VSL Preprint January 2012

Loop Invariant Symbolic Execution for Parallel Programs

Stephen F. Siegel and Timothy K. Zirkel
Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA

This paper has been published as:

S. F. Siegel and T. K. Zirkel. Loop Invariant Symbolic Execution for Parallel Programs. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 412–427, 2012.

Loop Invariant Symbolic Execution for Parallel Programs

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. Techniques for verifying program assertions using symbolic execution exhibit a significant limitation: they typically require that (small) bounds be imposed on the number of loop iterations. For sequential programs, there is a way to overcome this limitation using loop invariants. The basic idea is to assign new symbolic constants to the variables modified in the loop body, add the invariant to the path condition, and then explore two paths: one which executes the loop body and checks that the given invariant is inductive, the other which jumps to the location just after the loop. For parallel programs, the situation is more complicated: the invariant may relate the state of multiple processes, these processes may enter and exit the loop at different times, and they may be at different iteration counts at the same time. In this paper, we show how to overcome these obstacles. Specifically, we introduce the notion of *collective loop invariant* and a symbolic execution technique that uses it to verify assertions in message-passing parallel programs with unbounded loops, generalizing the sequential technique.

1 Introduction

1.1 Loop Invariant Symbolic Execution for Sequential Programs

Symbolic execution can be used to verify that assertions hold for all possible executions of a (sequential or parallel) program [9, 10]. The basic technique involves the exploration of a state-transition system in which *symbolic constants* $X_1, X_2, \ldots$ are used as inputs and/or initial values of variables. In each symbolic state, variables hold symbolic expressions in the X_i. A boolean-valued path condition variable pc is also included in the state. Control follows the usual program semantics, though at a branch on condition c, a nondeterministic choice is made, and pc is updated according to $\mathsf{pc} \leftarrow \mathsf{pc} \wedge c$ or $\mathsf{pc} \leftarrow \mathsf{pc} \wedge \neg c$, depending on the choice. The search is pruned whenever pc is seen to be unsatisfiable; automated theorem proving techniques are used for this purpose. When control reaches an assertion on expression e, the expression $\mathsf{pc} \Rightarrow e$ is evaluated to yield a boolean-valued symbolic expression, the validity of which is checked using the theorem

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

prover. If all reachable states are explored and all claims have been established, one has proved that the assertions can never be violated. We will refer to this technique as *standard symbolic execution*.

Unfortunately, because of loops, there are usually an infinite number of reachable states. This is true even for programs in which all loops terminate after a finite number of iterations for any input—e.g., a program which takes as input an integer n and has a loop that iterates from 1 to n. One way to deal with this is to place bounds on the number of loop iterations and/or the values of variables (such as n) which determine loop iterations (see, e.g., [2,13,15]). In this approach, one sacrifices soundness for tractability: if the technique concludes no violations are possible for executions in which each loop iterates no more than B times, it is still possible that an assertion may be violated for some execution where a loop iterates $B + 1$ times. Clearly, this is not satisfactory. Moreover, the number of states tends to blow up as the iteration bound increases, and in general there is no way to know how large the bound must be before one can conclude that the assertions hold for arbitrary numbers of iterations.

For sequential programs, one solution to this problem is a technique we call *loop invariant symbolic execution* (LISE) [1, 8, 14]. In this approach, we assume we are given a boolean expression e_l for each loop l. We also assume we are given a set of variables W_l which contains all variables that could be modified in the loop body. Additional symbolic constants are used to represent the values held by variables after an arbitrary number of loop iterations: call these $Y_{l,v}$, where $v \in W_l$. LISE explores a state-transition system which is similar to the system used in the standard technique, but modified in specific ways. When control reaches a loop location l, the validity of e_l is checked in the usual way. This establishes the base case for an inductive proof. Then, for each $v \in W_l$, v is assigned $Y_{l,v}$, discarding the old value for v. The expression e_l is then evaluated in this new environment and the result is added to the path condition, thereby encoding in the state the assumption that the invariant holds after an arbitrary number of executions of the loop body. Next, a nondeterministic choice is made between the *true* and *false* branches, as usual. For the *true* branch, control passes into the loop body and symbolic execution proceeds normally, except that the "back edges" which return control to the loop location are removed. In place of these is an assertion that again checks the validity of the invariant, establishing that e_l is inductive. Choosing the *false* branch moves control to the point just after the loop, from which the search proceeds normally.

Assuming there are a finite number of initial states, LISE, if applied to every loop in the sequential program, results in a transition system with a finite number of reachable states. (In the worst case, the number of such states is exponential in the size of the program.) The procedure is sound: if all of these states are explored and along the way each claim is proved, the result is a proof that the assertions (including the loop invariants) can never be violated on any execution of the program.

The limitations of the LISE approach are well-known. First, finding appropriate invariants is hard. Second, LISE can raise a "false alarm" for a number

of reasons: (1) the theorem prover fails to prove a valid claim, (2) a claim is not valid because a previous loop invariant was too weak, and (3) a claim is not valid because the user-supplied expression is not actually a loop invariant, i.e., it does not hold upon reaching the loop or is not inductive. Finally, the technique only establishes partial correctness—it does not show that each loop will iterate only a finite number of times. On the other hand, there has been significant progress in finding loop invariants automatically (e.g., [5, 14]). Automated theorem proving technology is also advancing steadily, and other techniques can be used to establish termination. All in all, LISE appears to be a promising approach for a very difficult problem.

