Technical Report UD-CIS-2011/03

Stephen F. Siegel and Timothy K. Zirkel*

Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
{siegel|zirkeltk}@udel.edu http://vsl.cis.udel.edu

Abstract. Symbolic execution is a powerful technique for automatically verifying properties of programs. Symbolic techniques have been developed for a variety of classes of assertions, to verify parallel as well as sequential programs, and even to verify functional equivalence of two programs. However, one limitation of these applications is that they typically require that constant (often small) bounds be imposed on the number of loop iterations. We present a generalization of symbolic execution that can reason about loops without bounds. This approach requires a user-provided loop invariant, but is otherwise fully automatic. Most importantly, it generalizes naturally to multi-process message-passing programs, and to verification of functional equivalence. We have realized the technique in the Toolkit for Accurate Scientific Software and demonstrated its effectiveness on several examples.

1 Introduction

Symbolic execution has been used to verify properties of complex sequential and parallel programs. While often effective, it has a serious limitation: it requires that a finite bound be placed on the number of iterations of each loop in the program. Moreover, the number of states tends to blow up as the iteration bound increases, and there is no way to know how large the bound must be before one can conclude that the properties hold for arbitrary numbers of iterations.

One solution to this problem is to use loop invariants. The technique described in [1] takes this approach. Upon arriving at the loop location, the invariant is checked, establishing the base case for an inductive proof. The variables modified in the loop body are then assigned fresh symbolic constants. After executing the loop body, control returns to the loop location and the invariant is checked again, establishing the inductive step.

The technique in [1] also attempts to discover an appropriate invariant automatically, working backwards from an assertion one wishes to prove after the loop. There have been many other advances in the automatic discovery of loop invariants in recent years, e.g., [2, 3]. However, none of these is guaranteed to find the right invariant, and in such cases user-provided invariants must be used.

* Supported by the U.S. National Science Foundation grants CCF-0733035 and CCF-0953210, and the University of Delaware Research Foundation

Java PathFinder [4] and Caduceus [5] both allow for annotating code with loop invariants, which are then used in verification.

In this paper we present an invariant-based loop technique which generalizes the approach of [1] in several ways. First, the components of the path condition are distributed over a *stack* which is maintained in the state. Elements are pushed onto and popped from the stack as control enters and exits the loop body, respectively, enabling a precise analysis of complex loop nests. Also, our method discovers the variables modified in the loop body dynamically, in a way that is almost perfectly precise. Most importantly, our method generalizes to multi-process, message-passing programs, such as those written using MPI [6], and can be used to verify the functional equivalence of two programs.

Our method currently requires a manually-provided invariant, though in future work we hope to incorporate automatic invariant generation techniques. Also, our invariants tend to be relatively simple. For one, they only need to reference those variables that are modified in the loop—relations involving other variables are automatically maintained, as those variables keep their original symbolic values. Also, when comparing two programs, the invariants tend to have a simple form: a conjunction of expressions of the form $x = y$, where x is a variable in one program and y is a corresponding variable in the other. There are other limitations: the user-provided invariant may not actually be invariant, though the method will at least detect and report this. A correct invariant may not be strong enough to prove a subsequent assertion, yielding a false alarm. A false alarm might also be issued because the theorem prover was unable to prove a valid assertion. The method might not converge, i.e., it may attempt to explore an infinite number of states. Finally, when an actual violation is found, the counterexample produced may not be as easily understood as one produced by a technique that explicitly iterates over the loop. In spite of these limitations, our method is *conservative*—if it reports an assertion holds, it must hold for all executions. We have demonstrated the effectiveness of the method on several examples, and it appears to be a promising approach to a very difficult problem.

There have been other approaches to functional equivalence verification. For example, Verdoolaege et al. have developed a technique for verifying functional equivalence of affine programs with static control flow [7]. It is fully automatic and does not require invariants or other hints from the user. However, it does not apply when a loop condition is non-affine (e.g., `j<i*i`) or to programs with non-static branch conditions, or to multi-process programs.

The paper is organized as follows: in §2 we provide the necessary background on symbolic execution. §3 provides a detailed discussion of the algorithm applied to sequential programs, including a proof of safety. §4 describes the extension to multi-process programs. We have implemented the full technique using the Toolkit for Accurate Scientific Software (TASS, [8]), a symbolic execution-based verification tool for C/MPI programs. In §5, we reports on a number of experiments using this implementation and draw conclusions.

2 Symbolic Execution Background

2.1 Program Graphs

Notation. For sets X and Y, let $\mathsf{Func}(X, Y)$ denote the set of all functions from X to Y. If $f \in \mathsf{Func}(X, Y)$, $x_0 \in X$ and $y_0 \in Y$, then $f[x_0 : y_0]$ denotes the function which maps x_0 to y_0 and otherwise agrees with f. If $S \subseteq X$ then $f(S) \stackrel{\text{def}}{=} \{y \in Y \mid \exists x \in X.y = f(x)\}$. Let $\mathbb{B} \stackrel{\text{def}}{=} \{\mathit{true}, \mathit{false}\}$.

Let $\mathsf{Val} \supseteq \mathbb{B}$ be a set of *values*, V a set of program *variables*, and $\mathsf{Eval}(V) = \mathsf{Func}(V, \mathsf{Val})$. Let $\mathsf{Expr}(V)$ denote the set of expressions over V. The semantics of expressions are defined by a function $\mathsf{eval}_V \colon \mathsf{Expr}(V) \times \mathsf{Eval}(V) \to \mathsf{Val}$. Let $\mathsf{BoolExpr}(V)$ be the subset of $\mathsf{Expr}(V)$ consisting of all expressions of boolean type: for any $g \in \mathsf{BoolExpr}(V)$ and $\eta \in \mathsf{Eval}(V)$, $\mathsf{eval}_V(g, \eta) \in \mathbb{B}$.

A *program graph* PG over V is a tuple $(\mathsf{Loc}, \mathsf{Act}, \mathsf{effect}, \mathsf{Tran}, \mathsf{Loc}_0, g_0)$ where

1. Loc is a set of *locations* and Act is a set of *actions*,
2. $\mathsf{effect} \colon \mathsf{Act} \times \mathsf{Eval}(V) \to \mathsf{Eval}(V)$ is the *effect function*,
3. $\mathsf{Tran} \subseteq \mathsf{Loc} \times \mathsf{BoolExpr}(V) \times \mathsf{Act} \times \mathsf{Loc}$ is the *conditional transition relation*,
4. $\mathsf{Loc}_0 \subseteq \mathsf{Loc}$ is a set of *initial locations*,
5. $g_0 \in \mathsf{BoolExpr}(V)$ is the *initial condition*.

$\mathsf{State} \stackrel{\text{def}}{=} \mathsf{Loc} \times \mathsf{Eval}(V)$ is the set of *concrete states* of PG. $\mathsf{State}_0 = \{\langle l, \eta\rangle \in \mathsf{State} \mid l \in \mathsf{Loc}_0 \wedge \mathsf{eval}_V(g_0, \eta) = \mathit{true}\}$ is the set of *initial states*. The *next-state function* $\mathsf{next} \colon \mathsf{State} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{State})$ is defined as follows: let $s = \langle l, \eta\rangle \in \mathsf{State}$, $t = \langle l'', g, \alpha, l'\rangle \in \mathsf{Tran}$. If $l = l''$ and $\mathsf{eval}_V(g, \eta) = \mathit{true}$, $\mathsf{next}(s, t) = \{\langle l', \mathsf{effect}(\alpha, \eta)\rangle\}$, else $\mathsf{next}(s, t) = \emptyset$. The set $\mathsf{Reach} \subseteq \mathsf{State}$ consists of all states s for which there exist $n \geq 0$, $s_0, \ldots, s_n \in \mathsf{State}$ and $t_1, \ldots t_n \in \mathsf{Tran}$ such that $s_0 \in \mathsf{SState}_0$, $s_n = s$ and $s_i \in \mathsf{next}(s_{i-1}, t_i)$ for $1 \leq i \leq n$.

