Ranking Abstraction of Recursive Programs*

Ittai Balaban[1], Ariel Cohen[1], and Amir Pnueli[1,2]

[1] Dept. of Computer Science, New York University
[2] Dept. of Computer Science, Weizmann Institute of Science,
Rehovot 76100, Israel

Abstract. We present a method for model-checking of safety and liveness properties over procedural programs, by combining state and ranking abstractions with procedure summarization. Our abstraction is an augmented finitary abstraction [KP00, BPZ05], meaning that a concrete procedural program is first augmented with a well founded ranking function, and then abstracted by a finitary state abstraction. This results in a procedural abstract program with strong fairness requirements which is then reduced to a finite-state *fair discrete system* (FDS) using procedure summarization. This FDS is then model checked for the property.

1 Introduction

Procedural programs with unbounded recursion present a challenge to symbolic model-checkers since they ostensibly require the checker to model an unbounded call stack. In this paper we propose the integration of ranking abstraction [KP00, BPZ05], finitary state abstraction, procedure summarization [SP81], and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. The inputs to this method are a sequential procedural program together with state and ranking abstractions. The output is either "success", or a counterexample in the form of an abstract error trace. The method is sound, as well as complete, in the sense that for any valid property, a sufficiently accurate *joint* (state and ranking) abstraction exists that establishes its validity.

The method centers around a fixpoint computation of procedure summaries of a finite-state program, followed by a subsequent construction of a behaviorally equivalent nondeterministic procedure-free program. Since we begin with an infinite-state program that cannot be summarized automatically, a number of steps involved in abstraction and LTL model-checking need to be performed over the procedural (unsummarized) program. These include augmentation with non-constraining observers and fairness constraints required for LTL verification and ranking abstraction, as well as computation of state abstraction. Augmentation with global observers and fairness is modeled in such a way as to make the associated properties observable once procedures are summarized. In computing the abstraction, the abstraction of a procedure call is handled by abstracting "everything but" the call itself, i.e., local assignments and binding of actual parameters to formals and of return values to variables.

* This research was supported in part by NSF grant CCR-0205571, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 267–281, 2006.

The method relies on machinery for computing abstraction of first order formulas, but is orthogonal as to how abstraction is actually computed. We have implemented a prototype based on the TLV symbolic model-checker [Sha00] by extending it with a model of procedural programs. Specifically, given a symbolic finite-state model of a program, summaries are computed using BDD techniques in order to derive a *fair discrete system* (FDS) free of procedures to which model-checking is applied. The tool is provided, as input, with a concrete program and predicates and ranking components. It computes *predicate abstraction* [GS97] automatically using the method proposed in [BPZ05]. We have used this implementation to verify a number of canonical examples, such as Ackerman's function, the Factorial function and a procedural formulation of the 91 function.

While most components of the proposed method have been studied before, our approach is novel in that it reduces the verification problem to that of symbolic model-checking. Furthermore, it allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is supported directly by the model and related algorithms, rather than it being specified in a property.

1.1 Related Work

Recent work by Podelski et al. [PSW05] generalizes the concept of summaries to capture effects of computations between arbitrary program points. This is used to formulate a proof rule for total correctness of recursive programs with nested loops, in which a program summary is the auxiliary proof construct (analogous to an inductive invariant in an invariance proof rule). The rule and accompanying formulation of summaries represent a framework in which abstract interpretation techniques and methods for ranking function synthesis can be applied. In this manner both [PSW05] and our work aim at similar objectives. The main difference from our work is that, while we strive to work with abstraction of predicates, and use relations (and their abstraction) only for the treatment of procedures, the general approach of [PSW05] is based on the abstraction of relations even for the procedure-less case. A further difference is that, unlike our work, [PSW05] does not provide an explicit algorithm for the verification of aribtrary LTL properties. Instead it relies on a general reduction from proofs of termination to LTL verification.

Recursive State Machines (RSMs) [AEY01, ABE$^+$05] and Extended RSMs [ACEM05] enhance the power of finite state machines by allowing for the recursive invocation of state machines. They are used to model the control flow of programs containing recursive procedure calls, and to analyze reachability and cycle detection. They are, however, limited to programs with finite data. On the other hand, the method that we present in this paper can be used to verify recursive programs with infinite data domains by making use of ranking and finitary state abstractions.

In [BR00], an approach similar to ours for computing summary relations for procedures is implemented in the symbolic model checker Bebop. However, while Bebop is able to determine whether a specific program statement is reachable, it cannot prove termination of a recursive boolean program or of any other liveness property.

The paper is organized as follows: In Section 2 we present the formal model of (procedure-free) fair discrete systems, and model-checking of LTL properties over them.

Section 3 formalizes recursive procedural programs presented as flow-graphs. In Section 4 we present a method for verifying the termination of procedural programs using ranking abstraction, state abstraction, summarization, construction of a procedure-free FDS, and finally, model-checking. In Section 5 we present a method for LTL model-checking of recursive procedural programs. Finally, Section 6 concludes and discusses future work.

2 Background

2.1 Fair Discrete Systems

The computation model, *fair discrete systems* (FDS) $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, consists of the following components:

- V : A finite set of *variables*. We define a *state* s to be an interpretation of the variables in V. Denoted by Σ is the set of all states of V.
- Θ : The *initial condition*. It is an assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$.
- $\mathcal{J}$: A set of *justice (weak fairness)* requirements (assertions).
- $\mathcal{C}$: A set of *compassion (strong fairness)* requirements (assertions). Each compassion requirement is a pair $\langle p, q\rangle$ of state assertions.

A *run* of an FDS is a sequence of states $\sigma : s_0, s_1, ...$, satisfying the following:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For every $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

A *computation* of an FDS is an infinite run which also satisfies:

- *Justice:* For every $J \in \mathcal{J}$, σ contains infinitely many states satisfying J.
- *Compassion:* For every $\langle p, q\rangle \in \mathcal{C}$, σ should include either only finitely many p-states, or infinitely many q-states.