1.2 Extending LISE to Parallel Programs: Challenges

The goal of this paper is to extend LISE to parallel programs. The main challenge concerns loops that span multiple processes, such as the `while` loop in Figure 1. The code is in C and is written in the typical "SPMD" style using the Message Passing Interface (MPI). Conceptually, each of the $n =$ `nprocs` processes executes its own copy of this code in its own address space, i.e., with no shared memory. Each process has a unique integer *rank* between 0 and $n-1$ (inclusive) and initializes its copies of `s` and `t` to $n(n-1)/2$. In each iteration of the loop, each process of positive rank sends its rank to the process of rank 0. Process 0 receives these messages in any order (using the *wildcard* `MPI_ANY_SOURCE`) and sums them, storing the result in `s`. The claimed invariant `s = t` holds trivially on all processes of positive rank (since neither variable is modified) and it seemingly holds on process 0, as $\sum_{i=1}^{n-1} i = n(n-1)/2$. However, the code contains a defect which can lead to a violation of the invariant on process 0: process 1, for example, could send its first message, race ahead to the next iteration and send its second message, and both messages could be received by process 0 in its first iteration. Hence any sound generalization of LISE to the parallel context must return a negative result (or at least, not return a positive result) on `race.c`.

```
int s = t = nprocs*(nprocs-1)/2;
#pragma TASS collective invariant I s == t;
while (true) {
  if (myrank == 0) {
    s = 0;
    for (j=1; j<nprocs; j++) {
      MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      s+=x;
    }
  } else MPI_Send(&myrank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

Fig. 1. `race.c`: the invariant may be violated when messages cross iteration boundaries. The pagma tells TASS to use LISE with the specified collective invariant when verifying that loop.

