enough' (in that case one refines the abstraction by adding more transition predicates). We can show, however, that our method is complete wrt. a fixed abstraction. Finally, we determine the complexity of the 'abstract model checking problem for LTL' (in the number of transition predicates); it is PSPACE-complete. I.e., it has the same complexity as in the special case of finite models (when each edge is expressed by one transition predicate).

To explain the approach of this paper, we look at the role that *invariants* play in the proof of a safety property. The safety property is translated to the non-reachability of a 'bad' state from an initial state. Its proof is phrased as the proof of a 'strong enough' invariant (an invariant is a state assertion that holds for every reachable state; 'strong enough' means that it does not hold for any bad state). The deductive proof schema consists of showing the inductiveness of a 'strong enough' invariant (the inductiveness is the closure under the successor operator post). The approach of this paper is to introduce concepts analogous to ['strong enough', inductive] invariants and show that they can be used to characterize the validity of a liveness property.

Following the abstract interpretation framework [10], an inductive invariant is obtained mechanically as the least fixed point of an abstraction of the post operator over a subdomain of the domain of sets of states. The subdomain consists of equivalence classes of states when *predicate abstraction* is used, as in software model checking (equivalent states satisfy the same predicates). Accordingly, the approach of this paper is to introduce the appropriate least fixed point operator and the appropriate domain and the appropriate predicate abstraction and to use these ingredients of the abstract interpretation framework to formulate algorithms computing transition invariants.

2 Examples

This section is informal. The formal exposition starts in the next section.

Termination We use the following simple program to illustrate the use of transition invariants for termination.

```
    int n,i,j,A[n];
    i=n;
l1: while (i>=0) {
      j=0;
l2:   while (j<=i-1) {
        if (A[j]>=A[j+1])
          swap(A[j],A[j+1]);
        j=j+1;
      }
      i=i-1;
    }
```

```
l1: if (i>=0) j=0;

l2: if (i-j>=1) {
      j=j+1;
      goto l2;
    } else {
      i=i-1;
      goto l1;
    }
```

For legibility, we concentrate on the skeleton shown on the right, which consists of the statements `st1`, `st2`, `st3`.

```
l1: if (i>=0)   { i:=i;   j:=0;   goto l2; }   - st1
l2: if (i-j>=1) { i:=i;   j:=j+1; goto l2; }   - st2
l2: if (i-j<1)  { i:=i-1; j:=j;   goto l1; }   - st3
```

Each of the *abstract statements* below must be read as a one-line program.

```
l1: if (true)   {i:=Any;   j:=Any;   goto l2; }  - a1
l2: if (true)   {i:=Any;   j:=Any;   goto l1; }  - a2
l1: if (i>=0)   {i:=i-Pos; j:=Any;   goto l1; }  - a3
l2: if (i>=0)   {i:=i-Pos; j:=Any;   goto l2; }  - a4
l2: if (i-j>=1) {i:=i-Nat; j:=j+Pos; goto l2; }  - a5
```

We notice that `st1` is approximated by `a1`, `st2` by `a5` and `st3` by `a2`. In fact, every sequence of program statements is approximated by one of `a1`, ..., `a5`. This means that the set $\{\texttt{a1}, \ldots, \texttt{a5}\}$ is a transition invariant in our terminology.

For example, every sequence of program statements that leads from `l2` to `l2` is approximated by `a4` if it passes through `l1`, and by `a5` otherwise. The following table assigns to each abstract statement the set of sequences of program statements that it approximates. All non-assigned sequences are not feasible.

a1	$\texttt{st1}(\texttt{st2} \mid \texttt{st3st1})^*$
a2	$(\texttt{st2} \mid \texttt{st3st1})^*\texttt{st3}$
a3	$\texttt{st1}(\texttt{st2} \mid \texttt{st3st1})^*\texttt{st3}$
a4	$(\texttt{st2} \mid \texttt{st3st1})^*\texttt{st3st1}(\texttt{st2} \mid \texttt{st3st1})^*$
a5	$\texttt{st2}^+$

According to our formal development in the following sections (see Theorem 1), the transition invariant above is 'strong enough' to prove termination, which means: each of its abstract statements, viewed in isolation as a one-line program, is terminating.

To prove that a set of abstract statements is indeed a transition invariant, we show that it is inductive or that it can be strengthened by an inductive one. The inductiveness of the transition invariant means that each composition of an abstract statement with a program statement is approximated by the transition invariant. This is in general weaker than requiring that each composition of abstract statements must be approximated by the transition invariant.

The composition of the abstract statement `a1` with the program statement `st3` yields the abstract statement `l1: if (true) { i:=Any; j:=Any; goto l1; }`, which is not approximated by the transition invariant. Thus, the transition invariant is not inductive. We strengthen it by the inductive transition invariant given below.

```
l1: if (i>=0)   { i:=i-Nat; j:=Any;   goto l2; }
l2: if (true)   { i:=i-Pos; j:=Any;   goto l1; }
l1: if (i>=0)   { i:=i-Pos; j:=Any;   goto l1; }
l2: if (i>=0)   { i:=i-Pos; j:=Any;   goto l2; }
l2: if (i-j>=1) { i:=i-Nat; j:=j+Pos; goto l2; }
```

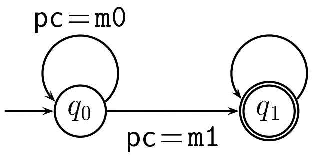
This transition invariant is computed by our method; it corresponds to the output produced by our implementation.

Fairness We use the following simple program "Up-down" to illustrate the use of transition invariants for *fair termination*.

```
 int x=0, y=0;
l0: while (x=0) y++; || m0: x=1;
l1: while (y>0) y--; || m1:
l2:                  ||
```

Termination is the inevitability of the location (`l2, m1`). For formal reason, the program has a self-loop in this location.

Termination can be proven only under the fairness assumption that the process on the right-hand side will eventually move from `m0` to `m1`. This assumption is encoded by the Büchi automaton below.



The transition invariant that we compute for this problem contains 49 abstract statements. Below we give the two critical ones.

```
l0_m0_q0: if (true) { x:=Any;
                      y:=y+Pos; goto l0_m0_q0; }  - a1
l1_m1_q1: if (y>0)  { x:=Any;
                      y:=y-Pos; goto l1_m1_q1; }  - a2
```

The abstract statement `a1` does not terminate. However, by the formal theory that we establish, its termination is not needed because its executions do not visit the Büchi accepting state q_1 infinitely often. This is where the fairness assumption comes in; the loop in `l0_m0` is not a fair execution. The 'fair' abstract statement `a2` terminates.