An FDS $\mathcal{D}$ is said to be *feasible* if it has at least one computation.

A *synchronous parallel composition* of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \mid\mid\mid \mathcal{D}_2$, is specified by the FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, where

$$\begin{array}{lll} V = V_1 \cup V_2, & \rho = \rho_1 \wedge \rho_2, & \Theta = \Theta_1 \wedge \Theta_2, \\ \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2, & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 & \end{array}$$

Synchronous parallel composition is used for the construction of an observer system $\mathcal{O}$, which evaluates the behavior of another system $\mathcal{D}$. That is, running $\mathcal{D} \mid\mid\mid \mathcal{O}$ will allow $\mathcal{D}$ to behave as usual while $\mathcal{O}$ evaluates it.

2.2 Linear Temporal Logic – LTL

LTL is an extension of propositional logic with two additional basic temporal operators, $\bigcirc$ (Next) and $\mathcal{U}$ (Until), from which $\Diamond$ (Eventually), $\Box$ (Always), and $\mathcal{W}$ (Waitingfor) can be derived. An LTL formula is a combination of assertions using the boolean operators $\neg$ and $\wedge$ and the temporal operators:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\,\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

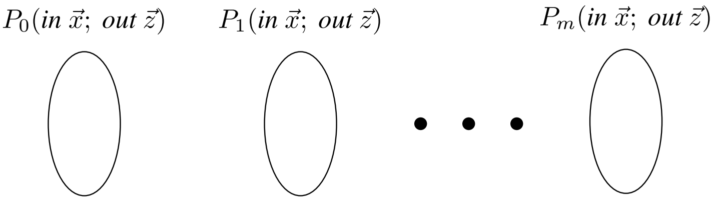
An LTL formula φ is *satisfied* by computation σ, denoted $\sigma \models \varphi$, if φ holds at the initial state of σ. An LTL formula φ is $\mathcal{D}$*-valid*, denoted $\mathcal{D} \models \varphi$, if all the computations of an FDS $\mathcal{D}$ satisfy φ.

Every LTL formula φ is associated with a *temporal tester*, an FDS denoted by $T[\varphi]$. A tester contains a distinguished boolean variable x such that for every computation σ of $T[\varphi]$, for every position $j \geq 0$, $x[s_j] = 1 \iff (\sigma, j) \models \varphi$. This construction is used for model-checking an FDS $\mathcal{D}$ in the following manner:

- Construct a temporal tester $T[\neg\varphi]$ which is initialized with $x = 1$, i.e. an FDS that comprises just those computations that falsify φ.
- Form the synchronous parallel composition $\mathcal{D} \mid\mid\mid T[\neg\varphi]$, i.e. an FDS for which all of its computations are of $\mathcal{D}$ and which violate φ.
- Check feasibility of $\mathcal{D} \mid\mid\mid T[\neg\varphi]$. $\mathcal{D} \models \varphi$ if and only if $\mathcal{D} \mid\mid\mid T[\neg\varphi]$ is infeasible.

3 Recursive Programs

A program P consists of $m+1$ modules: $P_0, P_1, \ldots, P_m$, where P_0 is the main module, and $P_1, \ldots, P_m$ are procedures that may be called from P_0 or from other procedures.


$P_0(in\ \vec{x};\ out\ \vec{z})$
$P_1(in\ \vec{x};\ out\ \vec{z})$
$P_m(in\ \vec{x};\ out\ \vec{z})$

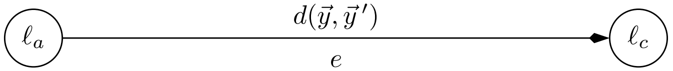
Each module P_i is presented as a flow-graph with its own set of locations $\mathcal{L}_i = \{\ell^i_0, \ell^i_1, \ldots, \ell^i_t\}$. It must have ℓ^i_0 as its only entry point, ℓ^i_t as its only exit, and every other location must be on a path from ℓ^i_0 to ℓ^i_t. It is required that the entry node has no incoming edges and that the terminal node has no outgoing edges.

The variables of each module P_i are partitioned into $\vec{y} = (\vec{x}; \vec{u}; \vec{z})$. We refer to $\vec{x}, \vec{u}$, and $\vec{z}$ as the input, working (local), and output variables, respectively. A module cannot modify its own input variables.

3.1 Edge Labels

Each edge in the graph is labeled by an instruction that has one of the following forms:

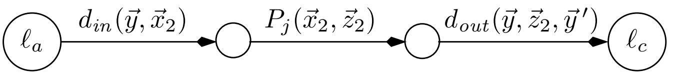
- A *local change* $d(\vec{y}, \vec{y}\,')$, where d is an assertion over two copies of the module variables.



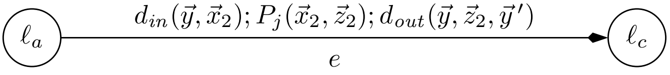
(1)

It is required that $d(\vec{y}, \vec{y}\,')$ implies $\vec{x}\,' = \vec{x}$.

- A *procedure call* $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$, where $\vec{x}_2$ and $\vec{z}_2$ are fresh copies of the input and output parameters $\vec{x}$ and $\vec{z}$, respectively.


ℓ_a
$d_{in}(\vec{y}, \vec{x}_2)$
$P_j(\vec{x}_2, \vec{z}_2)$
$d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$
ℓ_c

This instruction represents a procedure call to procedure P_j where several elements are non-deterministic. The assertion $d_{in}(\vec{y}, \vec{x}_2)$ determines the actual arguments that are fed in the variables of $\vec{x}_2$. It may also contain an enabling condition under which this transition is possible. The assertion $d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ updates the module variables $\vec{y}$ based on the values returned by the procedure P_j via the output parameters $\vec{z}_2$. It is required that $d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ implies $\vec{x}\,' = \vec{x}$. Unless otherwise stated, we shall use the following description as abbreviation for a procedure call.