```
1    int n; /* input variable; assume n >= 0 */
2    int i = 0, x = 0;
3    #pragma TASS collective invariant I i==PROC[1-myrank]@main.i && \
         x==2*((i+1-myrank)/2);
4    while(i < 2*n) {
5      if (myrank == 0 && i%2 == 0) {
6        MPI_Recv(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7      } else if (myrank == 1 && i%2 == 1) {
8        x = i+1;
9        MPI_Send(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
10     }
11     i++;
12   }
13   assert(x==2*n);
```

Fig. 2. `stagger.c`: a 2-process program in which messages sent on odd iterations are received in even iterations. The program is correct: the invariant and assertion hold on all executions. A synchronization at the loop location would cause deadlock. The notation `PROC[1-myrank]@main.i` references the variable `i` in the function `main` of process with rank `1-myrank`.

If processes were required to synchronize at loop locations, a straightforward generalization of LISE might be possible. But as the example above shows, this is not the case: processes may enter or exit the loop at very different times and may be at different iterations at the same time. A *loop-synchronizing* technique—one which considers only the subset of executions in which processes synchronize at loop locations—cannot be sound: in `race.c`, for example, the invariant holds on all such executions, but, as we have seen, fails on other executions.

Figure 2 is an example of the dual problem: the code is correct, but imposing synchronization at the loop location would cause deadlock. This 2-process program takes as input a nonnegative integer n and iterates from 0 to $2n-1$. On odd iterations, process 1 sends a message to process 0, and on even iterations, process 0 receives. Note that the message is received in the iteration immediately preceding the one in which it is sent. This is not paradoxical; it simply requires process 1 to keep an iteration ahead of process 0. Again, a useful generalization of LISE should be able to verify the final assertion in this code.

In this paper, we present a sound generalization of LISE to parallel message-passing programs with a given, fixed number of processes (i.e., we are not dealing with the much harder problem of parametrized verification, which entails verifying correctness for all values of `nprocs`). Our approach builds on the notion of *collective assertions* [19]. A collective assertion is associated to locations in several processes and may refer to variables in all these processes. As control in a process passes through one of these locations, a snapshot of the process state is taken and inserted into a queue. Once there is a snapshot from every process, one snapshot from each process is dequeued and these snapshots are composed to form a global state at which the assertion is evaluated and checked. Violations are reported if an assertion fails, if any queue is nonempty at termination, or if assertions are not encountered in the same order in every process.

Our generalization of LISE uses *collective loop invariants*, special collective assertions that play the role of ordinary loop invariants in sequential LISE. We have implemented the technique in the *Toolkit for Accurate Scientific Software* (TASS, [21, 22]), a symbolic execution-based verification framework for C/MPI programs. In addition to verifying assertions in a single program, TASS can verify the functional equivalence of two programs, and the LISE technique enables the equivalence verification to work with unbounded loops too.

2 Multiprocess Loop Invariant Symbolic Execution

2.1 Notation and Formal Framework

We use the notation and framework of [21]. 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. X^* denotes the set of all finite sequences of elements of X. For $\xi \in X^*$ and $x \in X$, $\mathit{enqueue}(x, \xi) \in X^*$ denotes the sequence resulting from appending x to the end of ξ. For non-empty ξ, $\mathit{first}(\xi) \in X$ denotes the first element of ξ and $\mathit{dequeue}(\xi) \in X^*$ denotes the result of removing that element from ξ. Similarly, $\mathit{push}(x, \xi) \in X^*$ denotes the result of appending x to ξ, $\mathit{top}(\xi)$ denotes the last element, and $\mathit{pop}(\xi) \in X^*$ denotes the result of removing the last element.

We may give names $n_1, \ldots, n_r$ to the components of a Cartesian product of sets $X = X_1 \times \cdots \times X_r$. Given $x \in X$, $x.n_i$ denotes the i^{th} component of x, and $x[n_i : v]$ is the element of X identical to x except at the component named n_i, where the value is v.

Let $\mathbb{B} \stackrel{\text{def}}{=} \{\mathit{true}, \mathit{false}\}$, $\mathsf{Val} \supseteq \mathbb{B}$ 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 $\mathsf{eval}_V : \mathsf{Expr}(V) \times \mathsf{Eval}(V) \to \mathsf{Val}$. 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* 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. $\mathsf{effect} : \mathsf{Act} \times \mathsf{Eval}(V) \to \mathsf{Eval}(V)$ 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*.

Let SExpr denote the set of all symbolic expressions and $\mathsf{SBoolExpr}$ the set of boolean-valued symbolic expressions. A symbolic state includes the path condition (an element of $\mathsf{SBoolExpr}$), a location, and a function assigning a symbolic expression to each variable. Let $\mathsf{valid} : \mathsf{SBoolExpr} \to \mathbb{B}$ be a function with the property that for any ϕ, if $\mathsf{valid}(\phi) = \mathit{true}$ then ϕ is valid, i.e., any assignment of concrete values to the symbolic constants occurring in ϕ causes ϕ to evaluate to *true*. This function models a conservative theorem prover. Let

$\mathsf{seval}\colon \mathsf{Expr}(V) \times \mathsf{Func}(V, \mathsf{SExpr}) \to \mathsf{SExpr}$ denote the symbolic evaluation function.

A parallel program has a fixed number $n \geq 1$ of processes and is modeled as follows. The processes are identified with the set $\mathsf{Proc} = \{0, \ldots, n-1\}$. Each process p has a set of local variables V_p. In addition there is a set of shared variables V_{sh}. Shared variables may be used to represent the channels (buffered messages) when modeling a message-passing program. Process p is modeled as a program graph over $V_p \cup V_{\mathsf{sh}}$. We assume the location sets from two different processes are disjoint, as are the action sets.