2.2 Symbolic semantics

The syntax and semantics of symbolic expressions can be defined in different ways. A straightforward approach uses tree structures in which the leaf nodes are symbolic constants or concrete values, and other nodes correspond to operators. Different operators may be considered, and the set of expressions may be reduced modulo an equivalence relations. (cf. [9]). We take an axiomatic approach which is general enough to encompass all of these choices as specific cases.

Let $\mathcal{X}$ be a set of *symbolic constants*, SExpr a set of *symbolic expressions* over $\mathcal{X}$, and $\mathsf{Eval}(\mathcal{X}) \stackrel{\text{def}}{=} \mathsf{Func}(\mathcal{X}, \mathsf{Val})$. Let

$$\begin{aligned} \mathsf{eval}_{\mathcal{X}} &\colon \mathsf{SExpr} \times \mathsf{Eval}(\mathcal{X}) \to \mathsf{Val} \\ \mathsf{seval} &\colon \mathsf{Expr}(V) \times \mathsf{Func}(V, \mathsf{SExpr}) \to \mathsf{SExpr} \\ \mathsf{seffect} &\colon \mathsf{Act} \times \mathsf{Func}(V, \mathsf{SExpr}) \to \mathsf{Func}(V, \mathsf{SExpr}) \end{aligned}$$

be functions satisfying

$$\mathsf{eval}_V(e, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) = \mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(e, \xi), \theta) \tag{1}$$

$$\mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \mathsf{seffect}(\alpha, \xi) = \mathsf{effect}(\alpha, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) \tag{2}$$

for all $e \in \mathsf{Expr}(V)$, $\xi \in \mathsf{Func}(V, \mathsf{SExpr})$, $\theta \in \mathsf{Eval}(\mathcal{X})$, and $\alpha \in \mathsf{Act}$. Here, $\mathsf{eval}_{\mathcal{X}}(-, \theta)$ denotes the function from SExpr to Val which maps ϕ to $\mathsf{eval}_{\mathcal{X}}(\phi, \theta)$. The first function specifies how a symbolic expression is evaluated, given an assignment of concrete values to symbolic constants. The second defines how a program expression is evaluated, given a symbolic value for each variable, to yield a symbolic value. The third defines the semantics of the actions on the symbolic level. The constraints (1) and (2) essentially assert the consistency of the symbolic and concrete semantics.

Assume $\mathsf{Val} \cup \mathcal{X} \subseteq \mathsf{SExpr}$ and that $\mathsf{eval}_{\mathcal{X}}(\lambda, \theta) = \lambda$ and $\mathsf{eval}_{\mathcal{X}}(X, \theta) = \theta(X)$ for all $\lambda \in \mathsf{Val}$, $X \in \mathcal{X}$, and $\theta \in \mathsf{Eval}(\mathcal{X})$. Let $\mathsf{SBoolExpr}$ be the symbolic expressions of boolean type: $\mathsf{eval}_{\mathcal{X}}(\phi, \theta) \in \mathbb{B}$ for any $\phi \in \mathsf{SBoolExpr}$ and $\theta \in \mathsf{Eval}(\mathcal{X})$. Assume there is an operator $\wedge$ on $\mathsf{SBoolExpr}$ that has the obvious interpretation: $\mathsf{eval}_{\mathcal{X}}(\phi_1 \wedge \phi_2, \theta) = \mathsf{eval}_{\mathcal{X}}(\phi_1, \theta) \wedge \mathsf{eval}_{\mathcal{X}}(\phi_2, \theta)$ for all θ.

The set of *symbolic states* is

$$\mathsf{SState} \stackrel{\text{def}}{=} \mathsf{SBoolExpr} \times \mathsf{Loc} \times \mathsf{Func}(V, \mathsf{SExpr}).$$

The first component is the *path condition*. For each program variable $v \in V$, we choose a symbolic constant $X_v \in \mathcal{X}$ in such a way that the mapping $\xi_0 \colon V \to \mathsf{SExpr}$ defined by $\xi_0(v) = X_v$ is injective. (Assume $|\mathcal{X}| \geq |V|$, so this is possible.) Let $\phi_0 = \mathsf{seval}(g_0, \xi_0)$, and define the set of initial symbolic states by

$$\mathsf{SState}_0 = \{\langle \phi_0, l_0, \xi_0 \rangle \mid l_0 \in \mathsf{Loc}_0\}.$$

An expression $\phi \in \mathsf{SBoolExpr}$ is *valid* if $\mathsf{eval}_{\mathcal{X}}(\phi, \theta) = \mathit{true}$ for all $\theta \in \mathsf{Eval}(\mathcal{X})$. Let $\mathsf{valid} \colon \mathsf{SBoolExpr} \to \mathbb{B}$ be a function with the property that for any ϕ, if $\mathsf{valid}(\phi) = \mathit{true}$ then ϕ is valid. This function models the role of a conservative theorem prover. Define $\mathsf{nsat} \colon \mathsf{SBoolExpr} \to \mathbb{B}$ by $\mathsf{nsat}(\phi) = \mathsf{valid}(\neg\phi)$. If $\mathsf{nsat}(\phi) = \mathit{true}$ then ϕ is not satisfiable.

Define $\mathsf{snext} \colon \mathsf{SState} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{SState})$ as follows. Let $\hat{s} = \langle \phi, l, \xi \rangle \in \mathsf{SState}$ and $t = \langle l'', g, \alpha, l' \rangle \in \mathsf{Tran}$. If $l \neq l'' \vee \mathsf{nsat}(\phi \wedge \mathsf{seval}(g, \xi))$ then $\mathsf{snext}(\hat{s}, t) = \emptyset$. Otherwise

$$\mathsf{snext}(\hat{s}, t) = \{\langle \phi \wedge \mathsf{seval}(g, \xi), l', \mathsf{seffect}(\alpha, \xi) \rangle\} \tag{3}$$

The set SReach of reachable symbolic states is defined as usual.

2.3 Relations Between Concrete and Symbolic Semantics

Concretization. Each symbolic state $\hat{s}$ determines a set of concrete states $\gamma(\hat{s})$, where $\gamma \colon \mathsf{SState} \to \mathcal{P}(\mathsf{State})$ is defined by

$$\gamma(\langle \phi, l, \xi \rangle) = \{\langle l, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi \rangle \mid \theta \in \mathsf{Eval}(\mathcal{X}) \wedge \mathsf{eval}_{\mathcal{X}}(\phi, \theta)\}. \tag{4}$$

Let $\hat{s}, \hat{s}' \in \mathsf{SState}$. We say $\hat{s}$ *subsumes* $\hat{s}'$ if $\gamma(\hat{s}) \supseteq \gamma(\hat{s}')$. We say $\hat{s}$ and $\hat{s}'$ are *equivalent*, written $\hat{s} \sim \hat{s}'$, if $\gamma(\hat{s}) = \gamma(\hat{s}')$. Define $\hat{\gamma} \colon \mathcal{P}(\mathsf{SState}) \to \mathcal{P}(\mathsf{State})$ by $\hat{\gamma}(S) = \bigcup_{\hat{s} \in S} \gamma(\hat{s})$. The following are proved in [10]:

Theorem 1. *The following all hold:*

1. $\hat{\gamma}(\mathsf{SState}_0) = \mathsf{State}_0$,
2. $\hat{\gamma}(\mathsf{snext}(\hat{s}, t)) = \bigcup_{s \in \gamma(\hat{s})} \mathsf{next}(s, t)$ *for any* $\hat{s} \in \mathsf{SState}$ *and* $t \in \mathsf{Tran}$,
3. $\hat{\gamma}(\mathsf{SReach}) = \mathsf{Reach}$.