ℓ_a
$d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$
e
ℓ_c

(2)

Example 1 (The 91 Function). Consider the functional program specified by

$$F(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } F(F(x + 11)) \tag{3}$$

We refer to this function as F_{91}. Fig. 1 shows the procedural version of F_{91}. In the figure, as well as subsequent examples, the notation $\vec{v}_1 := f(\vec{v}_2)$ denotes $\vec{v}_1' = f(\vec{v}_2) \wedge$ *pres*$(\vec{y} - \vec{v}_2)$, with *pres*$(\vec{v})$ defined as $\vec{v}' = \vec{v}$, for some set of variables $\vec{v}$. ▟

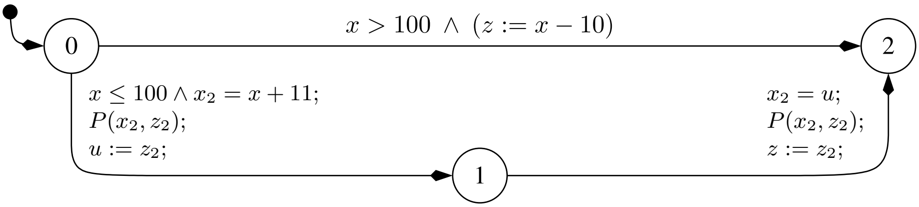

0
$x > 100 \;\wedge\; (z := x - 10)$
2
$x \leq 100 \wedge x_2 = x + 11;$
$P(x_2, z_2);$
$u := z_2;$
1
$x_2 = u;$
$P(x_2, z_2);$
$z := z_2;$

Fig. 1. Procedural program F_{91}

3.2 Computations

A *computation* of a program P is a maximal (possibly infinite) sequence of states and their labeled transitions:

$$\sigma : \langle \ell_0^0; (\xi, \vec{\bot}, \vec{\bot}) \rangle \xrightarrow{\lambda_1} \langle \ell^1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_2} \langle \ell^2; \vec{\nu}_2 \rangle \cdots$$

where each $\vec{\nu}_i = (\xi_i, \eta_i, \zeta_i)$ is an interpretation of the variables $(\vec{x}, \vec{u}, \vec{z})$. The values $\vec{\bot}$ denote uninitialized values. Labels in the transitions are either names of edges in the program or the special label *return*. Each transition $\langle \ell; \vec{\nu} \rangle \xrightarrow{\lambda} \langle \ell'; \vec{\nu}' \rangle$ in a computation must be justified by one of the following cases:

Assignment: There exists an assignment edge e of the form presented in Diagram (1), such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_c$ and $\langle \vec{\nu}, \vec{\nu}' \rangle \models d(\vec{y}, \vec{y}')$.

Procedure Call: There exists a call edge e of the form presented in Diagram (2), such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_0^j$, and $\vec{\nu}' = (\xi', \vec{\bot}, \vec{\bot})$, where $\langle \vec{\nu}, \xi' \rangle \models d_{in}(\vec{y}, \vec{x}_2)$.

Return: There exists a procedure P_j (the procedure from which we return), such that $\ell = \ell_t^j$ (the terminal location of P_j). The run leading up to $\langle \ell; \vec{\nu} \rangle$ has a suffix of the form

$$\langle \ell_1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_1} \underbrace{\langle \ell_0^j; (\xi; \vec{\bot}; \vec{\bot}) \rangle \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_k} \langle \ell; (\xi; \eta; \zeta) \rangle}_{\sigma_1}$$

such that the segment σ_1 is *balanced* (has an equal number of *call* and *return* labels), $\lambda_1 = e$ is a call edge of the form presented in Diagram (2), where $\ell' = \ell_c$, $\lambda = $ *return*, and $\langle \vec{\nu}_1, \zeta, \vec{\nu}' \rangle \models d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$.
This definition uses the computation itself in order to retrieve the context as it were before the corresponding call to procedure P_j.

For a run $\sigma_1 : \langle \ell_0^0; (\xi, \vec{\bot}, \vec{\bot}) \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle \ell; \vec{\nu} \rangle$, we define the *level* of state $\langle \ell; \vec{\nu} \rangle$, denoted *Lev*$(\langle \ell; \vec{\nu} \rangle)$, to be the number of "call" edges in σ_1 minus the number of "return" edges.

4 Verifying Termination

This section presents a method for verifying termination of procedural programs. Initially, the system is augmented with well-founded ranking components. Then a finitary state abstraction is applied, resulting in a finite-state procedural program. Procedure summaries are computed over the abstract, finite-state program, and a procedure-free FDS is constructed. Finally, infeasibility of the derived FDS is checked, showing that it does not possess a fair divergent computation. This establishes the termination of the original program.

4.1 A Proof Rule for Termination

The application of a ranking abstraction to procedures is based on a rule for proving termination of loop-free procedural programs. We choose a well founded domain $(\mathcal{D}, \succ)$, such that for each procedure P_i with input parameters $\vec{x}$, we associate a *ranking function* δ_i that maps $\vec{x}$ to $\mathcal{D}$. For each edge e in P_i, labeled by a procedure call as shown in Diagram (2), we generate the descent condition $D_e(\vec{y}) : d_{in}(\vec{y}, \vec{x}_2) \;\rightarrow\; \delta_i(\vec{x}) \succ \delta_j(\vec{x}_2)$. The soundness of this proof rule is stated by the following claim:

Claim 1 (Termination). If the descent condition $D_e(\vec{y})$ is valid for every procedure call edge e in a loop-free procedural program P, then P terminates.