The global program graph PG is the program graph over $V = V_{\mathsf{sh}} \cup \bigcup_{p=0}^{n-1} V_p$ defined as follows: the (initial) global location set is the Cartesian product of the local (initial) location sets; the initial condition is the conjunction of the local initial conditions; the global action set is the union of the local action sets; the effect function on an action in process p is extended to act trivially on variables that are not in V_p or V_{sh}; and each local transition $\langle l_p, g, \alpha, l'_p \rangle$ in process p introduces the set of all global transitions $\langle (l_0, \ldots, l_p, \ldots, l_{n-1}), g, \alpha, (l_0, \ldots, l'_p, \ldots, l_{n-1}) \rangle$ where l_j is any location in process j, for $j \in \mathsf{Proc} \setminus \{p\}$. Given any global transition t, define $\mathsf{proc}(t)$ to be the unique $p \in \mathsf{Proc}$ such that the action component of t is in Act_p. This is the standard interleaving model for a multiprocess program: each execution step consists of a single step in one process, while all other processes remain put.

2.2 Collective Assertions and Loop Invariants

We assume the program is annotated with *collective assertions*, which are identified with symbols in a set Id. Each $\mathsf{id} \in \mathsf{Id}$ determines (1) a nonempty set of processes $\mathsf{procs}(\mathsf{id}) \subseteq \mathsf{Proc}$, (2) a set of locations $\mathsf{dom}(\mathsf{id}) \subseteq \bigcup_{p \in \mathsf{procs}(\mathsf{id})} \mathsf{Loc}_p$, and (3) a function $\mathsf{assertion}_{\mathsf{id}}\colon \mathsf{dom}(\mathsf{id}) \to \mathtt{BoolExpr}(\bigcup_{p \in \mathsf{procs}(\mathsf{id})} V_p)$. Note that the expression associated to a location in process p may refer to variables in other processes, but not to shared variables. Without loss of generality, assume that every location is involved in at most one collective assertion. (No-ops may be inserted as necessary to meet this requirement.)

LISE requires some additional structure in the program graph for a process p. Specifically, we assume that certain locations in the program graph are designated as *LISE loop locations*. These have exactly two outgoing transitions: the *true* and *false* branches. Each acts as a no-op, i.e., the action associated to each is trivial. If the guard for the *true* branch is g, the guard for the *false* branch is $\neg g$. We further assume that g does not involve any shared variables. The set of LISE loop locations is denoted $\mathsf{LoopLoc}_p$, and the set of transitions emanating from them LTran_p. The function $\mathsf{isTrue}\colon \mathsf{LTran}_p \to \mathbb{B}$ tells whether a loop transition corresponds to the *true* or *false* branch.

The soundness of the multiprocess LISE technique does not require any further structural assumptions: if the technique 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"

loops, without any jumps into or out of the loop bodies, and the loop locations correspond in the usual way to those loop statements. In a program graph generated in this way, there are certain restrictions on the sequence of loop operations that can occur along a control path through a process. Consider, for example, a process with two LISE loop locations, where t_i indicates the *true* branch for loop i and f_i the *false* branch. The sequence t_1, t_1, t_2, f_2, f_1 is possible: it can arise if the second loop is nested inside the first, and on some execution path the first loop is entered and then re-entered (without executing the second loop), while on the second iteration the second loop is executed once. A sequence such as t_1, t_2, f_1, on the other hand, cannot occur: the second loop would have to exit before the first could exit.

A *collective loop invariant* is a collective assertion id for which the elements of $\mathsf{dom}(\mathsf{id})$ are LISE loop locations. We assume every LISE loop location l_p participates in exactly one collective loop invariant $\mathsf{id}(l_p)$. This does not mean every loop in the program is required to participate in a collective invariant since there is no requirement that every node arising from a source-code loop be included in $\mathsf{LoopLoc}_p$.

2.3 Multiprocess LISE State

For $p \in \mathsf{Proc}$, let $\mathsf{ProcState}_p \stackrel{\text{def}}{=} \mathsf{Loc}_p \times \mathsf{Func}(V_p, \mathsf{Val})$. Call the first component $\mathsf{location}$ and the second eval. A *LISE record* is a structure used to record information about the state of a collective action. The components of a LISE record are as follows:

1. $\mathsf{id} \in \mathsf{Id}$, a symbol uniquely identifying a collective action,
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 process state,
3. $\mathsf{writeset}$, a function which associates to each $p \in \mathsf{procs}(\mathsf{id})$ a subset of V_p, the current estimate of the set of local variables modified in the loop body,
4. $\mathsf{isTrue} \in \mathbb{B}$, is this record for the execution of the *true* branch of a loop?,
5. $\mathsf{ppc} \in \mathsf{SBoolExpr}$, assumptions made during this loop iteration, and
6. $\mathsf{relation} \in \mathsf{SBoolExpr}$, a predicate relating new symbolic constants to values from the previous iteration.

The set of LISE records is denoted $\mathsf{LRecord}$. Components 1 and 2 are used for all collective actions. Components 3–6 are relevant only for loop actions; for non-loop actions, components 5 and 6 are *true*.

A *LISE state* is a structure with the following components:

1. $\mathsf{pc}_0 \in \mathsf{SBoolExpr}$, the permanent component of the path condition, used to record assumptions made when control is not inside a loop,
2. $\mathsf{location} \in \mathsf{Loc} = \mathsf{Loc}_0 \times \cdots \times \mathsf{Loc}_{n-1}$,
3. $\mathsf{eval} \in \mathsf{Func}(V, \mathsf{SExpr})$, a function giving the symbolic value of each variable,
4. $\mathsf{queue} \in \mathsf{LRecord}^*$, a FIFO queue of incomplete LISE records, and
5. $\mathsf{stack} \in \mathsf{LRecord}^*$, a stack of complete, *true*-branch, loop records.

The set of all LISE states is denoted LState. In initial LISE states, the queue and stack are empty. For $s \in \mathsf{LState}$, the *path condition* associated to s is

$$\mathsf{pc}(s) \stackrel{\text{def}}{=} s.\mathsf{pc}_0 \wedge \bigwedge_{r \in s.\mathsf{stack} \cup s.\mathsf{queue}} r.\mathsf{ppc} \wedge r.\mathsf{relation}.$$

This plays the same role as the path condition in standard symbolic execution: the search is pruned whenever $\mathsf{pc}(s)$ is determined to be unsatisfiable, and $\mathsf{pc}(s)$ is used as the assumption when checking assertions.