3 Transition Invariants

Our formal exposition is based on *command formulas.*

Example 1 *The command formula c below represents the statement*

if *y>=0* **then** *x=x+1;*

of a program over variables $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *where* $\mathtt{l}_1$ *and* $\mathtt{l}_2$ *are the labels before and after the statement.*

$$c \stackrel{def}{=} \underbrace{\mathsf{pc}=\mathtt{l}_1 \wedge y \geq 0}_{guard} \wedge \underbrace{x' = x+1 \wedge y' = y \wedge \mathsf{pc}'=\mathtt{l}_2}_{action}$$

In a command formula, the subformula over unprimed variables $x_1, \ldots, x_n$ forms the *guard* (enabling condition). The remaining conjuncts form the *action* (update of the variables). Usually, they are of the form $x' = \mathtt{E}$, where `E` is the update expression over unprimed variables (translating assignments `x := E`).

From now on, we assume that the program is given as a set C of command formulas. The translation from programs to sets of command formulas is standard for many programming languages, including concurrent ones.

A basic observation is that one can use command formulas of a more general form than the one that corresponds to programs in order to denote relations between states of a more general kind than the transition relations denoted by programs. We will next introduce some notation and define how general command formulas ϕ denote relations between states.

The n-tuple $X = (x_1, \ldots, x_n)$ consists of the variables appearing in the program. Usually, one or many program counter variables ("`pc`") appear among the x_i's; they range over the program labels. The free variables of a command formula ϕ are among the variables $x_1, \ldots, x_n$ and their primed versions $x'_1, \ldots, x'_n$.

A state s is a valuation of the program variables $x_1, \ldots, x_n$. The set of all states is denoted by S. The value of the program variable x in a state s is $s(x)$. A pair of program states s and s' satisfies a command formula ϕ, formally $(s, s') \vDash \phi$, if ϕ evaluates to true after interpreting x_i by $s(x_i)$ and x'_i by $s'(x_i)$ for all i. The transition relation denoted by the command formula ϕ is the set of all state pairs that satisfy ϕ.

$$\rightarrow_\phi \stackrel{\text{def}}{=} \{(s, s') \mid (s, s') \vDash \phi\}$$

Given the program in the form of the command formula ϕ, the state s' is reachable from the state s in one execution step if $s \rightarrow_\phi s'$ (which means that the pair (s, s') satisfies ϕ), and reachable in a non-empty sequence of execution steps if $s {\rightarrow_\phi}^+ s'$. As usual, $\rightarrow^+$ denotes the transitive (but not reflexive) closure of the relation $\rightarrow$.

A *transition formula* Φ is a set of command formulas. An example of a transition formula is the program C. We use the terms *disjunction* and *set* of command formulas interchangeably. The terminology and notation above for command formulas extend canonically to sets.

Definition 1 Transition Invariant.
A transition invariant of a program C is a transition formula Ψ that holds of every pair of states s and s' such that s' is reachable from s in a non-empty sequence of execution steps.

That is, the transition relation of a transition invariant Ψ contains the transitive closure of the transition relation of the program C, formally ${\rightarrow_C}^+ \subseteq \rightarrow_\Psi$.

Invariants We assume that the given program comes with a state formula Init denoting the set of initial states. A *state formula* or *state assertion* is a formula whose free variables are the program variables (including the program counter pc); it denotes a set of states. An *invariant* Inv is a state assertion that holds for every reachable state (reachable from an initial state).

We construct the formula Inv_Ψ from a transition invariant Ψ as the disjunction of Init with the formula that denotes the set of direct successor states of initial states of C under statements in Ψ (here $[X/X']$ refers to the renaming of the primed by the unprimed version of each program variable).

$$\mathsf{Inv}_\Psi \stackrel{\text{def}}{=} \mathsf{Init} \vee (\exists X\ (\mathsf{Init} \wedge \Psi))[X/X']$$

Remark 1 *Given the transition invariant Ψ and the state formula Init denoting the set of initial states, the formula Inv_Ψ is an invariant of the program.*

Conversely, given an invariant Inv of the program C, the transition formula $\mathsf{Inv} \wedge \mathsf{Inv}[X'/X]$ is a transition invariant not for the program C itself but for the program $\mathsf{Inv} \wedge C$ obtained by strengthening the guards with information about reachable states. As usual, we extend conjunction to sets of formulas in the canonical way, i.e., $\Phi_1 \wedge \Phi_2 \overset{\text{def}}{=} \{\phi_1 \wedge \phi_2 \mid \phi_1 \in \Phi_1 \text{ and } \phi_1 \in \Phi_2\}$.

Well-founded Command Formulas The command formula ϕ is *well-founded* if the transition relation $\to_\phi$ (strictly speaking, its inverse) is well-founded, i.e., there is no infinite sequence of states $\{s_i\}_{i=1}^{\infty}$ such that each consecutive pair of states satisfies the command formula, formally $(s_i, s_{i+1}) \models \phi$ for all i. In terms of program executions this means that the one-line program represented by the command formula is terminating whatever its initial states are (i.e., if its initial condition is true).

Notation of Meta-Variables We use ϕ for general command formulas, ψ for those in transition invariants, c for those in the program, and their uppercase version to sets thereof, i.e., Φ for general transition formulas, Ψ for transition invariants and C for the program.

4 Termination

The program is *terminating* if every execution starting in an initial state is finite. This is a special case of an LTL property; technically this section is subsumed by the next one. We single out termination because of its singular importance. Its treatment is possible without introducing Büchi automata.

Theorem 1 (Transition Invariants and Termination) *The program represented by the set of command formulas C is terminating if there exists a finite transition invariant Ψ such that all command formulas in $\mathsf{Inv}_\Psi \wedge \Psi$ are well-founded.*

Proof. Assume, for a proof by contraposition, that Ψ is a finite transition invariant for C, and that C is not terminating. We show that at least one command formula in $\mathsf{Inv}_\Psi \wedge \Psi$ is not well-founded.

By the assumption that C is not terminating, there exists an infinite sequence of states $\pi \overset{\text{def}}{=} \{s_i\}_{i=1}^{\infty}$ such that s_1 is an initial state and $s_i \to_{c_i} s_{i+1}$ for all i, where $c_i \in C$.