Properties. Suppose π is a predicate on State representing some "bad" quality, and the goal is to verify that $\pi(s)$ does not hold on any $s \in \mathsf{Reach}$. We say that a predicate $\hat{\pi}$ on SState is a *conservative lift* of π if for any $\hat{s} \in \mathsf{SState}$,

$$(\exists s \in \gamma(\hat{s}).\pi(s)) \Rightarrow \hat{\pi}(\hat{s}). \tag{5}$$

Suppose, for example that an assertion $\neg e \in \mathsf{Expr}(V)$ is associated to a location l_1. Thus $\pi(\langle l, \eta \rangle)$ holds if and only if $l = l_1 \wedge \mathsf{eval}_V(e, \eta) = \mathit{true}$. Define $\hat{\pi}$ by $\hat{\pi}(\langle \phi, l, \xi \rangle) = \mathit{true}$ if and only if $l = l_1 \wedge \mathsf{valid}(\phi \Rightarrow \mathsf{seval}(e, \xi))$. Then $\hat{\pi}$ is a conservative lift of π. The following is easily obtained from Theorem 1(3):

Corollary 1. *Let* $\hat{\pi}$ *be a conservative lift of the predicate* π*. If* $\exists s \in \mathsf{Reach}.\pi(s)$ *then* $\exists \hat{s} \in \mathsf{SReach}.\hat{\pi}(\hat{s})$*.*

3 The Loop Technique for Sequential Programs

The loop technique requires some additional structure in the program graph PG. Specifically, we assume that certain locations in the program graph are designated as *loop locations*. These have exactly two outgoing transitions: the *true* and *false* branches. Each acts as a no-op, i.e., the (concrete or symbolic) action associated to each is trivial. If the guard for the *true* branch is ϕ, the guard for the *false* branch is $\neg\phi$. The set of loop locations is denoted $\mathsf{LoopLoc}$, and the set of loop transitions LTran. The function $\mathsf{isTrue}\colon \mathsf{LTran} \to \mathbb{B}$ tells whether a loop transition corresponds to the *true* branch. In addition, we assume each $l \in \mathsf{LoopLoc}$ is given a loop invariant $\mathsf{assertion}(l) \in \mathsf{BoolExpr}(V)$.

The safety of the algorithm we are about to describe does not require any further assumptions: if the algorithm reports that a property holds then the property must hold on every concrete execution of PG. However, it is unlikely to be effective (i.e., to converge and not return a spurious violation) unless PG is generated from a program written in a language with structured "while" or "for" loops, without any jumps into or out of the loop bodies, and the loop locations correspond in the usual way to those loop statements.

3.1 Description of Algorithm

Additional symbolic operations. We first explain some additional operations in the symbolic framework that are required for the loop technique.

Let $\phi \in \mathsf{SExpr}$ and $Y \in \mathcal{X}$. We say ϕ is *independent of* Y if for all $\theta_1, \theta_2 \in \mathsf{Eval}(\mathcal{X})$ such that $\theta_1(x) = \theta_2(x)$ for all $x \in \mathcal{X} \setminus \{Y\}$, $\mathsf{eval}_{\mathcal{X}}(\phi, \theta_1) = \mathsf{eval}_{\mathcal{X}}(\phi, \theta_2)$. We assume there is a function which, given any ϕ, returns a finite list of symbolic constants $X_1, \ldots, X_n$, with the property that for any $Y \in \mathcal{X} \setminus \{X_1, \ldots, X_n\}$, ϕ is independent of Y. We say that the X_i are *involved in* ϕ. For example, if the

symbolic expressions are trees, this list could be computed by traversing the tree and accumulating the symbolic constants that occur in leaf nodes. We require that an ordered list be returned because the order will be used below to place symbolic expressions into a canonical form.

Assume there are operators $\vee$, $\neg$, and $\Rightarrow$ on $\mathsf{SBoolExpr}$, and that these have the obvious interpretations. Assume there is an operator $=\colon \mathsf{SExpr} \times \mathsf{SExpr} \to \mathsf{SBoolExpr}$ such that, for any $\theta \in \mathsf{Eval}(\mathcal{X})$, $\mathsf{eval}_{\mathcal{X}}(\phi_1 = \phi_2, \theta) = \mathit{true}$ if and only if $\mathsf{eval}_{\mathcal{X}}(\phi_1, \theta) = \mathsf{eval}_{\mathcal{X}}(\phi_2, \theta)$.

We assume there is an operator for substituting one symbolic constant for another in any symbolic expression. Formally, let $X, Y \in \mathcal{X}$. For any $\phi \in \mathsf{SExpr}$, $\phi[X \leftarrow Y] \in \mathsf{SExpr}$ and satisfies

$$\mathsf{eval}_{\mathcal{X}}(\phi[X \leftarrow Y], \theta) = \mathsf{eval}_{\mathcal{X}}(\phi, \theta[X : \theta(Y)]) \qquad (\forall \theta \in \mathsf{Eval}(\mathcal{X})). \tag{6}$$

For example, in a symbolic expression framework supporting real arithmetic, if $\phi = 2XY + Z$, then $\phi[X \leftarrow Y] = 2Y^2 + Z$.

The Loop Transition System. We now describe the extended transition system used by the loop technique. First, a *loop record* r is an element of

$$\mathsf{LoopRecord} \stackrel{\text{def}}{=} \mathsf{LoopLoc} \times \mathcal{P}(V) \times \mathsf{SBoolExpr}.$$

The components of r are denoted $r.\mathsf{location}$, $r.\mathsf{writeset}$, and $r.\mathsf{assumption}$, respectively. The set $r.\mathsf{writeset}$ is used to store the variables that may be modified in the course of executing the loop body corresponding to $r.\mathsf{location}$. The expression $r.\mathsf{assumption}$ is used to record the assumptions made in the course of executing the loop body.

A *loop state* s is an element of

$$\mathsf{LState} \stackrel{\text{def}}{=} \mathsf{SBoolExpr} \times \mathsf{Loc} \times \mathsf{Func}(V, \mathsf{SExpr}) \times \mathsf{LoopRecord}^*.$$

The components of s are denoted $s.\mathsf{permanentPC}$, $s.\mathsf{location}$, $s.\mathsf{val}$, and $s.\mathsf{stack}$, respectively. The first three components play the same role as the corresponding components in SState, except that $s.\mathsf{permanentPC}$ records only those assumptions that are made whenever control is not inside any loop (i.e., in the outermost scope). The fourth component is a finite sequence of loop records, treated as a stack. Entries are pushed onto the stack when control enters a loop, and are popped when control exits the loop.

Define $\mathsf{pc}(s) \stackrel{\text{def}}{=} s.\mathsf{permanentPC} \wedge \bigwedge_{r \in s.\mathsf{stack}} r.\mathsf{assumption}$. The *projection map* $\rho\colon \mathsf{LState} \to \mathsf{SState}$ is defined by $\rho(s) = \langle \mathsf{pc}(s), s.\mathsf{location}, s.\mathsf{eval} \rangle$. Hence ρ gathers together all of the individual "path condition variables," into a single path condition, and ignores all of the other information in the loop stack.