The writeset is used to determine the set of variables modified in the loop body. This is done dynamically, during exploration of the state space. The set is empty when control first reaches the loop, and a variable is only added to the set when it is actually modified. This allows a precise estimate of the set, especially in the presence of dynamically allocated memory and pointers. The set can grow with each loop iteration, but in most cases will converge after two or three iterations.

The queue stores information on incomplete collective actions: those for which at least one process involved in the action has reached that point, but at least one process has not. The stack is used for completed collective loop entry actions only. These are records for which all processes have entered the loop, but at least one has not yet exited the loop. If we think of the current location of the collective program as the location of the slowest process, the stack represents the loop nest the collective program is currently in.

A record is created when a process reaches a location for a collective assertion and is the first process to do so. The new record is enqueued at the end of the FIFO queue. It is dequeued once every process has reached the assertion. If it is a record for entering a collective loop, once the record is dequeued it is pushed onto the stack. It is eventually popped from the stack once every process has exited that loop. Hence at any time, the record for the oldest collective action is at the bottom of the stack, the actions become more recent as one moves up the stack, then to the beginning of the queue, and through to the end of the queue, which contains the most recent record.

The queue may have to record multiple consecutive iterations of the loop, due to the possible iteration lag between processes. The relation predicate is used to maintain a coherent relation between these successive iterations. It is discarded as soon as the record is complete and moved to the stack. This issue does not arise in sequential LISE, since no lag is possible.

It will be necessary to locate the records for the nest of loops a process p is inside in a given state s. This is computed using a stack T, initially empty. The algorithm iterates over the records r occurring in $s.\mathsf{stack}$ and $s.\mathsf{queue}$ from oldest to newest. For each such r, if $r.\mathsf{id}$ is a collective loop invariant and $r.\mathsf{snapshots}(p) \neq \mathsf{null}$, the following two steps are taken in order: (1) if T is nonempty and the top entry of T is a record for $r.\mathsf{id}$, pop T; (2) if $r.\mathsf{isTrue}$ is *true*, push r onto T. The resulting *loop nest stack* will contain the desired records, with the innermost loop at the top. We define $\mathsf{currentLoopRecord}(p, s)$ to be the top entry in this stack, or null if the stack is empty.

2.4 The Multiprocess LISE Next-State Function

We now describe the next-state function on LState. Let $s \in \mathsf{LState}$ and $t \in \mathsf{Tran}$ be a transition in process p. If $t \notin \mathsf{LTran}$, the update to the state proceeds as in standard symbolic execution, with the following exceptions: (1) a predicate that is to be added to the path condition is added instead to the ppc field for $\mathsf{currentLoopRecord}(p, s)$ (or to $s.\mathsf{pc}_0$ if that record is null), and (2) any variable modified is added to the sets $r.\mathsf{writeset}(p)$ for each r in the loop nest stack for p.

If the new location is annotated by a non-loop collective assertion, the usual collective assertion verification actions are taken: a snapshot of the local state is stored in the queue, and if the snapshot completes a record, the record is dequeued and the assertion checked. As these actions are a subset of those taken when executing a loop transition, we now turn to the more general situation.

So assume $t \in \mathsf{LTran}$. The next-state function is defined in Figure 3. The first task is to determine whether t represents an initial entry into the loop (from outside the loop body) or a re-entry, i.e., a return to the loop location after executing the loop body one or more times. This is determined by examining the current loop record r_0 for p: if r_0 is non-null and its identifier is the same as that of t, this is a re-entry, in which case W is assigned the writeset from the previous iteration; otherwise, W is assigned the empty set.

Next, we determine whether a record for this collective action already exists by checking if there is any record in the queue with a null snapshot for p. If there is, the oldest such record r_1 is selected and we check that the identifier of that record agrees with that of t. (If it does not agree, then p has encountered the collective assertions in a different order than another process, and an error is reported.) We further check that the isTrue flag of t agrees with that of r_1; if this fails then one process has exited the loop while another entered the loop on corresponding iterations, and an error is reported. If instead, p is the first to reach this collective action, a new record is instantiated with all snapshots null, writesets empty, and the ppc and relation predicates *true*.

The next step is the assignment of fresh symbolic constants to variables in W. The state is scanned to find the least m such that no Y_i occurs in the state for $i > m$. The new valuation eval' is equivalent to the old one, except that each variable in W has been assigned one of the Y_i. The guard for t is then re-evaluated using eval' and added to $r_1.\mathsf{ppc}$ to form the new ppc ϕ. The expression equating each new symbolic constant to the old value of the corresponding variable is added to the relational predicate to form the new relational predicate ψ. These predicates are used to create a new record r_2, which also incorporates the snapshot of the state of p. This new record either replaces the old one in the queue, or is inserted at the end of the queue. The new state s' is the same as s, except with the new location, valuation eval', and the modified queue.

We next deal with the case when the previous action completed the record r_3 at the front of the queue. If there is any snapshot still null in r_3, the record remains incomplete, and s' is returned. Otherwise, the snapshot valuations are united to form a single valuation which is used to evaluate the invariant expres-