We define a function f that maps an ordered pair of indices of elements in the sequence π to one of the command formulas in the transition invariant Ψ as follows.

$$f(k,l) \stackrel{\text{def}}{=} \psi \in \Psi, \quad \text{where } (s_k, s_l) \models \psi$$

The function f exists because Ψ is a transition invariant for C, and thus we can choose arbitrarily one command formula from the (finite) set $\{\psi \in \Psi \mid (s_k, s_l) \models \psi\}$ as the image of the pair (k,l). The range of the function f is finite since Ψ is finite.

$$\begin{tikzcd} \cdots \arrow[r] & s_k \arrow[r] \arrow[rr, bend right, "\psi"'] & \cdots \arrow[r] & s_l \arrow[r] & \dots \end{tikzcd}$$

Given π, the function f induces an equivalence relation $\sim$ on pairs of positive integers (in this proof we always consider pairs whose first element is smaller than the second one).

$$(k,l) \sim (k',l') \stackrel{\text{def}}{=} f(k,l) = f(k',l')$$

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

By Ramsey's theorem [28], there exists an infinite set of positive integers K such that all pairs of elements in K belong to the same equivalence class, say $[(m,n)]_\sim$ with $m, n \in K$. That is, for all $k, l \in K$ such that $k < l$ we have $(k,l) \sim (m,n)$. We fix m and n.

Let $\{k_i\}_{i=1}^{\infty}$ be the ascending sequence of elements of K. Let the command formula ψ denote the command formula $f(m,n)$. Since $(k_i, k_{i+1}) \sim (m,n)$, the function f maps each pair (k_i, k_{i+1}) to ψ.

$$\begin{tikzcd} \cdots \arrow[r] & s_{k_i} \arrow[r] \arrow[rr, bend right, "\psi"'] \arrow[rrrr, bend left, "\psi"] & \cdots \arrow[r] & s_{k_j} \arrow[r] \arrow[rr, bend right, "\psi"'] & \cdots \arrow[r] & s_{k_l} \arrow[r] & \dots \end{tikzcd}$$

Hence, the infinite sequence $\{s_{k_i}\}_{i=1}^{\infty}$ is induced by ψ.

$$s_{k_i} \to_\psi s_{k_{i+1}}, \quad \text{for all } i \geq 1$$

Since we assume that s_1 is an initial state, every state s_{k_i} satisfies the invariant Inv_Ψ, and hence the infinite sequence is induced also by $\mathsf{Inv}_\Psi \wedge \psi$. Hence, the command formula $\mathsf{Inv}_\Psi \wedge \psi$ is not well-founded. Therefore at least one command formula in $\mathsf{Inv}_\Psi \wedge \Psi$ is not well-founded. $\square$ The statement of the theorem still holds when one replaces the invariant Inv_Ψ by any other finite invariant.

The sufficient condition for program termination given in Theorem 1 has three components: the 'transitive closure' property of transition invariants, the finiteness and the 'disjunctive well-foundedness'. The examples below show that the first resp. the second component can not be omitted.[2]

Example 2 *The transition formula* $\Psi = \{c_1, c_2\}$ *given by the two command formulas* $c_1 \equiv x > 0 \wedge x' < x$ *and* $c_2 \equiv y > 0 \wedge y' < y$ *is finite and 'disjunctively well-founded'. The transition relation induced by* Ψ *is not terminating, as can be seen from the infinite sequence* $(0,1), (1,0), (0,1), \ldots$ *of states.*

Example 3 *The program given by the single command formula* c *below (translating "*`while (x>=0) x++;`*") does not terminate for initial states where* $x \geq 0$*.*

$$c \stackrel{\textit{def}}{=} x \geq 0 \wedge x' = x + 1$$

The transition formula Ψ *below denotes a transition invariant for* $\{c\}$ *that consists of infinitely many well-founded command formulas (here* $\mathsf{Inv}_\Psi \wedge \Psi$ *is equivalent to* Ψ*).*

$$\Psi \stackrel{\textit{def}}{=} \{x = k \wedge x' > x \mid k \in \mathbb{N}\}$$

The strongest transition invariant (which denotes the transitive closure of the transition relation of the program) can in general not be used for the sufficient condition of termination (since it is infinite in general).

---

[2] To compare the standard approach to termination proof and our approach (in its restriction to termination), we view a *ranking function* (defined by the expression $e[X]$ in the program variables) as a (transitive) ranking relation (the transition formula $e[X'] < e[X]$ in primed and unprimed program variables). The ranking relation approximates the transition relation of the program (and also its transitive closure). Termination follows from the well-foundedness of the ranking relation. By definition, a transition invariant approximates the transitive closure of the transition relation of the program (in general, a transition invariant is not transitive, even when it is inductive as defined in Definition 3). Termination then follows already from the 'disjunctive well-foundedness' of the transition invariant (by an argument that exploits the combinatorial property of Ramsey's theorem). Disjunctive well-foundedness is weaker than well-foundedness (take the disjunctively well-founded relation given by $x' < x \vee y' < y$). Theorem 1 states a condition under which disjunctive well-foundedness does imply well-foundedness. Namely, stated in terms of relations r and R instead of transition formulas:
If R can be decomposed into a union of well-founded relations ($R = R_1 \cup \ldots \cup R_n$, and $R_1, \ldots, R_n$ are well-founded) and $r^+ \subseteq R$, then r (as well as r^+) is well-founded.

5 LTL

We follow the automata-theoretic approach to verification [34] (see [33] for the generalization to infinite-state systems). We assume that the given LTL (Linear Temporal Logic [25]) property φ is represented by a Büchi automaton $A_{\neg\varphi}$ for its negation (more precisely, an LTL formula φ over the finite set of atomic propositions AP is represented by a Büchi automaton $A_{\neg\varphi}$ that accepts exactly the infinite sequences of program states that do not satisfy φ). We thus need not introduce the syntax of LTL. We use Büchi automata and their synchronous parallel composition with programs in the usual way, with the only difference that atomic propositions denote infinite sets of program states (i.e., an atomic proposition is a formula in the program variables X).

The Büchi automaton $A_{\neg\varphi} = (Q, \Sigma, \Delta, q_{init}, Acc)$ consists of the finite set of states Q, the finite alphabet $\Sigma = 2^{AP}$, the transition relation $\Delta \subseteq Q \times \Sigma \times Q$, the initial state $q_{init} \in Q$, and the set of accepting states $Acc \subseteq Q$.