A loop state s is *initial* if $s.\mathsf{permanentPC} = \mathsf{seval}(g_0, \xi_0)$, $s.\mathsf{location} \in \mathsf{Loc}_0$, $s.\mathsf{eval} = \xi_0$, and $s.\mathsf{stack}$ is empty. The set of initial loop states is LState_0. We say that a state s in SState or LState *involves* symbolic constant Y if at least one of the symbolic expressions occurring in s involves Y.

```
1  procedure lnext(s: LState, t: LTran): LState is
2  |  let t = ⟨location, guard, α0, location′⟩;
3  |  if ¬empty(s.stack) ∧ top(s.stack).location = location then
4  |  |  isReentry ← true; W ← top(s.stack).writeset;
5  |  else isReentry ← false; W ← ∅;
6  |  let W = {v1, ..., vk};
7  |  if ∃i: s involves Yi then m ← the maximum such i; else m ← 0;
8  |  ψ ← ⋀_{i=1}^{k} (Y_{m+i} = s.val(vi));
9  |  val′ ← s.val[v1 : Y_{m+1}] ··· [vk : Y_{m+k}];
10 |  claim ← seval(assertion(location), val′);
11 |  χ ← seval(guard, val′);
12 |  θ ← s.permanentPC;
13 |  pc ← θ ∧ χ ∧ ψ ∧ ⋀_{r∈s.stack} r.assumption;
14 |  if ¬valid(pc ⇒ claim) then error("Possible invariant violation");
15 |  if isReentry then stack′ ← pop(s.stack); else stack′ ← s.stack;
16 |  if isTrue(t) then stack′ ← push(stack′, ⟨location, W, χ ∧ claim⟩);
17 |  else if empty(stack′) then θ ← θ ∧ χ ∧ claim;
18 |  else
19 |  |  r2 ← top(stack′);
20 |  |  stack′ ← push(pop(stack′), r2[assumption : r2.assumption ∧ χ ∧ claim]);
21 |  return canonic(⟨θ, location′, val′, stack′⟩);
```

Fig. 1. Next-state function for loop transitions in a sequential program.

In addition to the symbolic constants X_v used as the initial values for variables, we assume $\mathcal{X}$ contains symbolic constants $Y_1, Y_2, \ldots$. As described below, these will be assigned to variables modified in the loop body at the beginning of each iteration. New Y variables are introduced by the next-state function, but they also disappear as they are over-written, records are popped from the stack, and they are swept up in the garbage collection-like routine canonic.

We now define $\mathsf{lnext}\colon \mathsf{LState} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{LState})$. Once defined, the set LReach of reachable loop states is defined using LState_0 and lnext in the usual way. Let $s \in \mathsf{LState}$ and $t = \langle l, g, \alpha, l' \rangle \in \mathsf{Tran}$. If $l \neq s.\mathsf{location} \vee \mathsf{nsat}(\mathsf{pc}(s) \wedge \mathsf{seval}(g, \xi))$, $\mathsf{lnext}(s, t) = \emptyset$. Otherwise we proceed as follows.

Suppose first t is not a loop transition. If s.stack is empty, lnext is defined as in (3), but with the path condition modification being applied to s.permanentPC. If the stack is non-empty, the path condition modification is applied instead to the assumption component of the loop record at the top of the stack. In addition, any variable whose value changes as a result of applying seffect is added to r.writeset for every loop record r on the stack.

If t is a loop transition, $\mathsf{lnext}(s, t)$ is defined in Fig. 1. The algorithm first determines whether there is already an entry for l at the top of the stack. If so, this represents the return to l after executing the loop body (as opposed to the initial arrival at l from outside the body). In that case, the writeset from the previous iteration will be used for the set W.

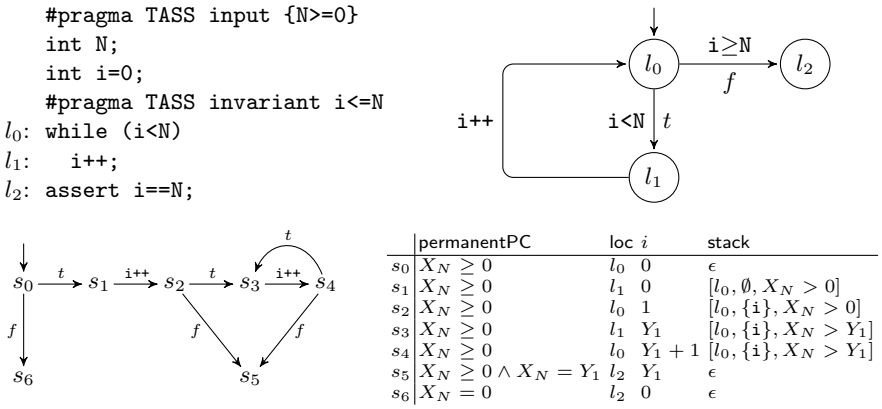
After finding an m such that s does not involve any Y_i with $i > m$, one such Y_i is introduced for each variable in W. The predicate ψ asserts that Y_i equals the current value of the corresponding variable, for each $v \in W$. Then the Y_i are assigned to the variables, overwriting the old values. Next, the loop invariant expression associated to l and the guard are evaluated in the resulting environment. Now the theorem prover is invoked to prove that under all current assumptions, the loop invariant holds; if this fails, an error is reported.

If this is a re-entry, the stack is popped. This is where the algorithm "forgets" any assumptions made during the previous iteration of the loop. What happens next depends on whether t represents the *true* or the *false* branch of the loop. If *true*, a new loop record is created and pushed onto the stack. In this new record, W is used for the writeset; in this way, the writeset can only increase with each successive iteration. The assumption is initialized to the conjunction of χ (the loop condition) and the invariant expression. Note that the relational expression ψ (which was used to establish the validity of the invariant) is *not* included. Hence the algorithm "forgets" the relation between the new symbolic constants and the original values of the variables. The only information "remembered" is that encoded in the invariant (claim) and the loop condition (χ).

If t is a *false* branch, then no new entry is pushed onto the stack. Instead, $\chi \wedge$ claim is added to an existing assumption: either the current top entry in the stack (if the stack is non-empty) or the permanentPC (if the stack is empty). Hence, upon exiting a loop, the record for that loop is popped, and only the information encoded in the invariant and the negation of the loop condition are recorded in the new state.

The function canonic is invoked before returning the new state. This function renames the Y_i involved in the state so that there are no gaps in the indexes. For example, if only Y_2, Y_3, and Y_7 are involved in s, then canonic might replace Y_2 with Y_1, Y_3 with Y_2, and Y_7 with Y_3, everywhere those constants occur. The Y_i are also placed in a canonical order, determined by placing a total order on the variables, and on the traversal of all symbolic expressions. The process is analogous to "heap canonicalization" performed by many model checkers for transforming equivalent heap configurations into a single representative form. It is done for the same reason: to help determine that a new state is equivalent to one that has been seen before. Without this step, convergence would not be possible, since m could simply increase with every loop iteration.