Proof: (Sketch) A non-terminating computation of a loop-free program must contain a subsequence of the form

$$\langle \ell_0^0; (\xi_0, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_0}^0; (\xi_0, \eta_0, \zeta_0) \rangle, \langle \ell_0^{j_1}; (\xi_1, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_1}^{j_1}; (\xi_1, \eta_1, \zeta_1) \rangle,$$
$$\langle \ell_0^{j_2}; (\xi_2, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_2}^{j_2}; (\xi_2, \eta_2, \zeta_2) \rangle, \langle \ell_0^{j_3}; (\xi_3, \vec{\bot}, \vec{\bot}) \rangle, \ldots$$

where, for each $k \geq 0$, *Lev*$(\langle \ell_0^{j_k}; (\xi_k, \vec{\bot}, \vec{\bot}) \rangle) = $ *Lev*$(\langle \ell_{i_k}^{j_k}; (\xi_k, \eta_k, \zeta_k) \rangle) = k$. If the descent condition is valid for all call edges, this leads to the existence of the infinitely descending sequence

$$\delta_0(\xi_0) \succ \delta_{j_1}(\xi_1) \succ \delta_{j_2}(\xi_2) \succ \delta_{j_3}(\xi_3) \succ \cdots$$

which contradicts the well-foundedness of the δ_i's. ■

Space limitations disallow a proof of the following completeness result:

Claim 2 (Completeness). The method of proving termination is complete for loop-free programs.

Validity of the condition D_e is to be interpreted semantically. Namely, $D_e(\vec{y})$ should hold for every $\vec{\nu}$, such that there exists a computation reaching location ℓ_a with $\vec{y} = \vec{\nu}$.

4.2 Ranking Augmentation of Procedural Programs

Ranking augmentation was suggested in [KP00] and used in [BPZ05] in conjunction with predicate abstraction to verify liveness properties of non-procedural programs. In its application here we require that a ranking function be applied only over the input parameters. Each procedure is augmented with a *ranking observer* variable that is updated at every procedure call edge e, in a manner corresponding to the descent condition D_e. For example, if the observer variable is *inc* then a call edge

$$d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2; \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$$

is augmented to be

$$d_{in}(\vec{y}, \vec{x}_2) \wedge \mathit{inc}' = \mathit{sign}(\delta(\vec{x}_2) - \delta(\vec{x}));\ P_j(\vec{x}_2; \vec{z}_2);\ d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \wedge \mathit{inc}' = 0$$

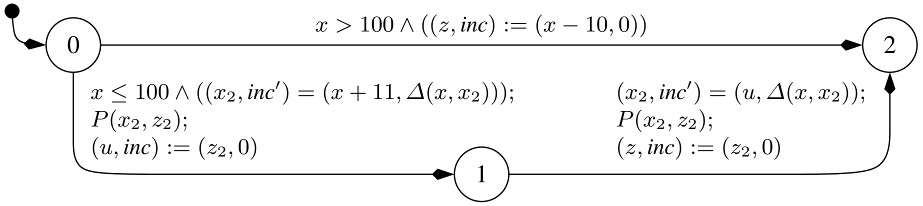
All local assignments are augmented with the assignment $\mathit{inc} := 0$, as the ranking does not change locally in a procedure. Well foundedness of the ranking function is captured by the compassion requirement $(\mathit{inc} < 0, \mathit{inc} > 0)$ which is being imposed only at a later stage.

Unlike the termination proof rule, the ranking function need not decrease on every call edge. Instead, a program can be augmented with multiple similar components, and it is up to the feasibility analysis to sort out their interaction and relevance automatically.

Example 2 (Ranking Augmentation of Program F_{91}). We now present an example of ranking abstraction applied to program F_{91} of Fig. 1. As a ranking component, we take

$$\delta(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } 0 \textbf{ else } 101 - x$$

Fig. 2 presents the program augmented by the variable *inc*. ■



0
$x > 100 \wedge ((z, \mathit{inc}) := (x - 10, 0))$
2
$x \leq 100 \wedge ((x_2, \mathit{inc}') = (x + 11, \Delta(x, x_2)));$
$P(x_2, z_2);$
$(u, \mathit{inc}) := (z_2, 0)$
$(x_2, \mathit{inc}') = (u, \Delta(x, x_2));$
$P(x_2, z_2);$
$(z, \mathit{inc}) := (z_2, 0)$
1

Fig. 2. Program F_{91} augmented by a Ranking Observer. The notation $\Delta(x_1, x_2)$ denotes the expression $\mathit{sign}(\delta(x_2) - \delta(x_1))$.

4.3 Predicate Abstraction of Augmented Procedural Programs

We consider the application of finitary abstraction to procedural programs, focusing on predicate abstraction for clarity. We assume a predicate base that is partitioned into $\vec{T} = \{\vec{I}(\vec{x}), \vec{W}(\vec{y}), \vec{R}(\vec{x},\vec{z})\}$, with corresponding abstract (boolean) variables $\vec{b}_T = \{\vec{b}_I, \vec{b}_W, \vec{b}_R\}$. For each procedure the input parameters, working variables, and output parameters are $\vec{b}_I$,$\vec{b}_W$, and $\vec{b}_R$, respectively.

An abstract procedure will have the same control-flow graph as its concrete counterpart, where only labels along the edges are abstracted as follows:

- A local change relation $d(\vec{y},\vec{y}')$ is abstracted into the relation

$$D(\vec{b}_T, \vec{b}'_T): \quad \exists \vec{y}, \vec{y}'.\, \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}'_T = \vec{T}(\vec{y}') \wedge d(\vec{y},\vec{y}')$$

- A procedure call $d_{in}(\vec{y},\vec{x}_2); P_j(\vec{x}_2,\vec{z}_2); d_{out}(\vec{y},\vec{z}_2,\vec{y}')$ is abstracted into the abstract procedure call $D_{in}(\vec{b}_T, \vec{b}^2_I); P_j(\vec{b}^2_I, \vec{b}^2_R); D_{out}(\vec{b}_T, \vec{b}^2_R, \vec{b}'_T)$, where

$$\begin{array}{ll} D_{in}(\vec{b}_T, \vec{b}^2_I) & : \exists \vec{y}, \vec{x}_2.\, \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}^2_I = \vec{I}(\vec{x}_2) \wedge d_{in}(\vec{y},\vec{x}_2) \\ D_{out}(\vec{b}_T, \vec{b}^2_R, \vec{b}'_T) & : \exists \vec{y}, \vec{x}_2, \vec{z}_2, \vec{y}' \left\{ \begin{array}{l} \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}^2_R = \vec{R}(\vec{x}_2,\vec{z}_2) \wedge \vec{b}'_T = \vec{T}(\vec{y}')\wedge \\ d_{in}(\vec{y},\vec{x}_2) \wedge d_{out}(\vec{y},\vec{z}_2,\vec{y}') \end{array} \right\} \end{array}$$

Example 3 (Abstraction of Program F_{91}).
We apply predicate abstraction to program F_{91} of Fig. 1. As a predicate base, we take

$$\vec{I}: \{x > 100\}, \qquad \vec{W}: \{u = g(x+11)\}, \qquad \vec{R}: \{z = g(x)\}$$

where

$$g(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91$$

The abstract domain consists of the corresponding boolean variables $\{B_I, B_W, B_R\}$. The abstraction yields the abstract procedural program $P(B_I, B_R)$ which is presented in Fig. 3. ⌟

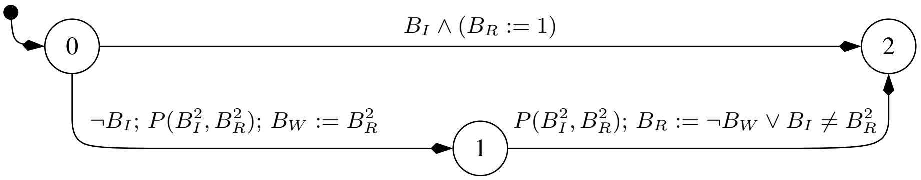

$B_I \wedge (B_R := 1)$
0
2
$\neg B_I$; $P(B_I^2, B_R^2)$; $B_W := B_R^2$
1
$P(B_I^2, B_R^2)$; $B_R := \neg B_W \vee B_I \neq B_R^2$

Fig. 3. An abstract version of Program F_{91}

Finally we demonstrate the joint (predicate and ranking) abstraction of program F_{91}.

Example 4 (Abstraction of Ranking-Augmented Program F_{91}).
We wish to abstract the augmented program from Example 2. When applying the abstraction based on the predicate set

$$\vec{I}: \{x > 100\}, \qquad \vec{W}: \{u = g(x+11)\}, \qquad \vec{R}: \{z = g(x)\}$$

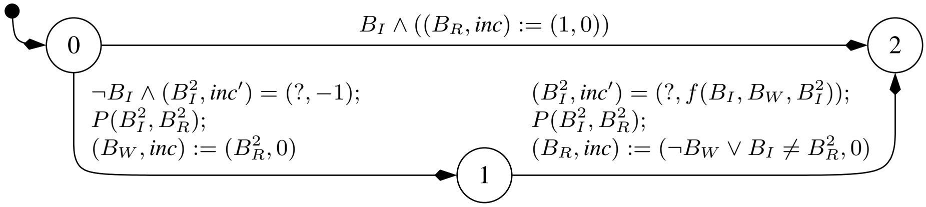

0
$B_I \wedge ((B_R, inc) := (1,0))$
2
$\neg B_I \wedge (B_I^2, inc') = (?, -1);$
$P(B_I^2, B_R^2);$
$(B_W, inc) := (B_R^2, 0)$
1
$(B_I^2, inc') = (?, f(B_I, B_W, B_I^2));$
$P(B_I^2, B_R^2);$
$(B_R, inc) := (\neg B_W \vee B_I \neq B_R^2, 0)$

Fig. 4. An abstract version of Program F_{91} augmented by a Ranking Observer

we obtain the abstract program presented in Fig. 4, where

$$f(B_I, B_W, B_I^2) \quad = \quad \begin{array}{lll} \textbf{if} & \neg B_I \wedge (B_I^2 \vee \neg B_I^2 \wedge B_W) & \textbf{then } -1 \\ \textbf{else if} & B_I \wedge B_I^2 & \textbf{then } 0 \\ & & \textbf{else } \ 1 \end{array}$$

Note that some (in fact, all) of the input arguments in the recursive calls are left non-deterministically 0 or 1. In addition, on return from the second recursive call, it is necessary to augment the transition with an adjusting assignment that correctly updates the local abstract variables based on the returned result.

It is interesting to observe that all terminating calls to this abstract procedure return $B_R = 1$, thus providing an independent proof that program F_{91} is partially correct with respect to the specification $z = g(x)$.

The analysis of this abstract program yields that $\neg B_I \wedge B_W$ is an invariant at location 1. Therefore, the value of $f(B_I, B_W, B_I^2)$ on the transition departing from location 1 will always be -1. Thus, it so happens that even without feasibility analysis, from Claim 1 we can conclude that the program terminates. ▟

4.4 Summaries

A procedure *summary* is a relation between input and output parameters. A relation $q(\vec{x}, \vec{z})$ is a summary if it holds for any $\vec{x}$ and $\vec{z}$ iff there exists a run in which the procedure is called and returns, such that the input parameters are assigned $\vec{x}$ and on return the output parameters are assigned $\vec{z}$.

Since procedures may contain calls (recursive or not) to other procedures, deriving summaries involves a fixpoint computation. An *inductive assertion network* is generated that defines, for each procedure P_j, a summary q^j and an assertion φ_a^j associated with each location ℓ_a. For each procedure we construct a set of constraints according to the rules of Table 1. The constraint $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \rightarrow q^j(\vec{x}, \vec{z})$ derives the summary from the assertion associated with the terminating location of P_j. All assertions, beside φ_0^j, are initialized false. φ_0^j, which refers to the entry location of P_j, is initialized true, i.e. it allows the input variables to have any possible value at the entry location of procedure P_j. Note that the matching constraint for an edge labeled with a call to procedure $P_i(\vec{x}_2; \vec{z}_2)$ encloses the summary of that procedure, i.e. the summary computation of one procedure comprises summaries of procedures being called from it.

An iterative process is performed over the constraints contributed by all procedures in the program, until a fixpoint is reached. Reaching a fixpoint is guaranteed since all variables are of finite type.

Table 1. Rules for Constraints contributed by Procedure P_j to the Inductive Assertion Network

Fact	**Constraint(s)**
	$\varphi_0^j = \mathsf{true}$ $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \rightarrow q^j(\vec{x}, \vec{z})$
$\ell_a \xrightarrow{d(\vec{y}, \vec{y}\,')} \ell_c$	$\varphi_a^j(\vec{y}) \ \wedge \ d(\vec{y}, \vec{y}\,') \rightarrow \varphi_c^j(\vec{y}\,')$
$\ell_a \xrightarrow{d_{in}(\vec{y}, \vec{x}_2)} \ell_c$	$\varphi_a^j(\vec{y}) \ \wedge \ d_{in}(\vec{y}, \vec{x}_2) \rightarrow \varphi_c^j(\vec{y}, \vec{x}_2)$
$\ell_a \xrightarrow{P_i(\vec{x}_2; \vec{z}_2)} \ell_c$	$\varphi_a^j(\vec{y}, \vec{x}_2) \ \wedge \ q^i(\vec{x}_2, \vec{z}_2) \rightarrow \varphi_c^j(\vec{y}, \vec{z}_2)$
$\ell_a \xrightarrow{d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')} \ell_c$	$\varphi_a^j(\vec{y}, \vec{z}_2) \ \wedge \ d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,') \rightarrow \varphi_c^j(\vec{y}\,')$

Claim 3 (Soundness). Given a minimal solution to the constraints of Table 1, q_j is a summary of P_j, for each procedure P_j.

Proof. In one direction, let $\sigma : s_0, \ldots, s_t$ be a computation segment starting at location ℓ_0^j and ending at ℓ_t^j, such that $\vec{x}[s_0] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$. It is easy to show by induction on the length of σ that $s_t \models \varphi_t^j(\vec{x}, \vec{u}, \vec{z})$. From Table 1 we obtain $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \rightarrow q^j(\vec{x}, \vec{z})$. Therefore $s_t \models q^j(\vec{x}, \vec{z})$. Since all edges satisfy $\vec{x} = \vec{x}\,'$, we obtain $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x}, \vec{y})$.

In the other direction, assume $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x}, \vec{y})$. From the constraints in Table 1 and the minimality of their solution, there exists a state s_t with $\vec{x}[s_t] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$ such that $s_t \models \varphi_t^j$. Repeating this reasoning we can, by propagating backward, construct a computation segment starting at ℓ_0 that initially assigns $\vec{v}_1$ to $\vec{x}$.

4.5 Deriving a Procedure-Free FDS

Using summaries of an abstract procedural program P_A, one can construct the *derived* FDS of P_A, labeled $\mathsf{derive}(P_A)$. This is an FDS denoting the set of *reduced computations* of P_A, a notion formalized in this section. The variables of $\mathsf{derive}(P_A)$ are partitioned into $\vec{x}, \vec{y}$, and $\vec{z}$, each of which consists of the input, working, and output variables of all procedures, respectively. The FDS is constructed as follows:

- Edges labeled by local changes in P_A are preserved in $\mathsf{derive}(P_A)$
- A procedure call in P_A, denoted by a sequence of edges of the form $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ from a location ℓ_a to a location ℓ_c, is transformed into the following edges:
  - A *summary* edge, specified by

$$\ell_a \xrightarrow[e]{\exists \vec{x}_2, \vec{z}_2 . d_{in}(\vec{y}, \vec{x}_2) \ \wedge \ q^j(\vec{x}_2, \vec{z}_2) \ \wedge \ d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')} \ell_c$$

  - A *call* edge, specified by

$$\ell_a \xrightarrow[e]{d_{in}(\vec{y}, \vec{x}\,')} \ell_0^j$$

- All compassion requirements, which are contributed by the ranking augmentation and described in Subsection 4.2, are imposed on $\mathsf{derive}(P_A)$.

The reasoning leading to this construction is that summary edges represent procedure calls that return, while call edges model non-returning procedure calls. Therefore, a summary edge leads to the next location in the calling procedure while modifying its variables according to the summary. On the other hand, a call edge connects a calling location to the entry location of the procedure that is being called. Thus, a nonterminating computation consists of infinitely many call edges, and a call stack is not necessary.

We now prove soundness of the construction. Recall the definition of a computation of a procedural program given in Subsection 3.2. A computation can be terminating or non-terminating. A terminating computation is finite, and has the property that every computation segment can be extended to a *balanced* segment, which starts with a calling step and ends with a matching return step. A computation segment is *maximally balanced* if it is balanced and is not properly contained in any other balanced segment.

Definition 1. *Let σ be a computation of P_A. Then the* reduction *of σ, labeled $\mathsf{reduce}(\sigma)$, is a sequence of states obtained from σ by replacing each maximal balanced segment by a summary-edge traversal step.*

Claim 4. For any sequence of states σ, σ is a computation of $\mathsf{derive}(P_A)$ iff there exists σ', a computation of P_A, such that $\mathsf{reduce}(\sigma') = \sigma$.