A run of $A_{\neg\varphi}$ on the infinite sequence $\sigma_1, \sigma_2, \dots$ is an infinite sequence of states $q_1, q_2, \dots$ such that q_1 is q_{init} and $(q_i, \sigma_i, q_{i+1}) \in \Delta$ for all $i \geq 1$. The run $q_1, q_2, \dots$ is accepting if an accepting state $q_{acc} \in Acc$ appears infinitely often. An infinite sequence $w = \sigma_1, \sigma_2, \dots$ is accepted by $A_{\neg\varphi}$ if there exists an infinite run on w.

Let $L : S \to 2^{AP}$ be a labelling function on program states that provides the set of all atomic propositions satisfied by the given program state. An infinite sequence of program states $s_1, s_2, \dots$ satisfies $\neg\varphi$ if and only if the infinite sequence of state labels $L(s_1), L(s_2), \dots$ is accepted by $A_{\neg\varphi}$.

The program C satisfies the LTL property φ if there exists no infinite sequence of program states that is a program trace of C *and* satisfies $\neg\varphi$, i.e., that is a *accepting run* of the *synchronous parallel composition* of C and $A_{\neg\varphi}$, to be introduced next.

We introduce a new program variable, the program counter pc_A ranging over the set of automaton states Q. A state of the product program is a valuation over the tuple of program variables X and the variable pc_A; we write it as a pair (s, q) of states of C and $A_{\neg\varphi}$, respectively.

Definition 2 Synchronous Parallel Composition $C \times A_{\neg\varphi}$.
The synchronous parallel composition of the program C and the Büchi automaton $A_{\neg\varphi}$ (with transition relation Δ) is the transition formula

$$C \times A_{\neg\varphi} \stackrel{def}{=} \{c \wedge c_{(q,\sigma,q')} \mid c \in C \text{ and } (q, \sigma, q') \in \Delta\},$$

where

$$c_{(q,\sigma,q')} \stackrel{def}{=} \mathsf{pc}_A = q \wedge \mathsf{pc}'_A = q' \wedge \bigwedge_{p\in\sigma} p \wedge \bigwedge_{p\notin\sigma} \neg p.$$

A run of $C \times A_{\neg\varphi}$ is an infinite sequence of state pairs $(s_1, q_1), (s_2, q_2), \ldots$ that starts in initial states of C resp. $A_{\neg\varphi}$, and such that each consecutive pair of state pairs satisfies the transition formula $C \times A_{\neg\varphi}$. The run is *accepting* if an accepting state of $A_{\neg\varphi}$ appears infinitely often in the infinite sequence $q_1, q_2, \ldots$.

Remark 2 *The run $s_1, s_2, \ldots$ of the program C does not satisfy the LTL property φ if and only if $(s_1, q_1), (s_2, q_2), \ldots$ is an accepting run of $C \times A_{\neg\varphi}$.*

In the statement below, we can use any other invariant instead of Inv_Ψ (which is the invariant obtained from a given transition invariant; see Remark 1).

Theorem 2 (Transition Invariants and LTL) *The program C satisfies the LTL property φ if there exists a finite transition invariant Ψ for $C \times A_{\neg\varphi}$ such that each command formula of the form*

$$\mathsf{Inv}_\Psi \wedge \mathsf{pc}_A = q_{acc} \wedge \psi$$

is well-founded, where ψ is a command formula in Ψ and q_{acc} is an accepting state of $A_{\neg\varphi}$.

Proof. For a proof by contraposition, assume that Ψ is a finite transition invariant for $C \times A_{\neg\varphi}$, and that the program C does not satisfy the LTL property φ. By Remark 2, there exists an accepting run $(s_1, q_1), (s_2, q_2), \ldots$ of $C \times A_{\neg\varphi}$ (starting in initial states of C resp. $A_{\neg\varphi}$) and an accepting state q_{acc} that appears in the sequence $q_1, q_2, \ldots$ infinitely often, say at each index in the infinite set of indices P.

$$P \stackrel{\text{def}}{=} \{i \mid q_i \equiv q_{acc}\}$$

Since Ψ is a transition invariant, we can define a function f from the set of ordered pairs of indices in P to the set of command formulas in Ψ as below.

$$f(k, l) \stackrel{\text{def}}{=} \psi \in \Psi, \quad \text{where } ((s_k, q_k), (s_l, q_l)) \models \psi$$

By Ramsey's theorem [28], there exists a command formula ψ in Ψ and an infinite set of indices K that is a subset of P, such that each pair of consecutive indices in K is mapped to ψ, formally $f(k_i, k_{i+1}) = \psi$ where k_i and k_{i+1} in K for $i \geq 1$.

In summary, each consecutive pair of state pairs $((s_{k_i}, q_{k_i}), (s_{k_{i+1}}, q_{k_{i+1}}))$ satisfies the command formula ψ, each state pair satisfies the guard $\mathsf{pc}_A = q_{acc}$, and, since the first element (s_1, q_1) of the infinite sequence is a pair of initial states (of C resp. $A_{\neg\varphi}$), every subsequent element and in particular every element of the form (s_{k_i}, q_{k_i}) satisfies the invariant Inv_Ψ of $C \times A_{\neg\varphi}$. This means, the infinite sequence $((s_{k_1}, q_{k_1}), ((s_{k_2}, q_{k_2}) \ldots$ is induced by the command formula $\mathsf{Inv}_\Psi \wedge \mathsf{pc}_A = q_{acc} \wedge \psi$. Which is, this command formula is not well-founded. $\square$

6 Synthesis of Inductive Transition Invariants via Transition Predicate Abstraction

We will develop a verification method for LTL properties via the automated synthesis of inductive transition invariants by iteration of the best abstraction of a fixed point operator over an abstract domain defined by predicates. This is akin to the automated synthesis of inductive invariants in the software model checking method for safety properties; the difference is that the fixed point operator is based on sequential composition of commands and the predicates range over transitions instead of states. The program C to which we refer in this section is either the program whose termination we want to check, or (more generally) its product with a Büchi automaton for the negation of the LTL formula that we want to check.

Inductive Transition Invariants From now on, we use a given logical consequence ordering $\models$ over transition formulas. If the transition relation of one transition formula Φ_2 is a consequence of another one Φ_1, then its transition relation contains the other one (formally, if $\Phi_1 \models \Phi_2$ then $\rightarrow_{\Phi_1} \subseteq \rightarrow_{\Phi_2}$). We do not require the converse. This means that $\models$ is a sound but not necessarily complete implementation of the validity of implication (a sound and complete implementation may be too inefficient or not even exist).