A Very Simple Example. The program of Fig. 2 has one input `N`, which is initialized to the symbolic constant X_N. The initial condition is `N` ≥ 0. The invariant associated to the loop is `i` $\leq$ `N`. The goal is to prove that the assertion `i` $=$ `N` holds on any execution. In the first iteration of the loop, `i` keeps its concrete value 0, as the algorithm has not yet "learned" that `i` is a variable that is modified in the loop body. When `i` is incremented to 1, `i` is added to the writeset. On the second iteration, `i` is assigned the new symbolic constant Y_1, at state s_3. The assumption in the loop record at this state is obtained by evaluating the loop condition `i<N` to get $Y_1 < X_N$ and the invariant `i<=N` to get $Y_1 \leq X_N$;



	permanentPC	loc	i	stack
s_0	$X_N \geq 0$	l_0	0	ϵ
s_1	$X_N \geq 0$	l_1	0	$[l_0, \emptyset, X_N > 0]$
s_2	$X_N \geq 0$	l_0	1	$[l_0, \{\mathtt{i}\}, X_N > 0]$
s_3	$X_N \geq 0$	l_1	Y_1	$[l_0, \{\mathtt{i}\}, X_N > Y_1]$
s_4	$X_N \geq 0$	l_0	$Y_1 + 1$	$[l_0, \{\mathtt{i}\}, X_N > Y_1]$
s_5	$X_N \geq 0 \wedge X_N = Y_1$	l_2	Y_1	ϵ
s_6	$X_N = 0$	l_2	0	ϵ

Fig. 2. A simple example to prove an assertion for all values of N. Top: source code with TASS annotations and program graph. Bottom: the reachable state space in the loop transition system; t represents the true branch and f the false branch.

the conjunction is equivalent to $X_N > Y_1$. In executing the *true* branch from s_4, we assign a new constant Y_2 to $\mathtt{i}$, the relational predicate ψ is $Y_1 + 1 = Y_2$, χ is $Y_2 < X_N$, and so pc is $Y_1 < X_N \wedge Y_1 + 1 = Y_2 \wedge Y_2 < X_N \wedge X_N \geq 0$. The theorem prover must prove $\mathsf{pc} \Rightarrow Y_2 \leq X_N$. The resulting state before canonicalization is

$$\langle X_N \geq 0, l_1, \{\mathtt{N} \mapsto X_N, \mathtt{i} \mapsto Y_2\}, [l_0, \{\mathtt{i}\}, Y_2 < X_N]\rangle.$$

Since this involves only Y_2, and not Y_1, $\mathsf{canonic}$ replaces all occurrences of Y_2 with Y_1, and the result is the state s_3 seen before. This illustrates how the loop technique may reduce an infinite-state problem to one that involves only a finite number of states. The assertion is verified in states s_5 and s_6. In the first case, this involves proving valid the expression $X_N \geq 0 \wedge X_N = Y_1 \Rightarrow Y_1 = X_N$. In the second case, the expression is $X_N = 0 \Rightarrow 0 = X_N$. All of these assertions are easily dispatched by automated theorem provers such as CVC3 [11], which is used by TASS.

3.2 Safety of the Sequential Technique

In this section we establish the safety of the loop technique for sequential programs. Our goal is to prove

Theorem 2. *Suppose* $\pi\colon \mathsf{State} \to \mathbb{B}$ *and* $\hat{\pi}\colon \mathsf{SState} \to \mathbb{B}$ *is a conservative lift of* π*. If there exists* $s \in \mathsf{Reach}$ *such that* $\pi(s) = true$*, then there exists* $\tilde{s} \in \mathsf{LReach}$ *such that* $\pi(\rho(\tilde{s})) = true$*.*

We begin with a lemma concerning transformations of symbolic states. While in a sense these are intuitively obvious, a formal proof is given in the appendix:

Lemma 1. *Let* $\hat{s} = \langle \phi, l, \xi \rangle \in \mathsf{SState}$*. The following hold:*

1. *Let* $v \in \mathsf{Var}$ *and* Y *be any symbolic constant such that* $\hat{s}$ *does not involve* Y*. Let* $\hat{s}' = \langle \phi', l, \xi' \rangle$*, where* $\xi' = \xi[v : Y]$ *and* $\phi' = (\phi \wedge Y = \xi(v))$*. Then* $\hat{s} \sim \hat{s}'$*.*
2. *Let* $X, Y \in \mathcal{X}$ *and suppose* $\hat{s}$ *does not involve* Y*. Let* $\hat{s}'$ *be the state in which any occurrence of* X *is replaced by* Y*. Then* $\hat{s}' \sim \hat{s}$*.*
3. *Suppose* $\phi, \phi' \in \mathsf{SBoolExpr}$ *and* $\phi \Rightarrow \phi'$ *is valid. Let* $\hat{s}' = \langle \phi', l, \xi \rangle \in \mathsf{SState}$*. Then* $\hat{s}'$ *subsumes* $\hat{s}$*.*

We use Lemma 1 to show

Proposition 1. *If* $\tilde{s} \in \mathsf{LState}$ *and* $t \in \mathsf{Tran}$*,* $\hat{\gamma}(\rho(\mathsf{lnext}(\tilde{s}, t))) \supseteq \hat{\gamma}(\mathsf{snext}(\rho(\tilde{s}), t))$*.*

Proof. Let $t = \langle l, g, \alpha, l' \rangle$. Say $\rho(\tilde{s}) = \langle \phi, l, \xi \rangle$. Without loss of generality, assume $l = \tilde{s}.\mathsf{location}$ and $\mathsf{nsat}(\phi \wedge \mathsf{seval}(g, \xi)) = \mathit{false}$ (else the right hand side is empty, and we are done). Let $\hat{s}_1 = \langle \phi \wedge \mathsf{seval}(g, \xi), l', \mathsf{seffect}(\alpha, \xi) \rangle$. By (3), $\mathsf{snext}(\rho(\tilde{s}), t) = \{\hat{s}_1\}$.

Suppose t is not a loop transition. Then $\mathsf{lnext}(\tilde{s}, t)$ is obtained from $\tilde{s}$ by adding $\mathsf{seval}(g, \xi)$ to the appropriate path condition variable, i.e., either to the $\mathsf{permanentPC}$ component or to the $\mathsf{assumption}$ component of a loop record on the stack, setting the $\mathsf{location}$ component to l', setting the eval component to $\mathsf{seffect}(\alpha, \xi)$, and possibly adding variables to the $\mathsf{writeset}$ components of loop records. The latter action has no impact on the projection. Hence $\rho(\mathsf{lnext}(\tilde{s}, t)) = \{\hat{s}_2\}$, where $\hat{s}_2 = \langle \zeta, l', \mathsf{seffect}(\alpha, \xi) \rangle$ and ζ is equivalent to $\phi \wedge \mathsf{seval}(g, \xi)$. Hence $\hat{s}_1 \sim \hat{s}_2$, and the two sets of concrete states are in fact equal.

Suppose t is a loop transition. Then the effect of lnext is equivalent to applying the following sequence of transformations to $\tilde{s}$. At each step, we keep track of what happens to the projection:

1. $\tilde{s}_1$ results from $\tilde{s}$ by adding the guard (ϕ) to one of the path condition variables and updating the location component to $\mathsf{location}'$; $\rho(\tilde{s}_1) \sim \hat{s}_1$,
2. $\tilde{s}_2$ results from $\tilde{s}_1$ by assigning a new symbolic constant to each variable in W and adding the corresponding relational predicate ψ to one of the path condition variables; by Lemma 1(1), $\rho(\tilde{s}_2) \sim \rho(\tilde{s}_1)$,
3. $\tilde{s}_3$ results from $\tilde{s}_2$ by adding claim to one of the path condition variables; since claim is already implied by the other path condition variables in $\tilde{s}_2$, $\rho(\tilde{s}_3) \sim \rho(\tilde{s}_2)$;
4. $\tilde{s}_4$ is obtained by removing ψ from whichever path condition variable it was added in step 2; by Lemma 1(3), $\rho(\tilde{s}_4)$ subsumes $\rho(\tilde{s}_3)$,
5. if $\mathsf{isReentry}$, $\tilde{s}_5$ is obtained from $\tilde{s}_4$ by popping the stack, else $\tilde{s}_5 = \tilde{s}_4$; by Lemma 1(3), $\rho(\tilde{s}_5)$ subsumes $\rho(\tilde{s}_4)$,
6. $\tilde{s}_6$ is obtained from $\tilde{s}_5$ by applying $\mathsf{canonic}$; by repeated applications of Lemma 1(2), $\rho(\tilde{s}_6) \sim \rho(\tilde{s}_5)$.

The result is that $\mathsf{lnext}(\tilde{s}, t) = \{\tilde{s}_6\}$ and $\rho(\tilde{s}_6)$ subsumes $\hat{s}_1$, as required. □

In the sequence of transformations in the proof, ψ is added to the state and later removed. This is necessary for the proof, but of course the end result is equivalent to never adding ψ in the first place, the approach taken in Fig. 1.

We can now establish that the reachable states in the loop transition system "cover" all reachable concrete states:

Proposition 2. $\hat{\gamma}(\rho(\mathsf{LReach})) \supseteq \mathsf{Reach}$.

Proof. Let $S = \hat{\gamma}(\rho(\mathsf{LReach}))$. We first show $S \supseteq \mathsf{State}_0$. Let $s \in \mathsf{State}_0$. By Theorem 1(1), there exists $\hat{s} = \langle \phi_0, l_0, \xi_0 \rangle \in \mathsf{SState}_0$ such that $s \in \gamma(\hat{s})$. Let $\tilde{s} = \langle \phi_0, l_0, \xi_0, \epsilon \rangle \in \mathsf{LState}_0$, where ϵ is the empty stack. Then $\rho(\tilde{s}) = \hat{s}$, so $s \in \gamma(\rho(\tilde{s}))$, as required.

We now assume $s \in S$ and $t \in \mathsf{Tran}$ and will show $\mathsf{next}(s,t) \subseteq S$. Say $s \in \gamma(\hat{s})$, where $\hat{s} = \rho(\tilde{s})$ and $\tilde{s} \in \mathsf{LReach}$. By Theorem 1(2) and Proposition 1,

$$\mathsf{next}(s,t) \in \hat{\gamma}(\mathsf{snext}(\hat{s},t)) \subseteq \hat{\gamma}(\rho(\mathsf{lnext}(\tilde{s},t))) \subseteq \hat{\gamma}(\rho(\mathsf{LReach})) = S.$$

Any set that contains State_0 and is closed under next contains Reach. □

The proof of Theorem 2 follows easily: suppose $s \in \mathsf{Reach}$ and $\pi(s) = \mathit{true}$. By Proposition 2, there exists $\tilde{s} \in \mathsf{LReach}$ such that $s \in \gamma(\rho(\tilde{s}))$. Since $\hat{\pi}$ is a conservative lift of π, $\hat{\pi}(\rho(\tilde{s})) = \mathit{true}$, as required.

4 The Loop Technique for Multi-process Programs

We consider a message-passing system with $n \geq 1$ processes. This includes equivalence checking of two programs: as described in [12], if one program has n processes and the other m, one may unite them into a single system with $n + m$ processes. For a loop which does not involve communication with other processes and for which the invariant is local, the sequential technique can be used without modification. For loops which "cut across" several processes, such as those in Fig. 4(b) and (c), a more general technique is required. The primary challenge is that processes may enter and leave the loop at different times.

The multi-process system can be modeled as a *channel system* CS [12]. For $1 \leq i \leq n$, let V_i be the set of variables for for process i. Assume $V_i \cap V_j = \emptyset$ if $i \neq j$. Let $V = \cup_{i=1}^{n} V_i$. Each process is represented as a program graph PG_i over V_i. In addition, there is a set Chan of FIFO channels, each with a specified capacity, used to store messages in transit from one process to another. The transitions of CS consist of the usual kind described in §2.1, local to a single a process, as well as send and receive operations on channels. Let $\mathsf{ProcState}_p = \mathsf{Loc}_p \times \mathsf{Func}(V_p, \mathsf{Val})$, the set of states for process p. The global state of CS consists of a process state for each p and the state of the channels.

We briefly summarize the notion of *collective assertion*; for details, see [12]. A collective assertion σ is an assertion that spans multiple processes. Each such σ is identified by a unique identifier $\mathsf{id} = \mathsf{id}(\sigma)$ in some set Id. The set of processes involved in id is denoted $\mathsf{procs}(\mathsf{id}) \subseteq \mathsf{Proc}$. Associated to id is a location l_p for each $p \in \mathsf{procs}(\mathsf{id})$. To each such location there is associated an expression $\mathsf{assertion}(l_p) \in \mathsf{BoolExpr}(V)$ which may involve variables from other processes. The semantics are defined as follows: when control in process p reaches l_p a "snapshot" of the process state is placed in a special FIFO queue for p. As soon as there is at least one snapshot in each queue, one entry is dequeued from each and these are composed into a global state in which all of the assertion

```
1  procedure execLoop(s: MPLState, t: LTran): MPLState is
2  |  let t = ⟨l, guard, α_0, l'⟩;
3  |  p ← proc(t);
4  |  r_0 ← currentLoopRecord(p, s);
5  |  if r_0 ≠ null ∧ r_0.id = id(l) then W ← r_0.writeset;
6  |  else W ← ∅;
7  |  if ∃i: s.queue[i].snapshots(p) = null then
8  |  |  let i_1 be the least such i; r_1 ← s.queue[i_1]; isNew ← false;
9  |  |  if r_1.id ≠ id(l) then error("out of order");
10 |  |  if r_1.isTrue ≠ isTrue(t) then error("conflicting loop exit");
11 |  else isNew ← true; r_1 ← ⟨id(l), λq.null, λq.∅, isTrue(t), true, true⟩;
12 |  let W = {v_1, ..., v_k};
13 |  if ∃i: Y_i is involved in s then m ← the maximum such i; else m ← 0;
14 |  eval' ← s.procStates(p).eval[v_1 : Y_{m+1}] ··· [v_k : Y_{m+k}];
15 |  procState' ← procStates(p)[eval ← eval'];
16 |  φ ← r_1.assumption ∧ seval(guard, eval');
17 |  ψ ← r_1.relation ∧ v_1 = Y_{m+1} ∧ ··· ∧ v_k = Y_{m+k};
18 |  r' ← ⟨id(l), r_1.snapshots[p : procState'], r_1.writeset[p : W], isTrue(t), φ, ψ⟩;
19 |  if isNew then queue' ← enqueue(s.queue, r');
20 |  else queue' ← s.queue[i_1 : r'];
21 |  procStates' ← s.procStates[p : procState'[location : l']];
22 |  return ⟨permanentPC, procStates', s.buffer, queue', s.stack⟩;
23 procedure completeLoop(s: MPLState): MPLState is
24 |  if empty(s.queue) then return s;
25 |  r_1 ← first(s.queue);
26 |  if ∃p: r_1.snapshots(p) = null then return s;
27 |  ξ ← ⋃_{p∈procs(r_1.id)} r_1.snapshots(p).eval;
28 |  claim ← seval(⋀_{p∈procs(r_1.id)} assertion(r_1.snapshots(p).location), ξ);
29 |  pc ← s.permanentPC ∧ ⋀_{r∈s.queue∪s.stack} r.assumption ∧ r.relation;
30 |  if ¬valid(pc ⇒ claim) then error("Possible invariant violation");
31 |  r'_1 ← r_1[assumption : r_1.assumption ∧ claim][relation ← true];
32 |  queue' ← dequeue(s.queue);
33 |  stack' ← s.stack;
34 |  if ¬empty(stack') ∧ top(stack').id = r_1.id then stack' ← pop(stack');
35 |  φ ← s.permanentPC;
36 |  if r'_1.isTrue then stack' ← push(stack', r'_1);
37 |  else if empty(stack') then φ ← φ ∧ r'_1.assumption;
38 |  else
39 |  |  r_2 ← top(stack');
   |  |  stack' ← push(pop(stack'), r_2[assumption : r_2.assumption ∧ r'_1.assumption]);
40 |  return ⟨φ, s.procStates, s.buffer, queue', stack'⟩;
```

Fig. 3. The next-state function for loop transitions in a multi-process program. Given state s and loop transition t, the next state is canonic(completeLoop(execLoop(s, t))).

expressions are evaluated. The assertion passes if and only if all of these evaluate to *true*. If a program involves several collective assertions, these are required to be encountered in the same order on each process, else the program is erroneous. It is also erroneous if a program terminates with any collective assertion queue non-empty. Collective assertions can be verified using standard symbolic techniques that incorporate the collective queues into the state.