Proof of the claim follows from construction of $\mathsf{derive}(P_A)$ in a straightforward manner. It follows that if σ is a terminating computation of P_A, then $\mathsf{reduce}(P_A)$ consists of a single summary step in the part of $\mathsf{derive}(P_A)$ corresponding to P_0. If σ is an infinite computation of P_A, then $\mathsf{reduce}(\sigma)$ (which must also be infinite) consists of all assignment steps and calls into procedures from which σ has not returned.

Claim 5 (Soundness – Termination). If $\mathsf{derive}(P_A)$ is infeasible then P_A is a terminating program.

Proof. Let us define the notion of abstraction of computations. Let $\sigma = s_0, s_1, \ldots$ be a computation of P, the original procedural program from which P_A was abstracted. The abstraction of σ is a computation $\alpha(s_0), \alpha(s_1), \ldots$ where for all $i \geq 0$, if s_i is a state in σ, then $\alpha(s_i) = [\vec{b}_I \mapsto \vec{I}(\vec{x}), \vec{b}_W \mapsto \vec{W}(\vec{y}), \vec{b}_R \mapsto \vec{R}(\vec{x}, \vec{z})]$.

Assume that $\mathsf{derive}(P_A)$ is infeasible. Namely, every infinite run of $\mathsf{derive}(P_A)$ violates a compassion requirement. Suppose that P has an infinite computation σ. Consider $\mathsf{reduce}(\sigma)$ which consists of all steps in non-terminating procedure invocations within σ. Since the abstraction of $\mathsf{reduce}(\sigma)$ is a computation of $\mathsf{derive}(P_A)$ it must be unfair with respect to some compassion requirement. It follows that a ranking function keeps decreasing over steps in $\mathsf{reduce}(\sigma)$ and never increases – a contradiction. ▟

4.6 Analysis

The feasibility of $\mathsf{derive}(P_A)$ can be checked by conventional symbolic model-checking techniques. If it is feasible then there are two possibilities: (1) The original system truly diverges, or (2) feasibility of the derived system is *spurious*, that is, state and ranking abstractions have admitted behaviors that were not originally present. In the latter case, the method presented here can be repeated with a refinement of either state or ranking abstractions. The precise nature of such refinement is outside the scope of this paper.

5 LTL Model Checking

In this section we generalize the method discussed so far to general LTL model-checking. To this end we adapt to procedural programs the method discussed in Subsection 2.2 for model-checking LTL by composition with temporal testers [KPR98]. We prepend the steps of the method in Section 4 with a *tester composition step* relative to an LTL property. Once ranking augmentation, abstraction, summarization, and construction of the derived FDS are computed, the resulting system is model-checked by conventional means as to feasibility of initial states that do not satisfy the property.

The main issue is that synchronous composition of a procedural program with a global tester, including justice requirements, needs to be expressed in terms of local changes to procedure variables. In addition, since LTL is modeled over infinite sequences, the derived FDS needs to be extended with idling transitions.

5.1 Composition with Temporal Testers

A temporal tester is defined by a unique global variable, here labeled t, a transition relation $\rho(\vec{z}, t, \vec{z}\,', t')$[1] over primed and unprimed copies of the tester and program variables, where t does not appear in $\vec{z}$, and a justice requirement. In order to simulate global composition with ρ, we augment every procedure with the input and output parameters t_i and t_o, respectively, as follows:

- An edge labeled by a local change is augmented with $\rho(\vec{z}, t_o, \vec{z}\,', t_o')$
- A procedure call of the form $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{x}_2, \vec{y}\,')$ is augmented to be $d_{in}(\vec{y}, \vec{x}_2) \;\wedge\; \rho(\vec{z}, t_o, \vec{x}_2, t_i^2); P_j((\vec{x}_2, t_i^2), (\vec{z}_2, t_o^2)); d_{out} \;\wedge\; \rho(\vec{z}_2, t_o^2, \vec{z}\,', t_o')$
- Any edge leaving the initial location of a procedure is augmented with $t_o = t_i$

Example 5. Consider the program in Fig. 5. Since this program does not terminate, we are interested in verifying the property $\varphi : (\diamondsuit\, z) \;\vee\; \square \diamondsuit\; at\_\ell_2$, specifying that either eventually a state with $z = 1$ is reached, or infinitely often location 2 of P_1 is visited. To verify φ we decompose its negation into its principally temporal subformulas, $\square\, \neg z$ and $\diamondsuit\, \square\, \neg at\_\ell_2$, and compose the system with their temporal testers. Here we demonstrate the composition with $T[\square\, \neg z]$, given by the transition relation $t = \neg z \wedge t'$ and the trivial justice requirement true. The composition is shown in Fig. 6. ■

As a side remark, we note that our method can handle global variables in the same way as applied for global variables of testers, i.e., represent every global variable by a set of input and output parameters and augment every procedure with these parameters and with the corresponding transition relations.

5.2 Observing Justice

In order to observe justice imposed by a temporal tester, each procedure is augmented by a pair of *observer* variables that consists of a working and an output variables. Let J be a justice requirement, P_i be a procedure, and the associated observer variables be J_u and J_o. P_i is augmented as follows: On initialization, both J_u and J_o are assigned

[1] We assume here that the property to be verified is defined over the output variable only.

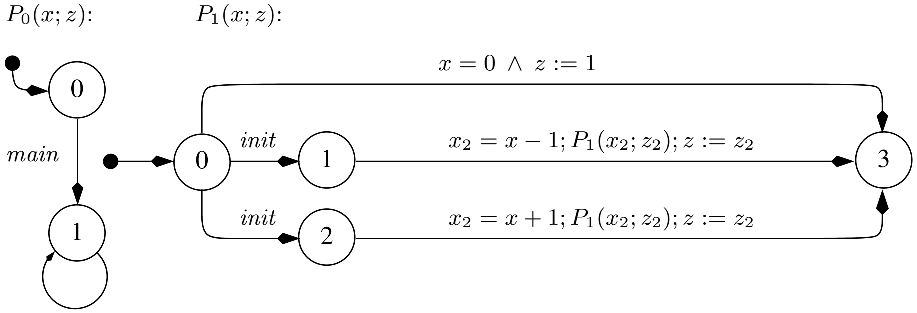

$P_0(x;z)$:
$P_1(x;z)$:
$x = 0 \wedge z := 1$
0
main
1
0
init
1
$x_2 = x - 1; P_1(x_2; z_2); z := z_2$
3
init
2
$x_2 = x + 1; P_1(x_2; z_2); z := z_2$

Fig. 5. A Divergent Program. *init* represents $x > 0 \wedge z := 0$, and *main* represents $x \geq 0 \wedge x_2 := x; P_1(x_2; z_2); z := z_2$.

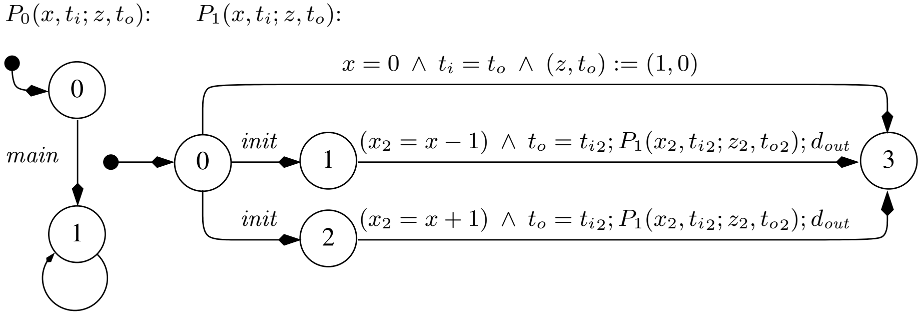

$P_0(x, t_i; z, t_o)$:
$P_1(x, t_i; z, t_o)$:
$x = 0 \wedge t_i = t_o \wedge (z, t_o) := (1, 0)$
0
main
1
0
init
1
$(x_2 = x - 1) \wedge t_o = t_{i2}; P_1(x_2, t_{i2}; z_2, t_{o2}); d_{out}$
3
init
2
$(x_2 = x + 1) \wedge t_o = t_{i2}; P_1(x_2, t_{i2}; z_2, t_{o2}); d_{out}$

Fig. 6. The Program of Fig. 5, Composed with $T[\Box \neg z]$. The assertion d_{out} represents $t_{o2} = (\neg z_2 \wedge t'_o) \wedge z := z_2$, *init* represents $x > 0 \wedge t_i = t_o \wedge z := 0$, and *main* represents $x \geq 0 \wedge (x_2 = x) \wedge (t_o = \neg z \wedge t_{i2}); P_1(x_2, t_{i2}; z_2, t_{o2}); d_{out}$.

true if the property J holds at that state. Local changes are conjoined with $J_u := J'$ and $J_o := (J_o \vee J')$. Procedure calls are conjoined with $J_u := (J' \vee J_o^2)$ and $J_o := (J_o \vee J' \vee J_o^2)$, where J_o^2 is the relevant output observer variable of the procedure being called.

While J_u observes J at every location, once J_o becomes true it remains so up to the terminal location. Since J_o participates in the procedure summary, it is used to denote whether justice has been satisfied within the called procedure.

5.3 The Derived FDS

We use the basic construction here in deriving the FDS as in Section 4.5. In addition, for every non-output observer variable J_u we impose the justice requirement that in any fair computation, J_u must be true infinitely often. Since LTL is modeled over infinite sequences, we must also ensure that terminating computations of the procedural program are represented by infinite sequences. To this end we simply extend the terminal location of procedure P_0 with a self-looping edge. Thus, a terminating computation is one that eventually reaches the terminal location of P_0 and stays idle henceforth.

In this section we use the notation $\mathsf{derive}(P_A)$ to denote the FDS that is derived from P_A and thus extended. The following claim of soundness is presented without proof due to space limitations.

Claim 6 (Soundness – LTL*).* Let P be a procedural program, φ be a formula whose principal operator is temporal, and P_A be the abstract program resulting from the composition of P with the temporal tester $T[\neg\varphi]$ and its abstraction relative to a state and ranking abstraction. Let t_o be the tester variable of $T[\neg\varphi]$. If $t_o = \mathtt{true}$ is an infeasible initial state of $\mathsf{derive}(P_A)$ then φ is valid over P.

6 Conclusion

We have described the integration of ranking abstraction, finitary state abstraction, procedure summarization, and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. Our approach is novel in that it reduces the verification problem of procedural programs with unbounded recursion to that of symbolic model-checking. Furthermore, it allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is being supported directly by the model, rather than being specified in a property.

We have implemented a prototype based on the TLV symbolic model-checker and tested several examples such as Ackerman's function, the Factorial function and a recursive formulation of the 91 function. We verified that they all terminate and model checked satisfiability of several LTL properties.

As further work it would be interesting to investigate concurrency with bounded context switching as suggested in [RQ05]. Another direction is the exploration of different versions of LTL that can relate to nesting levels of procedure calls, similar to the manner in which the CARET logic [AEM04] expresses properties of recursive state machines concerning the call stack.

References

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T.W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04*, pages 467–481.

[ACEM05] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS'05*, pages 61–765.

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV'01*, pages 207–220.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, pages 164–180.

[BR00] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'00*, pages 113–130.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*, pages 72–83.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *CAV'98*, pages 1–16.

[PSW05] A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In *ESOP'05*, pages 94–107.

[RQ05] J. Rehof and S. Qadeer. Context-bounded model checking of concurrent software. In *TACAS'05*, pages 93–107.

[Sha00] E. Shahar. *The TLV Manual*, 2000. http://www.cs.nyu.edu/acsys/tlv.

[SP81] M. Sharir and A. Pnueli. Two approaches to inter-procedural data-flow analysis. In Jones and Muchnik, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.