We define a *composition* operator $\circ$ on command formulas that corresponds to the composition of transition relations, i.e., $\rightarrow_{\phi_1 \circ \phi_2} = \rightarrow_{\phi_1} \circ \rightarrow_{\phi_2}$.

$$\phi_1 \circ \phi_2 \stackrel{\text{def}}{=} \exists X'' \, (\phi_1[X''/X'] \wedge \phi_2[X''/X])$$

Here we replace each primed variable in ϕ_1 and each unprimed variable in ϕ_2 by the corresponding double-primed one. The composition of transition formulas $\Phi_1 \circ \Phi_2$ is the set of command formulas obtained by pairwise composition of those in Φ_1 and Φ_2.

$$\Phi_1 \circ \Phi_2 = \{\phi_1 \circ \phi_2 \mid \phi_1 \in \Phi_1 \text{ and } \phi_2 \in \Phi_2\} \tag{1}$$

The transition formula Φ is a transition invariant for the program C if the formula Φ is a consequence of the composition of every non-empty sequence of command formulas in C (the converse is not necessarily true since the consequence relation $\models$ is not necessarily complete).

$$c_1 \circ \ldots \circ c_n \models \Phi, \quad \text{for all } n \geq 1 \text{ and } c_1, \ldots, c_n \in C$$

Using the Kleene star operator for the iterative composition, we can write the above sufficient condition for transition invariants as $C \circ C^{\star} \models \Phi$.

Definition 3 Induction.
The transition formula Φ is inductive *for the program C if Φ is a consequence of C and of Φ composed with C.*

$$C \vee \Phi \circ C \quad \models \quad \Phi$$

The condition in Definition 3 is weaker (not stronger) than $\Phi \circ \Phi \models \Phi$, i.e., an inductive transition formula is not necessarily closed under composition. By the next statement, we have a sufficient condition for transition invariants that it is effectively testable whenever $\models$ is.

Remark 3 *A transition formula is a transition invariant if it is inductive.*

Deductive Proof Schema By Remark 3 and Theorem 1, in order to prove that the program C is terminating it is sufficient to provide a finite set of well-founded command formulas and show that is inductive for C.

Similarly, we derive an inductive proof scheme for LTL properties from Remark 3 and and Theorem 2.

The deductive proof schema is complete in the sense of deductive completeness investigated e.g. in [19]. This is because the well-founded ranking relation induced by a ranking function is a transition invariant which consists of one well-founded command formula (see Footnote 2).

As with the proofs of safety properties, it is in general not possible to find 'strong enough' transition invariants automatically; thus, we design systematic methods that find transition invariants 'in the best possible way', in the sense made precise in abstract interpretation [10].

Transition Predicates We will obtain inductive transition invariants as least fixed points of abstractions of the *concrete* fixed point operator F that we define as follows.

$$F(\Phi) \stackrel{\text{def}}{=} \Phi \circ C$$

We can define the backward version by

$$B(\Phi) \stackrel{\text{def}}{=} \{g \wedge \phi[e/X] \mid \phi \in \Phi \text{ and } c \in C, \\ \text{where } c \text{ is } g \wedge X' = e\},$$

which does not require elimination of existentially quantified variables. Everything in the following also holds analogously for B. The least fixed points of F and B are equivalent; this is no longer the case when we abstract the two fixed point operators. That is, in the application of our algorithms to programs, the result returned may depend on the choice of the fixed point operator, F vs. B.

The (in general infinite) *concrete* domain D contains all transition formulas (i.e., all sets of command formulas in the given formalism); its partial ordering is the given consequence relation $\models$. The least fixed point of the operator F over the (upwards-complete) domain D always exists (it is possibly an infinite set). It is the strongest (inductive) transition invariant of the program.

Following [10], we may define the *abstract* domain $D^\#$ to be a finite subset of D such that $D^\#$ is a *Moore family*, i.e., $D^\#$ contains the supremum of D and is closed under conjunction. As a consequence (Theorem 5.1.0.3 in [10]), every formula in D has a minimal consequence in $D^\#$. The abstract domain $D^\#$ determines the *best abstraction* function from D into $D^\#$ by

$$\alpha(\Phi) \quad \stackrel{\text{def}}{=} \quad \bigwedge\{\Psi \in D^\# \mid \Phi \models \Psi\}. \tag{2}$$

In the case of *predicate abstraction*, the abstract domain $D^\#$ consists of the finite formulas built up from a given finite subset of D of 'atomic' formulas. The atomic formulas define a finite number of *transition predicates*. In contrast to predicates that are defined by formulas over unprimed program variables and apply to states, the (more general) transition predicates are defined by formulas over primed and unprimed variables and apply to transitions (i.e., pairs of states).

For example, given the transition predicates $\mathsf{pc} = \mathtt{l_i}$, $\mathsf{pc}' = \mathtt{l_i}$ (for each label $\mathtt{l_i}$), $\mathsf{z} \geq \mathsf{k}$, and $\mathsf{z}' \leq \mathsf{z} + \mathsf{k}$ for $\mathsf{z} \in \{x, y\}$ and $\mathsf{k} \in \{-1, 0, 1\}$, the abstraction of the command formula c in Example 1 is

$$\alpha(c) \quad \equiv \quad \mathsf{pc} = \mathtt{l_1} \wedge y \geq 0 \wedge x' \leq x + 1 \wedge y' \leq y \wedge \mathsf{pc}' = \mathtt{l_2}.$$

Given the abstract domain $D^\#$, the best abstraction of the fixed point operator F is the operator $F^\#$ defined below (for transition formulas Ψ in

$D^{\#}$).[3]

$$F^{\#}(\Psi) \stackrel{\text{def}}{=} \alpha(\Psi \circ C). \tag{3}$$

The monotonicity of the fixed point operator $F^{\#}$ is a direct consequence of the monotonicity of the composition and the abstraction function. By Tarski's fixed point theorem, the least fixed point of $F^{\#}$ exists. We denote the least fixed point of $F^{\#}$ above C by $\mathsf{lfp}(F^{\#}, C)$. The fixed point $\mathsf{lfp}(F^{\#}, C)$ is computed in the usual fashion.

$$\begin{array}{rcl}
\mathsf{lfp}(F^{\#}, C) & = & \Psi_1 \vee \ldots \vee \Psi_n, \text{ where} \\
\Psi_1 & = & \{\alpha(c) \mid c \in C\} \\
\Psi_{i+1} & = & \{\alpha(\psi \circ c) \mid \psi \in \Psi_i \text{ and } c \in C\} \\
\Psi_{n+1} & \models & \Psi_1 \vee \ldots \vee \Psi_n
\end{array}$$

Here, we implicitly apply (1) and the additivity of the abstraction function α. Since $D^{\#}$ is finite, the fixed point computation terminates after finitely many iterations.