Just as a loop invariant is a kind of assertion, a *collective loop invariant* is a collective assertion for which the associated locations are all loop locations. All of the usual requirements for collective assertions must hold for collective invariants. In addition, the associated loops in different processes must iterate the same number of times, else the program is erroneous.

We again define a symbolic state-transition system. First, a *multi-process loop record* is tuple with the following components:

1. $\mathsf{id} \in \mathsf{Id}$, an identifier for the collective operation,
2. $\mathsf{snapshots}$, a function which associates to each $p \in \mathsf{procs}(\mathsf{id})$ either the symbol null or an element of $\mathsf{ProcState}_p$, the snapshot of the state of the process when it reached the loop location,
3. $\mathsf{writeset}$, a function which associates to each $p \in \mathsf{procs}(\mathsf{id})$ a subset of Var_p,
4. $\mathsf{isTrue} \in \mathbb{B}$: is this record for the execution of the *true* branch?,
5. $\mathsf{assumption} \in \mathsf{SBoolExpr}$, path condition for this loop execution, and
6. $\mathsf{relation} \in \mathsf{SBoolExpr}$, a predicate relating new symbolic constants to values from the previous iteration.

The set of multi-process loop records is denoted $\mathsf{MPLRecord}$.

A *multi-process loop state* is a structure with the following components:

1. $\mathsf{permanentPC} \in \mathsf{SBoolExpr}$, the permanent component of the path condition,
2. $\mathsf{procStates} \in \mathsf{ProcState}_0 \times \cdots \times \mathsf{ProcState}_{n-1}$, the state of each process,
3. $\mathsf{buffer} \in \mathsf{Eval}(\mathsf{Chan})$, the value of each channel,
4. $\mathsf{queue} \in \mathsf{LoopRecord}^*$, queue of incomplete loop records, and
5. $\mathsf{stack} \in \mathsf{LoopRecord}^*$, stack of loops completely entered.

The set of all multi-process loop states is denoted $\mathsf{MPLState}$.

The next-state function for loop transitions is given in Fig. 3. Space does not permit a detailed description and proof of safety, but the basic ideas are the same as those for the sequential technique. We discuss some of its salient points.

The procedure is broken down into two steps. The first step substitutes the new symbolic constants and enters the information for the loop action into the appropriate place in the queue. This information includes the loop location, process snapshot, and the relational predicate. Only if this results in a *complete* record, i.e., one in which every process has made an entry, do we enter the second step, which dequeues the record and proceeds to modify the stack in a way that is very similar to the sequential case.

The queue can hold entries for *false* as well as *true* branches; the stack only holds entries for *true* branches. The relational predicates, which form part of the path condition, are recorded in the queue because they are needed to check the

validity of the invariant. This check can only take place once every process has reached the loop location. Once checked, the relational predicates are discarded, and the newly freed symbolic constants can be swept up.

Function currentLoopRecord(p) returns the record corresponding to the innermost loop process p is "in" in the current state. This record may be in the queue or the stack. (Since *true* and *false* branches occur in the queue, computing this is not as simple as looking at the latest entry for p.) This function is used when adding an assumption to the path condition, adding a variable to the writeset, or in determining if a loop transition is a "re-entry."

Finally, for $n = 1$, the multi-process technique reduces to the sequential one.

The safety of the multi-process technique can be established much as in the sequential case. Of course, convergence is not guaranteed. A necessary condition for convergence is that the loop "skew" (the maximum difference in iteration count between any two processes) be bounded. This is where an appropriate partial order reduction technique can help. Such a scheme produces a reduced subspace of the state space, exploiting the fact that in many cases, when multiple processes are enabled, it is safe to consider only the transitions emanating from a single process. When more than one process falls into this category, the POR scheme is free to choose any one of them. Following the *collective queue minimization* scheme of [12], we choose a process which has the smallest number of entries in the queue. This has the effect of keeping the process iteration counts as close together as possible. For the 2-process system of Fig. 4(c), used to compare two sequential programs, the two processes are completely independent, and the POR scheme effectively imposes a barrier at the top of the loop. The same happens in the MPI program (b).

5 Experiments and Conclusions

We have implemented the multi-process loop technique by extending the TASS collective assertion facility. The invariants are encoded as pragmas immediately preceding a *while* or *for* loop. The syntax is exactly the same as that for collective assertions except that the keyword `invariant` is used in place of `assert`.

We applied the technique to 7 examples. Each uses an integer input N and the goal is to verify a property for all N. The examples are: (1) `matrix`, a single sequential program for adding two $N \times N$ matrices in which we verify functional correctness for all N, (2) `count`, a multi-process program where each process loops from 1 to N and we check the loop variable equals N at termination, (3) `ring`, a single MPI program where processes send right and receive left N times and we verify absence of potential deadlock, (4) `mean`, two sequential programs for computing the mean (average) of an array of doubles, (5) `fib`, two sequential programs for computing the limiting ratio of two consecutive terms in the Fibonacci sequence to within a given tolerance, (6) `nested`, two sequential programs, one computing $\sum_{i=1}^{n} \sum_{j=1}^{i^2} ij$, the other $\sum_{i=1}^{n} i \sum_{j=1}^{i^2} j$, (7) `diffusion`, two programs solving the 1d-diffusion equation, one sequential, one MPI. In the last four examples, functional equivalence was verified.

```
#pragma TASS invariant L1 (i>=0 && i<=N) && (forall {int xi | 0<=xi && xi<i} \
    forall {int xj | 0<=xj && xj<N} c[xi][xj]==a[xi][xj]+b[xi][xj]);
while (i<N) {
  j=0;
#pragma TASS invariant L2 (j>=0 && j<=N) && (forall {int xi | 0<=xi && xi<i} \
    forall {int xj | 0<=xj && xj<N} c[xi][xj]==a[xi][xj]+b[xi][xj])          \
    && (forall {int xj | 0<=xj && xj<j} c[i][xj]==a[i][xj]+b[i][xj]);
  while (j<N) {
    c[i][j]=a[i][j]+b[i][j];
    j++;
  }
  i++;
}
```

(a) Matrix addition program with invariants (excerpt).

```
#pragma TASS collective invariant L i==PROC[right]@main.i;
while (i<N) {
  if (myrank%2==0) {
    MPI_Send(NULL, 0, MPI_INT, right, 0, MPI_COMM_WORLD);
    MPI_Recv(NULL, 0, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(NULL, 0, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(NULL, 0, MPI_INT, right, 0, MPI_COMM_WORLD);
  }
  i++;
}
```

(b) Ring: MPI program in which each process sends to right, receives from left.

```
#pragma TASS joint invariant LOOP true;
while (err>=tol) {
  i=j; j=k; k=i+j; tmp = k/j;
  if (tmp>=p) err=tmp-p; else err=p-tmp;
  p = tmp;
}
```

```
#pragma TASS joint invariant LOOP   \
  err==spec.err && p==spec.p && j>0 \
  && j==spec.j && k==spec.k && k>0;
while (err>=tol) {
  k=j+k; j=k-j; err=k/j-p; p=err+p;
  if (err < 0) err=-err;
}
```

(c) Fibonacci. Two functionally equivalent programs to compute ϕ.