```
1  procedure execLoop(s : LState, t : LTran) : LState is
2  | let t = ⟨l, guard, α_0, l'⟩; p ← proc(t); r_0 ← currentLoopRecord(p, s);
3  | if r_0 ≠ null ∧ r_0.id = id(l_p) then W ← r_0.writeset; else W ← ∅;
4  | if ∃i: s.queue[i].snapshots(p) = null then
5  | | let i_1 be the least such i; r_1 ← s.queue[i_1]; isNew ← false;
6  | | if r_1.id ≠ id(l_p) then error("out of order");
7  | | if r_1.isTrue ≠ isTrue(t) then error("conflicting loop exit");
8  | else isNew ← true; r_1 ← ⟨id(l_p), λq.null, λq.∅, isTrue(t), true, true⟩;
9  | let W = {v_1, ..., v_k};
10 | if ∃i: Y_i occurs in s then m ← the maximum such i; else m ← 0;
11 | eval' ← s.eval[v_1 : Y_{m+1}] ··· [v_k : Y_{m+k}];
12 | φ ← r_1.ppc ∧ seval(guard, eval');
13 | ψ ← r_1.relation ∧ s.eval(v_1) = Y_{m+1} ∧ ··· ∧ s.eval(v_k) = Y_{m+k};
14 | r_2 ← ⟨id(l_p), r_1.snapshots[p : ⟨l_p, eval'|_{V_p}⟩], r_1.writeset[p : W], isTrue(t), φ, ψ⟩;
15 | if isNew then queue' ← enqueue(s.queue, r_2); else queue' ← s.queue[i_1 : r_2];
16 | s' ← ⟨s.pc_0, l', eval', queue', s.stack⟩;
17 | r_3 ← first(queue');
18 | if ∃q ∈ procs(r_3.id): r_3.snapshots(q) = null then return s';
19 | ξ ← ⋃_{q∈procs(r_3.id)} r_3.snapshots(q).eval;
20 | claim ← seval(⋀_{q∈procs(r_3.id)} assertion_{r_3.id}(r_3.snapshots(q).location), ξ);
21 | if ¬valid(pc(s') ⇒ claim) then error("Possible invariant violation");
22 | r_4 ← r_3[ppc : r_3.ppc ∧ claim][relation ← true];
23 | queue' ← dequeue(queue');
24 | stack' ← s.stack;
25 | if ¬empty(stack') ∧ top(stack').id = r_4.id then stack' ← pop(stack');
26 | φ ← s.pc_0;
27 | if r_4.isTrue then stack' ← push(stack', r_4);
28 | else if empty(stack') then φ ← φ ∧ r_4.ppc;
29 | else
30 | | r_5 ← top(stack');
31 | | stack' ← push(pop(stack'), r_5[ppc : r_5.ppc ∧ r_4.ppc]);
32 | return canonic(⟨φ, l', eval', queue', stack'⟩);
```

Fig. 3. The next-state function for loop transitions in a multiprocess program.

sions and form the claim. The validity of claim (under the assumption of the path condition) is then checked using the theorem prover.

A new record r_4 is now formed from r_3 by adding claim to the ppc field and erasing the relational predicate. (The claim is no longer necessarily implied by the path condition, since the relational predicate has been removed.) The first record is now removed from the queue.

If the top entry on the stack is for a collective loop that matches the one in the record just dequeued, the stack is popped, "forgetting" all information from the previous iteration. There are now two cases: either r_4 represents a loop exit (*false* branch), or (re-)entry (*true* branch). If an entry, r_4 is pushed onto the stack. If an exit, r_4 is not inserted into the state, but the ppc field of r_4 is

recorded by adding it to the record r_5 now on the top of the stack (or to pc_0 if the stack is empty). As r_4 represents a *false* branch, $r_4.\mathsf{ppc}$ does not record any assumptions other than those arising from (1) the guards for the *false* loop branches from each process, and (2) the collective invariant claim. Hence, these are the only facts that are "remembered" when all processes exit the loop.

The function $\mathsf{canonic}$, invoked before returning the new state, 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 $\mathsf{canonic}$ might replace Y_2 with Y_1, Y_3 with Y_2, and Y_7 with Y_3, everywhere those symbols 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. This step is crucial for convergence in many cases.

2.5 Example

Consider the execution prefix of `stagger.c` (Figure 2) in which process 0 runs until it blocks at the receive, then process 1 runs until completing the send, then process 0 receives the message and proceeds to the top of the loop. Let $\mathtt{x}_0$ denote the copy of variable `x` in process 0, etc., use line numbers for locations, and ordered pairs for the values of a function at process 0 and 1. Then the state s arrived at has the form

$$\begin{aligned} s &= \langle X_1 \geq 0, \langle 4, 11\rangle, \{\mathtt{n}\colon X_1, \mathtt{i}_0\colon 1, \mathtt{x}_0\colon Y_1 + 1, \mathtt{i}_1\colon Y_1, \mathtt{x}_1\colon Y_1 + 1\}, r_1, r_0\rangle \\ r_1 &= [I,\ (\mathsf{null}, \langle 4, \{\mathtt{i}_1\colon Y_1, \mathtt{x}_1\colon 0\}\rangle),\ (\emptyset, \{\mathtt{i}_1, \mathtt{x}_1\}),\ \mathit{true},\ Y_1 < 2X_1,\ Y_1 = 1] \\ r_0 &= [I, \ldots,\ (\{\mathtt{i}_0, \mathtt{x}_0\}, \{\mathtt{i}_1, \mathtt{x}_1\}),\ \mathit{true},\ 0 < 2X_1,\ \mathit{true}]. \end{aligned}$$

Let us see what happens when process 0 executes the *true* loop branch from this state. The current loop record for process 0 is r_0, so $W = \{\mathtt{i}_0, \mathtt{x}_0\}$. We have $m = 1$, so $\mathtt{i}_0$ is assigned Y_2 and $\mathtt{x}_0$ Y_3, and

$$\begin{aligned} r_2 = r_3 = [I,\ (\langle 4, \{\mathtt{i}_0\colon Y_2, \mathtt{x}_0\colon Y_3\}\rangle, \langle 4, \{\mathtt{i}_1\colon Y_1, \mathtt{x}_1\colon 0\}\rangle),\ (\{\mathtt{i}_0, \mathtt{x}_0\}, \{\mathtt{i}_1, \mathtt{x}_1\}),\ \mathit{true}, \\ Y_1 < 2X_1 \wedge Y_2 < 2X_1,\ Y_1 = 1 \wedge Y_2 = 1 \wedge Y_3 = Y_1 + 1]. \end{aligned}$$

As this results in completing the first entry in the queue, we proceed to check that the path condition implies $\mathtt{i}_0 = \mathtt{i}_1 \wedge \mathtt{x}_0 = 2((\mathtt{i}_0 + 1) \div 2) \wedge \mathtt{x}_1 = 2(\mathtt{i}_1 \div 2)$ (where $\div$ denotes integer division), which reduces to checking $1 = 1 \wedge 2 = 2 \wedge 0 = 0$. The new record is

$$\begin{aligned} r_4 = [I,\ (\langle 4, \{\mathtt{i}_0\colon Y_2, \mathtt{x}_0\colon Y_3\}\rangle, \langle 4, \{\mathtt{i}_1\colon Y_1, \mathtt{x}_1\colon 0\}\rangle),\ (\{\mathtt{i}_0, \mathtt{x}_0\}, \{\mathtt{i}_1, \mathtt{x}_1\}),\ \mathit{true}, \\ Y_1 < 2X_1 \wedge Y_2 < 2X_1 \wedge Y_1 = Y_2 \wedge Y_3 = 2((Y_2 + 1) \div 2) \wedge 0 = 2(Y_1 \div 2),\ \mathit{true}] \end{aligned}$$

and the new state is $\langle X_1 \geq 0, \langle 5, 11\rangle, \{\mathtt{n}\colon X_1, \mathtt{i}_0\colon Y_2, \mathtt{x}_0\colon Y_3, \mathtt{i}_1\colon Y_1, \mathtt{x}_1\colon Y_1 + 1\}, \epsilon, r_4\rangle$, where ϵ is the empty sequence. Note how the original state s was still tied to the

initial iteration of the loop, but those ties have been dropped in the new state. In two more iterations the essential inductive step will take place and the path will return to a state seen before.

2.6 Soundness

Multiprocess LISE is sound for very general reasons. A more formal proof is given in [20], but the ideas are simple. The LISE transition system is related to that of standard symbolic execution via the *projection map* ρ: $\mathsf{LState} \to \mathsf{SState}$, defined by $\rho(s) = \langle \mathsf{pc}(s), s.\mathsf{location}, s.\mathsf{eval} \rangle$. One must show that for any transition and $s \in \mathsf{LState}$, the projection of the image of s under the LISE next-state function subsumes the image of $\rho(s)$ under the standard next-state function. The soundness of LISE then follows from that of the standard technique.

While the LISE next-state function is complicated, its image under ρ is easy to understand. Control flow is exactly the same as in the standard technique (unlike sequential LISE, no back edges are removed). The differences are due to three types of transformations on SState: (1) replace every occurrence of symbolic constant X by symbolic constant Y, where Y does not occur in the original state; (2) if v holds value f, assign a new symbolic constant Y to v and add the constraint $Y = f$ to the path condition, and (3) weaken the path condition (by dropping some clauses from the conjunction). The first two result in equivalent states, the third in a state which subsumes the original.

3 Implementation and Experiments

We implemented multiprocess LISE by extending the TASS collective assertion facility. The invariants are encoded as pragmas immediately preceding a *while* or *for* loop. The syntax is the same as that for collective assertions (described in [19]) except that the keyword `invariant` is used in place of `assert`.

TASS supports various MPI-specific partial order reduction (POR) schemes. Used in conjunction with LISE, these are often key to obtaining convergence. Note that a necessary condition for convergence is that the maximum difference in iteration count between any two processes be bounded. In the *full* state space for `stagger.c` (Figure 2), this condition does not hold, since it is possible for all the messages from process 1 to be buffered. In the reduced space considered by the default POR, however, the send and receive always occur synchronously, and the two processes can get at most 2 iterations apart. (The POR theory guarantees that if an assertion violation occurs in the full space then a violation also occurs in the reduced space, so the reduction is sound.) By applying LISE to the reduced space, TASS reduces the problem to a finite number of states.

TASS uses *comparative symbolic execution* [18] to verify the functional equivalence of two programs. In its original form, this entails forming the sequential composition of the two programs and adding an assertion that the outputs from the two programs agree. We have modified this technique to use the parallel, instead of sequential, composition. If the first program has n processes, and the

```
#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;
}
```

Fig. 4. Fibonacci. Two functionally equivalent programs to compute ϕ. The program on the left is the specification. The notation `spec.k` indicates the variable `k` in the specification program.

second m, the composite has $n + m$ processes and the two sets of processes just happen to never interact. With this modification, multiprocess LISE can be applied to the composite program. This is effective when the two programs have corresponding loops that can be related with a collective invariant, such as the programs of Figure 4. (Note "`joint`" is used for a collective invariant spanning multiple programs). In such cases, the POR will effectively keep the two programs as close together as possible in their iteration counts, enabling convergence in many cases.

We applied multiprocess LISE to 9 examples. All (except `race`) take an integer input N and the goal is to verify a property for all N. The first two examples are discussed in Section 1; the others are (3) `matrix`, a sequential program for adding two $N \times N$ matrices in which we assert functional correctness, (4) `count`, a multiprocess program where each process loops from 1 to N and we check the loop variable equals N at termination, (5) `ring`, an MPI program where processes send right and receive left N times and we verify deadlock-freedom, (6) `mean`, two sequential programs computing the mean of an array of doubles, (7) `fib` (Figure 4), two sequential programs computing the limiting ratio of two consecutive terms in the Fibonacci sequence to within a given tolerance, (8) `nested`, two sequential programs, one computing $\sum_{i=1}^{n} \sum_{j=1}^{i^2} ij$, the other $\sum_{i=1}^{n} i \sum_{j=1}^{i^2} j$, (9) `diffusion`, two programs solving the 1d-diffusion equation, one sequential, one using MPI. In the last four, functional equivalence was verified.

In all cases except `race` (which contains a defect), we were able to formulate invariants enabling verification for all N. Performance data is given Fig. 5; all experimental artifacts can be obtained from [22]. For the most challenging example, `diffusion`, we also compare the performance of LISE with the standard technique using various upper bounds on N. Two facts stands out: (1) the number of transitions explored by LISE is usually comparable to that number for a very small bound, but (2) the time consumed per transition in LISE is very large. Inspection reveals that most of this time is due to the increased number and complexity of the theorem prover calls.

4 Conclusion

Related work. Related symbolic execution-based approaches, in addition to those discussed in Section 1, include [16], which presents a technique in which multi-

ple control paths are combined and symbolically executed simultaneously. This eliminates the problem of loops in certain circumstances by combining multiple iterations into a single execution. However, some types of cyclic dependencies still require bounding the number of loop iterations, and it is not clear how to generalize the technique to parallel programs.

There have been other approaches to functional equivalence verification, but they tend to come with strong restrictions. For example, [24] presents a technique for verifying functional equivalence of affine programs with static control flow. It is fully automatic, requiring no invariants or other hints from the user, but 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 multiprocess programs. Peggy [23] represents functions as *program expression graphs* and attempts to show two sequential functions are equivalent by transforming the graph of one into another using a library of axioms. *Translation validation* tools, such as TVOC [6], check that certain compiler transformations, including complex loop transformations, preserve equivalence. But this approach typically requires additional information from the compiler when dealing with loop transformations, and it is not clear if it could be extended to verify the equivalence of, say, a parallel program constructed by hand from a very different sequential version, such as our `diffusion` example.

Another family of verification approaches that can handle unbounded loops works by generating verification conditions to be discharged by a theorem prover. VCC [4] allows users to annotate loops with invariants, and supports multithreaded programs. It includes a notion of one- and two-state object invariants that cut across threads, but there does not appear to be a straightforward way to extend this notion to relate loop executions across different threads.

CBMC [3] is a bounded model checker for C programs. It handles loops by unrolling them a finite number of times, and so cannot verify the correctness of programs with unbounded loops (but may find bugs in them).

name	nprocs	trans	proofs	time
race*	10	823	1	0.9
stagger	2	201	51	2.1
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

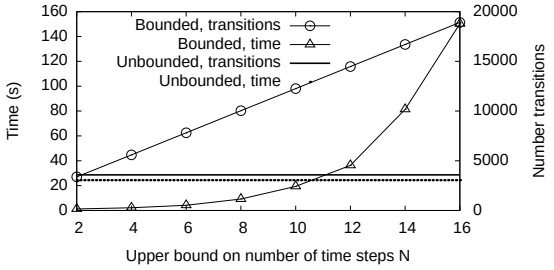

Fig. 5. Experimental results. Left: work required to verify (*=refute, stopping at first violation) using multiprocess LISE: number of processes (two numbers for equivalence verification), transitions, calls to theorem prover CVC3, and time (seconds). Right: verifying `diffusion` equivalence using upper bounds vs. using LISE to verify for all N.

Loop-extended symbolic execution [17] reasons about all possible executions of a loop by introducing auxiliary variables to represent the number of times the loop has been executed. The loop body is then analyzed to find any linear relationships between program variables and the auxiliary variables. To the best of our knowledge, this approach has not been applied to concurrent programs.

Poirot [11] uses Corral [12] to analyze concurrent C and .NET programs by converting them to sequential programs. The conversion involves bounding the number of possible context switches. Loops are then replaced with recursive procedure calls. Corral checks for bugs by iteratively increasing the maximum recursion depth. At a given recursion bound, if a bug is found it will be reported and the process terminates. Otherwise, the recursion bound is incremented and the process repeats until a timeout is reached.

Future work. The main challenges now are to find ways to discover collective loop invariants automatically and to reduce the number and cost of theorem-prover invocations. Another limitation is that in the current framework, the collective invariants cannot reference the shared part of the state, so there is no way to express, for example, that the number of buffered messages is invariant. It will be interesting to see if the technique can be extended to express such properties.

Acknowledgment. We are grateful to the reviewers of CAV 2011 and VMCAI 2012 for their comments and suggestions.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE '05. pp. 82–87. ACM, New York (2005)
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08 (2008)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS '04. pp. 168–176. Springer (2004)
4. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOL '09, LNCS, vol. 5674, pp. 23–42. Springer (2009)
5. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: ISSTA '11. ACM, New York (2011)
6. Goldberg, B.: Translation validation of loop optimizations and software pipelining in the TVOC framework - in memory of Amir Pnueli. In: Cousot, R., Martel, M. (eds.) SAS. LNCS, vol. 6337, pp. 6–21. Springer (2010)
7. Gopalakrishnan, G., Qadeer, S. (eds.): CAV 2011, LNCS, vol. 6806. Springer (2011)
8. Hantler, S.L., King, J.C.: An introduction to proving the correctness of programs. ACM Comput. Surv. 8, 331–353 (September 1976)
9. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer (2003)
10. King, J.C.: Symbolic execution and program testing. Comm. ACM 19(7), 385–394 (1976)

11. Lahiri, S., et al.: Poirot: The concurrency sleuth. http://research.microsoft.com/en-us/projects/poirot (2011)
12. Lal, A., Qadeer, S., Lahiri, S.: Corral: A whole-program analyzer for Boogie. Tech. Rep. MSR-TR-2011-60, Microsoft Research (May 2011)
13. Păsăreanu, C., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: ASE '10. ACM, New York (2010)
14. Păsăreanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In: Model Checking Software: 11th Intl. SPIN Workshop. LNCS, vol. 2989, pp. 164–181. Springer (2004)
15. Ramos, D., Engler, D.: Practical, low-effort equivalence verification of real code. In: Gopalakrishnan and Qadeer [7], pp. 669–685
16. Santelices, R., Harrold, M.J.: Exploiting program dependencies for scalable multiple-path symbolic execution. In: ISSTA '10. ACM, New York (2010)
17. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: ISSTA '09. pp. 225–236. ACM, New York (2009), http://doi.acm.org/10.1145/1572272.1572299
18. 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), Article 10, 1–34 (2008)
19. Siegel, S.F., Zirkel, T.K.: Collective assertions. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 387–402 (2011)
20. Siegel, S.F., Zirkel, T.K.: Symbolic execution for sequential and multi-process programs with unbounded loops. Tech. Rep. UD-CIS-2011/03, Univ. Delaware (2011)
21. Siegel, S.F., Zirkel, T.K.: TASS: The Toolkit for Accurate Scientific Software. Mathematics in Computer Science 5(4), 395–426 (2011)
22. Siegel, S.F., et al.: The Toolkit for Accurate Scientific Software. http://vsl.cis.udel.edu/tass (2011)
23. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: Gopalakrishnan and Qadeer [7], pp. 737–742
24. 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. LNCS, vol. 5643, pp. 599–613. Springer (2009)

A Symbolic Execution Background

A.1 Program Graphs

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* $\mathsf{next} \colon \mathsf{State} \times \mathsf{Tran} \to \wp(\mathsf{State})$ 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 $1 \leq i \leq n$.

A.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. [18]). 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 $\mathsf{snext} \colon \mathsf{SState} \times \mathsf{Tran} \to \wp(\mathsf{SState})$ 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.

A.3 Relations Between Concrete and Symbolic Semantics

Concretization. Each symbolic state $\hat{s}$ determines a set of concrete states $\gamma(\hat{s})$, where $\gamma \colon \mathsf{SState} \to \wp(\mathsf{State})$ 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 $\hat{\gamma} \colon \wp(\mathsf{SState}) \to \wp(\mathsf{State})$ by $\hat{\gamma}(S) = \bigcup_{\hat{s} \in S} \gamma(\hat{s})$. The following is proved in [21]:

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 $\exists \hat{s} \in \mathsf{SReach}.\hat{\pi}(\hat{s})$.*

B Proof of Soundness

In this section we establish the soundness of multiprocess LISE. 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 $\pi(s) = \mathit{true}$, then there exists $\tilde{s} \in \mathsf{LReach}$ such that $\pi(\rho(\tilde{s})) = \mathit{true}$.*

We begin with a lemma concerning transformations of symbolic states.

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 $\hat{s}$.*

Proof. 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,

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

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). □

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 pc_0 component or to the ppc 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 $\tilde{s}$. (The order is not the same as that of Figure 3, but the final effect is the same.) At each step, we keep track of what happens to the projection:

1. $\tilde{s}_1$ results from $\tilde{s}$ by adding the guard 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 possibly creating and enqueueing a new record; as the ppc and $\mathsf{relation}$ fields are *true* in the new record, $\rho(\tilde{s}_2) \sim \rho(\tilde{s}_1)$,
3. $\tilde{s}_3$ is obtained from $\tilde{s}_2$ by adding a snapshot and modifying a $\mathsf{writeset}$ field; as these have no effect on the projection, $\rho(\tilde{s}_3) = \rho(\tilde{s}_2)$,

4. $\tilde{s}_4$ results from $\tilde{s}_3$ by assigning a new symbolic constant to each variable in W and adding the corresponding relational predicate to one of the path condition variables; by Lemma 1(1), $\rho(\tilde{s}_4) \sim \rho(\tilde{s}_3)$,
5. $\tilde{s}_5$ results from $\tilde{s}_4$ by adding claim to one of the path condition variables; since claim is already implied by the other path condition variables in $\tilde{s}_4$, $\rho(\tilde{s}_5) \sim \rho(\tilde{s}_4)$,
6. $\tilde{s}_6$ is obtained by removing the relational predicate from whichever path condition variable it was added in the step above; by Lemma 1(3), $\rho(\tilde{s}_6)$ subsumes $\rho(\tilde{s}_5)$,
7. $\tilde{s}_7$ is obtained from $\tilde{s}_6$ by possibly popping the stack; by Lemma 1(3), $\rho(\tilde{s}_7)$ subsumes $\rho(\tilde{s}_6)$,
8. $\tilde{s}_8$ is obtained from $\tilde{s}_7$ by either moving a record from the queue to the stack, in which case $\rho(\tilde{s}_8) = \rho(\tilde{s}_7)$, or by simply removing the record, in which case, by Lemma 1(3), $\rho(\tilde{s}_8)$ subsumes $\rho(\tilde{s}_7)$,
9. $\tilde{s}_9$ is obtained from $\tilde{s}_8$ by applying $\mathsf{canonic}$; by repeated applications of Lemma 1(2), $\rho(\tilde{s}_9) \sim \rho(\tilde{s}_8)$.

The result is that $\mathsf{lnext}(\tilde{s}, t) = \{\tilde{s}_9\}$ and $\rho(\tilde{s}_9)$ subsumes $\hat{s}_1$, as required. □

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.