Strengthening Transition Invariants Given an invariant Inv, we can obtain a stronger transition invariant Ψ by using (an abstraction of) the fixed point operator F for the program $\{\mathsf{Inv} \wedge c \mid c \in C\}$ where each guard is strengthened by the invariant. The stronger transition invariant Ψ can in turn be used to construct a stronger invariant Inv_Ψ (see Remark 1).

LTL Software Model Checking The algorithm defined in Figure 1 is a semi-test for the validity of an LTL property for a program. We call it the 'LTL software model checking algorithm' because it uses the same main ingredient of the already cited algorithms known under software model checking for safety, namely fixed point iteration over an abstract domain defined by predicates. The correctness of a semi-test is the soundness of its definite answers.

Theorem 3 (Soundness) *The LTL software model checking algorithm is correct, i.e., if the algorithm returns "LTL Property Holds" then the input LTL property φ holds for the input program C.*

[3] The associativity of the composition operator $\circ$ is not preserved under predicate abstraction. I.e., the *abstract composition* $\circ^{\#}$ defined by $\psi_1 \circ^{\#} \psi_2 = \alpha(\psi_1 \circ \psi_2)$ is not associative. In other words, the abstractions of the command formulas do not generate a monoid. This is because we use the best possible abstractions. Similarly, the operator $F^{\#\#}$ defined by $F^{\#\#}(\Psi) = \Psi \circ^{\#} \alpha(C)$ is not the best abstraction of F.

input

program C with initial states Init

transition predicates defining abstract domain $D^\#$

Büchi automaton $A_{\neg\varphi}$ with initial state q_{init} and accepting states Acc

begin

$C := C \times A_{\neg\varphi}$

$\mathsf{Init} := \mathsf{Init} \wedge \mathsf{pc}_A = q_{init}$

$\alpha := \lambda\Phi.\ \bigwedge\{\Psi \in D^\# \mid \Phi \models \Psi\}$

$F^\# := \lambda\Psi.\ \alpha(\Psi \circ C)$

$\Psi := \mathsf{lfp}(F^\#, C)$

$\mathsf{Inv}_\Psi := \mathsf{Init} \vee (\exists X\ (\mathsf{Init} \wedge \Psi))[X/X']$

if foreach ψ in Ψ and $q_{acc} \in Acc$

well-founded$(\mathsf{Inv}_\Psi \wedge \mathsf{pc}_A = q_{acc} \wedge \psi)$

then

return("LTL Property Holds")

else

return("Don't Know")

end.

Figure 1: LTL software model checking.
The program is given as a set C of command formulas; its set of initial states is denoted by the state formula Init; the abstract domain $D^\#$ with glb operator $\bigwedge$ is given through a finite set of transition predicates ($D^\#$ consists of all Boolean combinations of those predicates); the operator $\circ$ over transition formulas is sequential composition; the Büchi automaton $A_{\neg\varphi}$ represents the negation of the LTL property φ to be checked; the operator $\times$ is the synchronous parallel composition of a program and the Büchi automaton.

```
input
      program C with initial states Init,
      transition predicates defining abstract domain D#
begin
      α := λΦ. ⋀{Ψ ∈ D# | Φ ⊨ Ψ}
      F# := λΨ. α(Ψ ∘ C)
      Ψ := lfp(F#, C)
      Inv_Ψ := Init ∨ (∃X (Init ∧ Ψ))[X/X']
      if foreach ψ in Ψ
            well-founded(Inv_Ψ ∧ ψ)
      then
            return("Terminating")
      else
            return("Don't Know")
end.
```

Figure 2: Termination algorithm (special case of LTL algorithm)

Proof. The correctness follows from the fact that a fixed point of an abstraction of an operator is also a fixed point of that operator [10], the fact that a fixed point of the composition operator F is an inductive transition formula, Remark 3 and Theorem 2. □

The algorithm in Figure 2 is a semi-test for termination. It is the special case of the one in Figure 1 where the Büchi automaton consists of one state (which is both initial and accepting, and which has a transition to itself). Its correctness can be shown using Theorem 1 instead of Theorem 2.

The LTL algorithm (or its restriction to termination) cannot be complete (result a definite answer always if the program is terminating) for decidability reasons. Instead, we have a different kind of completeness.

Theorem 4 (Abstraction Completeness) *If the abstract domain $D^{\#}$ contains a finite inductive transition invariant for the program $C \times A_{\neg\varphi}$ that consists of well-founded command formulas, then the LTL software model checking algorithm will succeed in proving the LTL property, i.e., it will return a definite answer ("LTL Property Holds").*

Proof. [Sketch] We use the characteristic property of best abstractions in the same way as for the completeness of abstraction-based proofs for safety properties (see [2, 9]). □

In this section, we have constructed one particular abstraction, based on transition predicates. Many more constructions are possible; see [10, 9]. The correctness of the algorithm will hold for any sound abstraction $F^{\#}$ of the concrete composition operator F.

Well-foundedness of Command Formulas The algorithms are parametrized by a test of (a sufficient condition for) the well-foundedness of the command formulas ψ in the abstract domains $D^{\#}$. This test returns the value of the expression "well-founded(ψ)".

In the implementation of the algorithm that we used for our examples, the test is based on linear programming. It is a decision procedure for a class of command formulas in linear arithmetic [26].

If the repetition $\psi \circ \psi \circ \ldots \circ \psi$ of any length is different from false, then the command formula ψ translates a program that consists of one `while` loop. We call it a simple `while` loop because its body consists of straight-line code (without if-then-else branching). Given a class of programs we want to verify, and given the corresponding abstract domain, the next step is to determine good sufficient termination conditions for the corresponding class of simple `while` loops.

Complexity We will next show that, in a complexity-theoretic sense, the semi-test that we have given is not optimal for the problem that it 'solves'. First we formally define what problem that is. We introduce the problem to decide, given a program and an abstraction (a set of transition predicates), whether 'the abstract program' satisfies the LTL property. Formally, whether the program has an inductive transition invariant, with well-founded command formulas, in the abstract domain defined by the transition predicates. (This is in analogy with the setting of Boolean programs with predicate abstraction in [1]: a Boolean program satisfies the given safety property if and only if the program has an inductive invariant, without 'bad' states, in the abstract domain defined by the predicates. We omit a formalization of 'Boolean transition programs'; their operational semantics would be based on sequential composition.)