Fig. 4. Examples of loop invariants and collective invariants.

name	nprocs	trans	proofs	time
matrix	1	30	103	5.2
count	10	138	55	3.7
ring	10	1521	70	4.3
mean	1+1	28	18	0.5
fib	1+1	60	29	1.4
nested	1+1	79	44	1.1
diffusion	1+10	3580	82	34.4

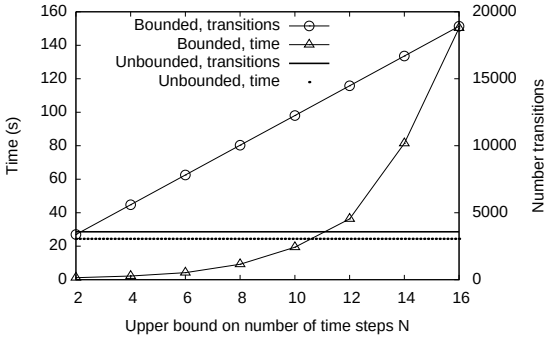


Fig. 5. Results of experiments. Left: work required to verify for all N using loop technique: number of processes (two numbers for equivalence verification), transitions, calls to theorem prover CVC3, and time (seconds). Right: verifying diffusion equivalence: using various upper bounds vs. using loop technique to verify for all N.

In each case, we were able to formulate invariants that allowed TASS to verify the property for all N. Three of the cases are shown in Fig. 4 and performance data is given Fig. 5; all of the experimental artifacts can be obtained from [8]. For the most challenging example, diffusion, we also compare the performance of the loop technique with the standard technique using various upper bounds on N. Two facts stands out: (1) the number of transitions explored in the loop technique is usually comparable to that number for a very small bound, but (2) the time consumed per transition in the loop technique is very large. Inspection reveals that most of this time is due to the increased number and complexity of the theorem prover calls. Our future work will seek to reduce the cost of these invocations, as well as ways to find the loop invariants automatically.

References

1. Păsăreanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In Graf, S., Mounier, L., eds.: Model Checking Software: 11th Intl. SPIN Workshop. Volume 2989 of LNCS., Springer (2004) 164–181
2. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL '02, ACM (2002) 191–202
3. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In Chechik, M., Wirsing, M., eds.: FASE. Volume 5503 of LNCS. Springer Berlin / Heidelberg (2009) 470–485
4. Anand, S., Păsăreanu, C., Visser, W.: JPF–SE: A symbolic execution extension to Java PathFinder. In Grumberg, O., Huth, M., eds.: TACAS. Volume 4424 of LNCS. Springer Berlin / Heidelberg (2007) 134–138
5. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. [13] 173–177
6. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009. `http://www.mpi-forum.org/docs/` (2009)
7. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In Bouajjani, A., Maler, O., eds.: CAV 2009. Volume 5643 of LNCS., Springer (2009) 599–613
8. S. F. Siegel et al.: The Toolkit for Accurate Scientific Software web page. `http://vsl.cis.udel.edu/tass` (2010)
9. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2) (2008) Article 10, 1–34
10. Siegel, S.F., Zirkel, T.K.: The Toolkit for Accurate Scientific Software. Technical Report UD-CIS-2011/01, University of Delaware (2011)
11. Barrett, C., Tinelli, C.: CVC3. [13] 298–302
12. Siegel, S.F., Zirkel, T.K.: Collective assertions. In Jhala, R., Schmidt, D., eds.: VMCAI 2011. Volume 6538 of LNCS. (2011) 387–402
13. Damm, W., Hermanns, H., eds.: CAV 2007. In Damm, W., Hermanns, H., eds.: CAV 2007. Volume 4590 of LNCS., Springer (2007)

A Proof of Lemma 1

Part (1): Let $s = \langle l, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi \rangle \in \gamma(\hat{s})$, where $\theta \in \mathsf{Eval}(\mathcal{X})$ and $\mathsf{eval}_{\mathcal{X}}(\phi,\theta) = \mathit{true}$. Let $\theta' = \theta[Y : \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta)]$. Since ϕ is independent of Y, $\mathsf{eval}_{\mathcal{X}}(\phi,\theta') = \mathsf{eval}_{\mathcal{X}}(\phi,\theta)$. Furthermore,

$$\mathsf{eval}_{\mathcal{X}}(Y,\theta') = \theta'(Y) = \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta) = \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta'),$$

so $\mathsf{eval}_{\mathcal{X}}(Y = \xi(v),\theta') = \mathit{true}$. Hence

$$\mathsf{eval}_{\mathcal{X}}(\phi',\theta') = \mathsf{eval}_{\mathcal{X}}(\phi,\theta') \wedge \mathsf{eval}_{\mathcal{X}}(Y = \xi(v),\theta') = \mathsf{eval}_{\mathcal{X}}(\phi,\theta) \wedge \mathit{true} = \mathit{true}.$$

Moreover,

$$\begin{aligned} \mathsf{eval}_{\mathcal{X}}(\xi'(w),\theta') &= \mathsf{eval}_{\mathcal{X}}(\xi(w),\theta') = \mathsf{eval}_{\mathcal{X}}(\xi(w),\theta) \qquad (\text{for all } w \in V \setminus \{v\}) \\ \mathsf{eval}_{\mathcal{X}}(\xi'(v),\theta') &= \mathsf{eval}_{\mathcal{X}}(Y,\theta') = \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta). \end{aligned}$$

Hence $\mathsf{eval}_{\mathcal{X}}(-,\theta') \circ \xi' = \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi$, so $s \in \gamma(\hat{s}')$. This shows $\gamma(s) \subseteq \gamma(s')$. The argument that $\gamma(s') \subseteq \gamma(s)$ is similar.

Part (2): Let $\hat{s} = \langle \phi, l, \xi \rangle$. Define $\xi' \in \mathsf{Func}(V, \mathsf{SExpr})$ by $\xi'(v) = \xi(v)[X \leftarrow Y]$, and let $\phi' = \phi[X \leftarrow Y]$. Then $\hat{s}' = \langle \phi', l, \xi' \rangle$.

Let $s = \langle l, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi \rangle \in \gamma(\hat{s})$, where $\theta \in \mathsf{Eval}(\mathcal{X})$ and $\mathsf{eval}_{\mathcal{X}}(\phi,\theta) = \mathit{true}$. Let $\theta' = \theta[Y : \theta(X)]$. Note that $\theta'(Y) = \theta(X) = \theta'(X)$. Hence $\theta'[X : \theta'(Y)] = \theta'[X : \theta'(X)] = \theta'$. Moreover, θ and θ' agree, except possibly at Y. By the semantics of the substitution operator and the fact that ϕ is independent of Y,

$$\mathsf{eval}_{\mathcal{X}}(\phi',\theta') = \mathsf{eval}_{\mathcal{X}}(\phi,\theta'[X : \theta'(Y)]) = \mathsf{eval}_{\mathcal{X}}(\phi,\theta') = \mathsf{eval}_{\mathcal{X}}(\phi,\theta) = \mathit{true}.$$

It follows that $s' \stackrel{\text{def}}{=} \langle l, \mathsf{eval}_{\mathcal{X}}(-,\theta') \circ \xi' \rangle \in \gamma(\hat{s}')$. Now, for any $v \in V$,

$$\begin{aligned} \mathsf{eval}_{\mathcal{X}}(\xi'(v),\theta') = \mathsf{eval}_{\mathcal{X}}(\xi(v)[X \leftarrow Y],\theta') &= \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta'[X : \theta'(Y)]) \\ &= \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta') = \mathsf{eval}_{\mathcal{X}}(\xi(v),\theta). \end{aligned}$$

Therefore $s = s' \in \gamma(\hat{s}')$. This shows $\gamma(\hat{s}) \subseteq \gamma(\hat{s}')$. The proof of the opposite inclusion is similar. □

Part (3): the proof is immediate from the definition of γ, equation (4).