We give an optimal algorithm and a lower bound and thus determine the complexity of the decision problem in the size of the set of transition predicates (which depend on the given program; the program is either the one whose termination we want to check, or more generally its product with a Büchi automaton for the negation of the LTL formula that we want to check; the transition predicates include usually the formulas $\mathsf{pc} = \mathtt{l_n}$ and $\mathsf{pc}' = \mathtt{l_n}$ and $\mathsf{pc}_A = q$ and $\mathsf{pc}'_A = q$ for program labels $\mathtt{l}_n$ of the program and states q

of the Büchi automaton).

Theorem 5 (Complexity of Abstract LTL Model Checking) *Given a fixed program and a set of transition predicates, the problem to decide whether the program has an inductive transition invariant with well-founded command formulas in the abstract domain defined by the transition predicates, is PSPACE-complete in the number of transition predicates.*

Proof. [Sketch] For the upper bound, we use not fixed point iteration but a non-deterministic algorithm that explores $\mathsf{lfp}(F^\#, C)$; again, as in the proof of Theorem 4, we use the characteristic property of best abstractions in the same way as for the completeness of abstraction-based proofs for safety properties (see [2, 9]). The lower bound is obvious from the finite-state case (the only values are the labels of concurrent programs; the strongest transition invariant is inductive and finite, hence the well-foundedness of its command formulas is a sufficient and necessary condition for program correctness). □

7 Related Work

Our use of Ramsey's theorem in the proofs of Theorems 1 and 2 is reminiscent of its use in Büchi's theory of ω-regular languages over a finite alphabet (see [32]). This theory is the basis of the automata-theoretic approach to LTL model checking [34]. The equivalence classes over segments correspond to the transformer functions of the Büchi automaton. We, however, restrict ourselves to finite sets of transformers over an infinite state space, as opposed to restricting oneself to transformers over a finite state space (in both cases, the sets of transformers are finite and thus induce an equivalence relation of finite index, which is the raison d'être of the finiteness restriction). As a consequence, we infer the existence of an *ultimately periodic* sequence of transformers ($\psi_0 \circ \psi^\omega$ or, in the notation of [32], $[v]_\approx \cdot [w]_\approx{}^\omega$), as opposed to an ultimately periodic sequence of states. For a more subtle difference, in our setting (see Footnote 3), the mapping of each finite segment of an infinite trace to its equivalence class is not a monoid homomorphism.

In [23], Lee, Jones and Ben-Amram present a termination analysis for functional programs; the analysis 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 inclusion test for Büchi automata. Our work takes what we think is the essence in [23], formulates it in a logic-based setting combined with abstract interpretation, connects it with predicate abstraction and explores how far one can get, pushing for greater generality. We use a different setting (imperative and concurrent instead of functional programs). That taken aside, a bold way to state the contribution of our work in comparison to theirs is that we go from termination to general LTL properties, and that we go from one fixed abstraction to generic abstract domains (specified by transition predicates). The specification of an abstract domain allows one to fine-tune an abstraction to the property one tries to prove; in a way, this possibility turns a program analysis into a verification method.

One can successively refine abstraction by adding more transition predicates. For example, adding $x' < x$ and then $x' < x - 1$ allows one to check the termination of the program `while(x>=0){x:=x+1; x:=x-2}`.

One practical advantage of our logic-based setup over the graph-based one of [23] is the possibility to take into account the specification of the initial states of a program (without it, one checks a—too strong—condition for termination, namely under the—too weak—assumption that every state can be an initial state); we do so in Theorems 1 and 2 by adding an invariant as a conjunct to each command formula in the transition invariant (e.g. the invariant Inv_Ψ derived from the transition invariant Ψ).

In [7, 8], Colon and Sipma give methods to prove termination that are highly successful in practice. The methods are based on linear arithmetic, as opposed to being parametrized by theorem provers. They work by isolating strongly connected components in the control flow graph of a program and by computing *ranking functions* for each one of them. No composition of program statements is considered, i.e., the program `while (x>=0) {x=x+1; x=x-2;}` cannot be proven terminating.

In [33], Vardi provides an automata-theoretic framework for verification of concurrent systems by appying infinite automata. The system satisfies is correct if its parallel composition with the automaton that accepts computations violating the property does not have infinite computations. Our LTL checking setup follows this framework.

In [19], Kesten, Pnueli and Vardi 'augment' finite-state abstractions with progress monitors to verify liveness properties. They were perhaps the first to point out finite-state abstractions as such are not sufficient for automated liveness proofs. The progress monitors incarnate ranking functions that have to be provided manually. Ranking functions are to liveness what invariants are to safety; in automated methods for safety, invariants are synthesized.

Previous approaches to automated liveness checking for infinite-state systems are based on the iteration of fixpoint operators over sets of states. When they account for fairness assumptions (as e.g. [31]), they need to *under*-approximate sets of states, which seems hard. Without fairness assumptions, liveness proofs for concurrent systems are in general flawed (for example, the non-starvation proofs for integer-valued communication protocols in [5, 11] are based on models where no processor is allowed to idle).

We are not aware of previous formalizations of transition invariants. The notion has appeared implicitely in special instances. For example, the meta-transitions of Boigelot and Wolper e.g. in [3] are (in fact, the strongest) transition invariants for simple loops. They may be useful for showing the well-foundedness of command formulas in transition invariants, as the work in [4] already indicates. The `modifies` clauses used by Rustan Leino and Kuncak e.g. in [20]; the clause `modifies x` expresses the the conjunction of $y' = y$ for all other variable y different from x is a transition invariant.

8 Conclusion, Future Work

Verifying liveness properties of software will always be a hard problem; any automated method will work at most in examples where no ingeniousness or creativity is required. We have presented a formal framework that may serve as a starting point for designing automated methods in a certain style. Namely, it suggets using least fixed point iteration in combination with abstract interpretation techniques and in particular transition predicate abstraction. I.e., this style relies on the same basis as the methods known under software model checking, which have demonstrated their strong potential in automated tools for verifying safety properties of software systems.

It is not at all obvious, however, whether the success of the software model checking method will extend from safety to liveness. We cannot predict whether one can achieve scalability. We still have to develop techniques for constructing abstract domains that are as rich for transition formulas as for state formulas. These techniques need to balance the cost for computing the abstraction with the cost for the test of well-foundedness (see also below).

We have done some preliminary practical experiments with a prototypical implementation in Sicstus Prolog [21] with a constraint solver for linear arithmetic (clpqr [17]); we use the CMU BDD package for the efficient implementation of the fixpoint test. We used our tool to prove non-starvation for the 2-process version of the bakery protocol. Non-starvation expresses that each time a process requests a resource it will finally access it. Its proof requires fairness assumptions (each process must make progress, at certain program locations). We express the fairness assumptions in the LTL formula "if fair then non-starvation" and use the tool LTL2BA [12] to translate its negation into a Büchi automaton. We then compute transition invariants for the 'synchronous parallel composition' of the protocol and the Büchi automaton, in the 'forward' (F) and the 'backward' (B) way. We give some measurements in Table 1. The experiments indicate that the size of the

Program	S	P	time (ms)	command formulas		
				total	loop	fair
Bakery(B)	218	0	1740	815	17	8
Bakery(F)	218	0	4480	815	17	8

Table 1: Transition invariants in numbers.
B and F stands for the backward and forward fixed point operators; S is the number of guarded commands in the composition of the protocol with the Büchi automaton for the negation of the LTL property "if fair then non-starvation"; P denotes the number of transition predicates in the abstract domain (excluding the ones for the program counters); the three numbers total, loops and acc-loops refer to the total number of command formulas in the transition invariants, the ones that correspond to loops in the control flow graph (the others are trivially terminating) and the ones that start from Büchi accepting locations (i.e. correspond to fair execution paths), respectively.

transition invariants gets too large when one encodes fairness assumptions by Büchi automata. We work on more direct ways to encode relevant specific kinds of fairness such as *weak fairness.*

Predicate abstraction is a way to delegate the creativity part of the synthesis of invariants to the choice of a set of predicates and defer the tediousness to the computation over an exponentially large abstract domain. The method becomes fully automated (and then the semi-test loses its termination guarantee) if combined with an automated procedure for counter-example guided abstraction refinement [6, 1, 16, 2]. In our experiments, we have generated the transition predicates using such a procedure (which we did not describe since it is quite analogous to the one for state predicates in [2]). Future work includes the investigation of efficient specialized procedures and their fundamental properties as in [13, 2].

The interest of transition invariants goes beyond liveness checking. It may be relevant for *interprocedural* safety checking, since the synthesis of transition invariants generalizes the *functional approach* to program analysis of Sharir/Pnueli [30].

Acknowledgements This work started with discussions with Chin Soon Lee and Neil Jones during their visit in Saarbrücken in September 2002. We thank Patrick Cousot, Kedar Namjoshi and Amir Pnueli for their remarks on ranking functions and finite-state abstraction during VMCAI in January 2003. Amir Pnueli contributed the insights formulated in Footnote 2. We

thank Bernd Finkbeiner for insightful comments and suggestions.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[2] T. Ball, A. Podelski, and S. K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In J.-P. Katoen and P. Stevens, editors, *Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172. Springer-Verlag, 2002.

[3] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In D. L. Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.

[4] A. Bouajjani, A. Collomb-Annichini, Y. Lakhnech, and M. Sighireanu. Analyzing fair parametric extended automata. In P. Cousot, editor, *SAS'01: Static Analysis Symposium*, volume 2126 of *LNCS*, pages 335–355. Springer, 2001.

[5] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite-state systems using Presburger arithmetics. In O. Grumberg, editor, *Proceedings of the Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411. Springer-Verlag, 1997.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[7] M. Colon and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *LNCS*, pages 67–81. Springer-Verlag, 2001.

[8] M. Colon and H. Sipma. Practical methods for proving program termination. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Ver-*

ification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings, volume 2404, pages 442–454. Springer, 2002.

[9] P. Cousot. Partial completeness of abstract fixpoint checking. In B. Y. Choueiry and T. Walsh, editors, *Abstraction, Reformulation, and Approximation, 4th International Symposium, SARA 2000, Horseshoe Bay, Texas, USA, July 26-29, 2000, Proceedings*, volume 1864 of *LNCS*, pages 1–15. Springer, 2000.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[11] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings of TACAS'99: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Springer LNCS*, pages 223–239. Springer-Verlag, 1999.

[12] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 53–65. Springer Verlag, 2001.

[13] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM (JACM)*, 47(2):361–416, 2000.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV'97: Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[15] J. Hatcliff and M. B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *LNCS*, pages 39–58. Springer, 2001.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *LNCS*, pages 526–538. Springer, 2002.

[17] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3.* Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[18] G. J. Holzmann. Software analysis and model checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *LNCS*, pages 1–16. Springer, 2002.

[19] 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.

[20] V. Kuncak and R. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[21] T. I. S. Laboratory. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.

[22] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *LNCS*, pages 98–112. Springer-Verlag, 2001.

[23] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Norris and J. James B. Fenwick, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

[24] K. S. Namjoshi. Lifting temporal proofs through abstractions. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, model checking, and abstract interpretation : 4th International Conference, VMCAI 2003 ; New York, NY, USA, January 9 - 11, 2003*, volume 2575 of *Lecture notes in computer science*, pages 174–188. Springer, 2003.

[25] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE Computer Society Press, 1977.

[26] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Proc. of VMCAI 2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer-Verlag, 2003.

[27] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 83–94. ACM Press, 2002.

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

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[30] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall Software Series, pages 189–233. Prentice-Hall, Englewood Cliffs , NJ , USA, 1981.

[31] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In *Proc. of CAV 1996*, volume 1102 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[32] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 133–192. Elsevier and MIT Press, 1990.

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

[34] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[35] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In H. R. Nielsen, editor, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–40. ACM Press, 2001.

INFORMATIK

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nrnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nrnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004		Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018		A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem
MPI-I-2003-1-017	G. Schfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems”
MPI-I-2003-1-015	A. Kovcs	Sum-Multicoloring on Paths

MPI-I-2003-1-014	G. Schfer, L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, T. Vredeveld	Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm
MPI-I-2003-1-013	I. Katriel, S. Thiel	Fast Bound Consistency for the Global Cardinality Constraint
MPI-I-2003-1-012		- not published -
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Polynomial Time Algorithms for Network Information Flow
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions
MPI-I-2003-1-004	E. Althaus, T. Polzin, S.V. Daneshmand	Improving Linear Programming Approaches for the Steiner Tree Problem
MPI-I-2003-1-003	R. Beier, B. Vcking	Random Knapsack in Expected Polynomial Time
MPI-I-2003-1-002	P. Krysta, P. Sanders, B. Vcking	Scheduling and Traffic Allocation for Tasks with Bounded Splittability
MPI-I-2003-1-001	P. Sanders, R. Dementiev	Asynchronous Parallel Disk Sorting
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Trff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schfer	Exp Lab A Tool Set for Computational Experiments
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Trff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics

MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches