A Symbolic Simulation Approach to Assertional Program Verification

John Matthews[1], J Strother Moore[2], Sandip Ray[2], and Daron Vroon[3]

[1] Galois Connections Inc., Beaverton, OR 97005.
[2] Dept. of Computer Sciences, University of Texas at Austin, Austin, TX 78712.
[3] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

Abstract. We present a method for automating deductive proofs of machine-level sequential programs modeled using operational semantics. Given programs annotated by the user with assertions at cutpoints, we show how to use the operational semantics of the machine to derive the verification conditions by symbolic simulation. No verification condition generator is required, nor is it necessary to manually specify an inductive invariant for the machine model. Both partial and total correctness are considered. The methodology has been formalized in both the ACL2 and Isabelle theorem provers, and applied to verify programs on operational machine models in ACL2.

1 Introduction

This paper presents a method for automating proofs of machine-level deterministic sequential programs modeled using operational semantics. In operational semantics, instructions of a program are modeled by their effect on the states of the underlying machine. Program verification involves proving that for every execution of the program from a state satisfying a given precondition, the state reached on termination satisfies the desired postcondition. Operational semantics provide clarity and concreteness to the modeled program; for example, it is possible in executable logics to run simulations to corroborate the model with the program as it is executed on actual machines [1]. However, traditional mechanical verification of operational models is tedious and complicated. It requires defining either an *inductive invariant* along with possibly a well-founded *ranking function* for every machine state, or a *clock function* [2] that characterizes, for each state, the number of machine transitions before termination. For realistic programs, the manual effort involved in the process is substantial.

Research in program verification theory has tended to favor *assertional reasoning*. In this approach, the program is annotated with assertions (and ranking functions) at certain *cutpoints*, that correspond to the entry and exit of basic blocks. The semantics of the language is then used to generate from these annotations a set of formulas called *verification conditions*; the formulas guarantee that whenever program control reaches a cutpoint the associated assertions hold for the corresponding state. These formulas are then verified, possibly using a theorem prover. Assertional reasoning in practice requires a *verification condition generator* (VCG) that generates verification conditions from an annotated program. The approach is attractive since user interaction is limited to annotating the program cutpoints and proving the verification conditions; no inductive invariant or clock function is necessary. However, the method is complicated by the need to implement a VCG for the target language. A VCG has to encode the language semantics, and often needs to perform non-trivial simplifications to keep generated formulas manageable. Further, to formally reconcile the application of a VCG with operational semantics, the VCG itself needs to be verified with respect to the operational model. The implementation of an efficient VCG for a realistic language, let alone its verification, is non-trivial.

In this paper, we present a methodology that combines the strengths of operational semantics and assertional reasoning while avoiding their weaknesses. Our method applies to operational models and hence affords concreteness and executability. However, the method does not require manually specifying an inductive invariant, ranking function, or clock for every reachable machine state. Instead, the user annotates the program with assertions (and ranking functions) at cutpoints, as in assertional reasoning. We show how to derive the verification conditions mechanically from such annotated programs *without* a VCG.

At the heart of our approach is the observation that it is possible to mechanically specify a function that returns, for every cutpoint, the number of machine steps necessary to reach the next subsequent cutpoint, if such a cutpoint exists. The function is specified by a tail-recursive definition, and hence is uniformly definable in logics such as ACL2 [3] and Isabelle/HOL [4, § 9.2.3] that are expressive enough to admit arbitrary tail-recursive equations. We use this function to prove certain symbolic simulation rules which are then used to derive verification conditions from the annotated program. The method unifies and extends previous efforts by the individual authors to connect assertional reasoning with operational semantics [5–7]. In particular, our approach provides a uniform methodology for proving both partial and total correctness. Our method has been implemented in the ACL2 [8, 9] and Isabelle [4] theorem provers, and has been applied in ACL2 to verify programs on existing operational machine models. However, the paper itself assumes no familiarity with these systems. The proof scripts are available from the web page of the third author [10].

The remainder of the paper is organized as follows. In Section 2, we provide some background on operational and assertional proofs. We formally present our approach in Section 3. In Section 4, we illustrate the application of our method on two operational machine models. In Section 5, we discuss related research. We conclude in Section 6 with a discussion on future research directions.

2 Background

Operational semantics involves modeling program instructions by their effect on states of the underlying machine. One models each state as a tuple that specifies values for all machine variables such as program counter (pc), registers, memory, and so on. The meaning of a program is formalized by the "next state function" $\mathit{next} : S \to S$, where S is the set of states. Given a state s, $\mathit{next}(s)$ returns the state s' obtained by executing the instruction pointed to by the pc in s. For example, if the instruction is a `LOAD`, then s' might be obtained from s by pushing the contents of some variable on the stack and advancing the pc.

To model executions of the machine, one then defines the function *run* that executes the machine for n steps.

$$\mathit{run}(s,n) \triangleq \begin{cases} s & \text{if } n = 0 \\ \mathit{run}(\mathit{next}(s), n-1) & \text{otherwise} \end{cases}$$

Program correctness is formalized by three predicates, which we refer to as *pre*, *post*, and *exit*. Predicates *pre* and *post* are the preconditions and postconditions. Predicate *exit* specifies the "final states" of the machine. If the machine halts after execution of the program, then *exit* can be defined as: $\mathit{exit}(s) \triangleq (\mathit{next}(s) = s)$. More commonly, one is interested in verifying a program component or subroutine, and *exit* recognizes the return of control from that component.

Partial correctness involves showing that if, starting from a state that satisfies *pre* the machine reaches an *exit* state, then *post* holds for the *exit* state. Nothing is claimed if the machine never reaches an *exit* state. *Total correctness* involves showing both partial correctness and *termination*, that is, the machine, starting from any state satisfying *pre*, eventually reaches an *exit* state. Formally, partial correctness and termination are specified by the following formulas:

Partial Correctness: $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s,n)) \Rightarrow \mathit{post}(\mathit{run}(s,n))$
Termination: $\forall s : \mathit{pre}(s) \Rightarrow (\exists n : \mathit{exit}(\mathit{run}(s,n)))$

In this paper, we assume that an operational model is provided by defining *next*, *pre*, *post*, and *exit*. Traditional verification then typically follows one of two approaches, namely *inductive invariants* or *clock functions*. For *inductive invariants*, one defines a predicate *inv* and proves the following properties:

1. $\forall s : \mathit{pre}(s) \Rightarrow \mathit{inv}(s)$
2. $\forall s : \mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{next}(s))$
3. $\forall s : \mathit{inv}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

These properties can be used to show partial correctness by first proving that for any $n \in \mathbb{N}$ and any state s, if $\mathit{inv}(s)$ holds, then $\mathit{inv}(\mathit{run}(s,n))$ holds. For termination, one additionally defines a *ranking function* $m : S \to W$ where set W is well-founded under some relation $\prec$, and proves the following property:

4. $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow m(\mathit{next}(s)) \prec m(s)$.

Well-foundedness then guarantees that the termination condition holds.

In the *clock functions* approach, the user defines a function $\mathit{clock} : S \to \mathbb{N}$ and proves the following formulas as theorems:

1. $\forall s : \mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$
2. $\forall s : \mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

The two conditions imply total correctness; termination is guaranteed since for every state s satisfying *pre*, there exists an n, namely $clock(s)$, such that $run(s,n)$ is an *exit* state. To express only partial correctness, one weakens the clock formulas 1 and 2 above so that $run(s, clock(s))$ satisfies *exit* and *post* only if an *exit* state is reachable from s. This is achieved by adding the predicate $(\exists n : exit(run(s,n)))$ as a conjunct in the antecedent of each formula. It is known [6] that inductive invariants and clock functions have the same logical strength. In this paper, the correctness theorems generated are in terms of clock functions.

Assertional reasoning involves attaching assertions at certain *cutpoints* of the program, which are entry and exit points of basic blocks. This notion is formalized by two predicates on the set S, namely *cut* and *assert*: *cut* recognizes the cutpoints and *assert* specifies the assertions that hold every time control reaches a *cut* state. A *VCG* uses these assertions to generate a set of *verification conditions*, which are then verified, possibly with a theorem prover. The guarantee from the conditions is informally stated as follows: "Let s be a *cut* state satisfying *assert*. Assume s is not an *exit* state. Let s' be the next *cut* state in an execution from s. Then $assert(s')$ must hold." The guarantee implies that if some cutpoint satisfies *assert*, then every subsequent cutpoint must satisfy *assert* until an *exit* state is reached. Thus, if (i) initial and *exit* states are cutpoints, (ii) *pre* implies *assert* at the initial states, and (iii) *assert* implies *post* at *exit*, then the first *exit* state reached by an execution from a *pre* state must satisfy *post*. In classic assertional methods [11, 12], the guarantee above is formalized by a *logic of programs*, where the meaning of an instruction is specified by an axiom schema over predicates on S. In an operational model, it is possible, though non-trivial, to show that each axiom schema is consistent with the model. Finally, to prove termination, one defines a ranking function $m : S \rightarrow W$, where W is a well-founded set, and verification conditions are extended to show that for any cutpoint s that is not an *exit* state, if s satisfies *assert*, and s' is the next cutpoint after s, then $m(s') \prec m(s)$. Well-foundedness now guarantees termination.

There are parallels between assertional reasoning and inductive invariants. The difference is that while inductive invariants involve assertions and ranking functions at every state, assertional methods only require them at cutpoints. Attaching assertions to every state requires a formal characterization of the reachable states, which becomes very complex in practice. However, assertional methods require two trusted tools, namely a theorem prover and a VCG. The outcome of our work is that the benefits of assertional reasoning can be obtained using symbolic simulation *without* a VCG.

3 Methodology

Assume that an operational model has been defined by functions *next*, *pre*, *post*, and *exit*, and predicates *cut* and *assert* are specified. We first define the function *csteps* that computes the number of steps to the first cutpoint from s.

$$csteps(s,i) \triangleq \begin{cases} i & \text{if } cut(s) \\ csteps(next(s), i+1) & \text{otherwise} \end{cases}$$

The definition is tail-recursive, and therefore logically consistent [3][4, §9.2.3]. If j is the minimum number of transitions required to reach a cutpoint from s, then $\mathit{csteps}(s, i)$ returns $i + j$; however, if no cutpoint is reachable from s, the definition does not specify the value returned. We now formalize the notion of "next cutpoint". To do this, we first fix a *dummy state* d such that $\mathit{cut}(d) \Leftrightarrow \forall s : \mathit{cut}(s)$. Thus if there is *some* state that is not a cutpoint, then d is not a cutpoint. State d can be uniformly defined using a choice operator. Then $\mathit{nextc}(s)$ returns the first cutpoint from s if some cutpoint is reachable, and otherwise d:

$$\mathit{nextc}(s) \triangleq \begin{cases} \mathit{run}(s, \mathit{csteps}(s, 0)) & \text{if } \mathit{cut}(\mathit{run}(s, \mathit{csteps}(s, 0))) \\ d & \text{otherwise} \end{cases}$$

The following formulas are formal renditions of verification conditions generated in assertional reasoning. In particular, **C4** specifies that if the assertions hold for some cutpoint s that is not an *exit* state, then they hold for the state s' that is the next cutpoint encountered in an execution from s.

C1: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{cut}(s) \wedge \mathit{assert}(s)$
C2: $\forall s : \mathit{exit}(s) \Rightarrow \mathit{cut}(s)$
C3: $\forall s : \mathit{exit}(s) \wedge \mathit{assert}(s) \Rightarrow \mathit{post}(s)$
C4: $\forall s : \mathit{cut}(s) \wedge \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{next}(s)))$

We now show how we can automatically generate correctness theorems from **C1**-**C4**. We will subsequently return to the problem of automating proofs of **C1**-**C4**.

To generate the correctness theorems, we define the function *esteps* as follows to count the number of transitions up to the first *exit* state.

$$\mathit{esteps}(s, i) \triangleq \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{esteps}(\mathit{next}(s), i + 1) & \text{otherwise} \end{cases}$$

If some *exit* state is reachable from s, then $\mathit{esteps}(s, 0)$ returns the number of transitions required to reach the first *exit* state. Indeed, from the discussions in Section 2, $\mathit{esteps}(s, 0)$ can be thought of as a generic clock function. Partial correctness now follows from the following theorem.

Theorem 1. *Given conditions* **C1**-**C4***, the following formulas are theorems:*

1. $\forall s, n : \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{esteps}(s, 0)))$
2. $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{esteps}(s, 0)))$

Proof sketch: To prove formula 1, note that if some *exit* state is reachable from s, then $\mathit{esteps}(s, 0)$ returns the first reachable *exit* state. To prove formula 2, note that by **C4**, if *assert* holds for some cutpoint s, then *assert* holds for every cutpoint reachable from s until (and including) the first *exit* state. Since, by **C1**, a *pre* state satisfies *assert*, it follows that the first *exit* state reachable from a *pre* state satisfies *assert*. Formula 2 then follows from **C3**. □

For termination, assertional reasoning requires a ranking function $m : S \rightarrow W$, where the set W is well-founded under some relation $\prec$. **C5** and **C6** below are formalizations of the termination condition.

C5: $\forall s : \mathit{cut}(s) \wedge \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow m(\mathit{nextc}(\mathit{next}(s))) \prec m(s)$
C6: $\forall s : \mathit{cut}(s) \wedge \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{cut}(\mathit{nextc}(\mathit{next}(s)))$

By the following theorem, total correctness follows from **C1-C6**.

Theorem 2. *Given conditions* **C1**-**C6***, the following formulas are theorems:*

1. $\forall s : \mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{esteps}(s, 0)))$
2. $\forall s : \mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{esteps}(s, 0)))$

Proof sketch: From Theorem 1 it suffices to show that from every state s satisfying *pre*, there exists some reachable *exit* state. By **C1** and **C6**, we can show that for every non-exit cutpoint p reachable from s, there is a subsequently reachable cutpoint p'. But, by **C5** and well-foundedness of $\prec$, eventually one of these cutpoints must be an exit state. □

How do we automate proofs of **C1-C6**? The complicated constraints are **C4** for partial correctness, and **C4-C6** for total correctness. These are exactly the constraints certified by a VCG in assertional proofs. To automate their proof without a VCG, we use the following two properties of *nextc*, which can be easily proven from the definitions of *nextc* and *csteps*.

SSR1: $\forall s : \neg \mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = \mathit{nextc}(\mathit{next}(s))$.
SSR2: $\forall s : \mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = s$.

For any state s such that a cutpoint is reachable from s in a fixed number of transitions, the theorems above can be used to determine $\mathit{nextc}(s)$. Formulas **SSR1** and **SSR2** can be treated as *symbolic simulation rules* whose application causes repeated expansion of the function *next* to ascertain if a cutpoint has been reached. If cutpoints correspond to entry and exit of the basic blocks of the program, then for every cutpoint s, the next cutpoint must be only a fixed number of transitions away; hence the rules are sufficient to prove **C4-C6**. Note that if an insufficient number of cutpoints are provided, for example if some loop is not cut, the rules might cause repeated expansion of *next*, leading to an infinite loop. This is exactly the behavior anticipated in the corresponding situation using assertional reasoning.

Theorems 1 and 2 and the symbolic simulation rules **SSR1** and **SSR2** have been proved using ACL2 and Isabelle. The proofs are independent of the actual definitions of *next*, *pre*, *post*, and *exit*; thus, the verification of concrete programs can be automated by instantiating the theorems for concrete machine models. Indeed, in ACL2, we have implemented a macro to generate partial (resp., total) correctness theorems for concrete programs by performing the following steps:

1. Mechanically generate functions *csteps*, *nextc*, and *esteps* for the concrete model and instantiate symbolic simulation theorems **SSR1** and **SSR2**.
2. Use symbolic simulation to prove constraints **C1-C4** (resp., **C1-C6**).
3. Prove correctness by instantiating Theorem 1 (resp., 2).

```
100  pushsi 1     *start*
102  dup
103  dup
104  pop 20                       fib0 := 1;
106  pop 21                       fib1 := 1;
108  sub                          n := max(n-1,0);
109  dup          *loop*
110  jumpz 127                    if n == 0, goto *done*;
112  pushs 20
113  dup
115  pushs 21
117  add
118  pop 20                       fib0 := fib0 + fib1;
120  pop 21                       fib1 := fib0 (old value);
122  pushsi 1
124  sub                          n := max(n-1,0);
125  jump 109                     goto *loop*;
127  pushs 20     *done*
129  add                          return fib0 + n;
130  halt         *halt*
```

Fig. 1. TINY Assembly Code for Generating the nth Fibonacci Sequence

4 Applying the Techniques

In this section, we demonstrate our method by verifying two illustrative programs on separate concrete machine models in ACL2. The operational details of the machines are irrelevant to this paper; the models were chosen simply because they had been previously formalized in ACL2 and are accessible to us.

4.1 An Iterative Program: Fibonacci on the TINY Machine

Consider the iterative assembly language program shown in Fig. 1 to generate the n-th Fibonacci number, based on this definition:

$$fib(n) \triangleq \begin{cases} 1 & \text{if } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

The program runs on TINY [1], a stack-based operational machine model with 32-bit word size. TINY has been developed at Rockwell Collins as an example of an analyzable, high-speed simulator. The program is a compilation of the standard iterative implementation to compute the Fibonacci sequence. In Fig. 1, the program counter values for the loaded program are shown to the left of each instruction, and pseudo-code for the high-level operations is shown at the extreme right of the corresponding rows. The two most recently computed values of the Fibonacci sequence are stored in memory addresses 20 and 21, and the loop counter n is maintained on the stack. Each loop iteration puts the sum of

Program Counter	Assertions
`*start*`	$(\texttt{tos}(s) = k) \wedge (0 \le k) \wedge \texttt{fib-loaded}(s)$
`*loop*`	$(\texttt{mem}[20] = \mathit{fix}(\mathit{fib}(k - \texttt{tos}(s)))) \wedge (0 \le \texttt{tos}(s) \le k) \wedge$ $(\texttt{mem}[21] = \mathit{fix}(\mathit{fib}(k - \texttt{tos}(s) - 1))) \wedge \texttt{fib-loaded}(s)$
`*done*`	$(\texttt{mem}[20] = \mathit{fix}(\mathit{fib}(k))) \wedge (\texttt{tos}(s) = 0) \wedge \texttt{fib-loaded}(s)$
`*halt*`	$(\texttt{tos}(s) = \mathit{fix}(\mathit{fib}(k)))$

Fig. 2. Assertions for the Fibonacci Program on TINY

these numbers in address 20, and moves the old value of 20 to 21. Since TINY performs 32-bit integer arithmetic, given a number k the program computes the low-order 32 bits of $\mathit{fib}(k)$. For this model, the cutpoints are states with program counter values associated with labels `*start*`, `*loop*`, `*done*`, and `*halt*` that correspond to program initiation, loop test, loop exit, and program termination respectively. The predicates *pre*, *post*, and *exit* for this model are as follows.

– $\mathit{pre}(k, s) \triangleq (\texttt{pc}(s) = *\texttt{start}*) \wedge (\texttt{tos}(s) = k) \wedge (0 \le k) \wedge \texttt{fib-loaded}(s)$
– $\mathit{post}(k, s) \triangleq (\texttt{tos}(s) = \mathit{fix}(\mathit{fib}(k)))$
– $\mathit{exit}(s) \triangleq (\texttt{pc}(s) = *\texttt{halt}*)$

Here $\texttt{pc}(s)$ and $\texttt{tos}(s)$ return the program counter and top of stack at state s, and $\mathit{fix}(n)$ returns the low-order 32 bits of n. Predicate `fib-loaded` holds at state s if the program in Fig. 1 is loaded in the memory starting at location `*start*`. Predicates *pre*, *post*, and *exit* specify the classical correctness conditions for a Fibonacci program: *pre* specifies that a non-negative integer k which is small enough to fit into the machine word is at the top of stack at program initiation, and *post* specifies that upon termination, $\mathit{fix}(\mathit{fib}(k))$ is at the top of the stack.

The assertions associated with the different cutpoints are shown in Fig. 2. They are fairly traditional. The key assertion is the loop invariant which specifies that the two most recently computed numbers stored at addresses 20 and 21 are $\mathit{fix}(\mathit{fib}(k - \texttt{n}))$ and $\mathit{fix}(\mathit{fib}(k - \texttt{n} - 1))$ respectively, where `n` is the loop count stored at the top of the stack when the control reaches the loop test.

For total correctness, we also need to specify a ranking function. The ranking function m we use maps the cutpoints to the well-founded set of ordinals below ϵ_0. Note that for this program it is possible to specify a ranking function that maps cutpoints to natural numbers; ordinals are used merely for succinctness and because of the extensive support provided for ordinal operations in ACL2 in the context of termination proofs [13]. Function m is defined as:

$$m(s) \triangleq \begin{cases} 0 & \text{if } \mathit{exit}(s) \\ (\omega \cdot_o \texttt{tos}(s)) +_o |{*}\texttt{halt}{*} - \texttt{pc}(s)| & \text{otherwise} \end{cases}$$

where ω is the first infinite ordinal, and $\cdot_o$ and $+_o$ are ordinal multiplication and addition operators. Informally, m can be viewed as a lexicographic ordering of the loop count and the difference between the location `*halt*` and $\texttt{pc}(s)$.

```
class Factorial {
  public static int fact (int n) {
   if (n > 0) return n*fact(n-1);
   else return 1;
  }
}
```

Fig. 3. Java Program for Computing Factorial

```
Method int fact (int)
0  ILOAD_0                                *start*
1  IFLE 12                                        if (n<=0) goto *done*
4  ILOAD_0
5  ILOAD_0
6  ICONST_1
7  ISUB
8  INVOKESTATIC #4 <Method int fact (int)>        x:= fact(n-1)
11 IMUL                                           x:= n*x
12 IRETURN                                *ret*   return x
13 ICONST_1                               *done*
14 IRETURN                                *base*  return 1
```

Fig. 4. M5 Bytecode for the Factorial Method

4.2 A Recursive Program: Factorial on the JVM

We now apply our method to verify JVM bytecodes for the Java factorial method `fact` shown in Fig. 3. The machine model we use is M5 [14], which has been developed at the University of Texas to formally reason about JVM bytecodes. M5 provides operational semantics for a significant fragment of the JVM in ACL2. It specifies 138 bytecodes, and supports features like invocation of static, special, and virtual methods, inheritance rules for method resolution, multithreading, and synchronization via monitors. The bytecodes for `fact`, shown in Fig. 4, are produced by disassembling output of `javac`, and can be loaded on to M5.

The factorial method is recursive. For recursive methods, the characterization of cutpoints must take into account not only the program counter but also the "depth" of recursive invocations. On the JVM, an invocation involves recording the return address in the current call frame of the executing thread and pushing a new call frame with the invoked method on the call stack. The precondition, postcondition, assertions etc., for the `fact` method are described below.

- The precondition specifies that some thread `th` is poised to execute the instructions of the `fact` method invoked with some 32-bit integer argument n, the call stack of `th` has height h, and the `pc` is at location labeled `*start*`.
- A state is a cutpoint if either **(i)** the call stack of `th` has height less than h (that is, the initial invocation has been completed), or **(ii)** the `pc` is in one of the locations labeled `*start*`, `*ret*`, or `*base*` (that is, the program is about to initiate execution of, or return from, the current invocation).

– A state s is an *exit* state if the call stack of `th` has height less than h.
– The postcondition specifies that *fix*(*mfact*(n)) is on the top of the operand stack of `th`, where *mfact* is the standard recursive definition of factorial:

$$mfact(n) \triangleq \begin{cases} 1 & \text{if } n \leq 1 \\ n \cdot mfact(n-1) & \text{otherwise} \end{cases}$$

– Let the height of the call stack for `th` at some cutpoint h'. If $h' < h$, the assertions specify merely that the postcondition holds. Otherwise, let $i := (n - h' + h)$. We assert that **(i)** the top i frames in the call stack correspond to successive invocations of `fact`, **(ii)** the return addresses are properly recorded on all the frames (other than the frame being executed), and **(iii)** if the `pc` is at location `*ret*` or `*base*` (that is, poised to return from the current frame), then *fix*(*mfact*(i)) is about to be returned.

The height of the call stack merely tracks the "recursion depth" of the execution. Further, since the assertions involve characterization of the different call frames in the call stack of the executing thread, one might be inclined to think that specification of assertions for recursive programs require careful consideration of the operational details of the JVM. In fact, that is not the case. The key insight is to recognize that the next instruction on a method invocation is *not* the following instruction on the byte stream of the caller but the first instruction of the callee. The instruction following the invocation is executed immediately after return from the callee. Hence one only needs to ensure that **(i)** the assertions at invocation can determine the execution of the callee, and **(ii)** the assertions at return can determine the subsequent execution of the caller. However, for a recursive program the caller and the callee are "copies" of the same method, and so the assertions must be a symmetric characterization of all call frames invoked in the recursive call. For the factorial program, the characterization is merely that in the i-th recursive call to `fact`, the system computes *fix*(*mfact*(i)), and this value is returned from the callee to the caller on return.

For termination, our ranking function m maps each cutpoint s to an ordinal representing the lexicographic pair consisting of the invocation argument for the current call frame and the height of the call stack at s. Note that along the successive recursive invocations of `fact`, the argument of the recursive calls decreases, while along the successive returns, the depth of the call stack decreases.

5 Related Work

Operational semantics were introduced by McCarthy [15]. The notion of assertions was used by Goldstein and von Neumann [16], and Turing [17], and made explicit by Floyd [18], Hoare [11], and Dijkstra [12]. Hoare and Dijkstra introduced *program logics* to provide a formal basis for the assertional method. King [19] wrote the first mechanized VCG. In mechanical theorem proving, operational semantics has been used extensively for program verification. Operational semantics has been particularly successful in ACL2 and its predecessor, Nqthm, which have been used for verifying programs in several large machine models [2, 20, 21]. Operational models have also been used in Isabelle/HOL for formalization of Java and the JVM [22], and in PVS to model graphical state chart languages [23]. Assertional methods have been applied, using a verified VCG, to reason about pointers in Isabelle [24], and C programs in HOL [25].

Our work is influenced by two related previous efforts by the individual authors, namely Moore [5] for verifying partial correctness, and Matthews and Vroon [7] for showing termination. Indeed, a key motivation of our work has been to extend these two methods to uniformly handle both partial and total correctness. To our knowledge, the method of [5] is the first to integrate assertional reasoning with operational semantics without requiring a verified VCG. However, while we design symbolic simulation rules to determine the subsequent cutpoint given a state s, the method of [5] defines a tail-recursive predicate *inv*:

$$inv(s) \triangleq \begin{cases} assert(s) & \text{if } cut(s) \\ inv(next(s)) & \text{otherwise} \end{cases}$$

The method then attempts to prove that *inv* is an inductive invariant, and reduces the proof to the constraint **C4** discussed in Section 3. For a cutpoint s statisfying *assert*, the definition of *inv* is used as a symbolic simulation rule to determine if *assert* holds for a subsequent cutpoint s'; symbolic simulation generates the same proof obligation as ours. However, this approach cannot be applied for termination, since there is no symbolic simulation rule to determine the value of the ranking function at s'. We overcome this limitation by constructing symbolic simulation rules to compute s' directly.

On the other hand, the method of [7] specifies symbolic simulation rules similar to ours, based on tail-recursive clock functions, which are then used in termination proofs. Our work differs in the treatment of assertions and cutpoints. For example, in [7], a single predicate *at-cutpoint* characterizes the cutpoints together with their assertions. However, this is inadequate for partial correctness proofs. The problem is that conflating assertions with cutpoints causes function *nextc* to "skip past" cutpoints that do not satisfy their corresponding assertion, on their way to one that does. However, one of those skipped cutpoints could be an *exit* state, and so the postcondition can not be inferred. Thus partial correctness becomes unprovable. Characterization of the cutpoints must be disentangled from the assertions in order to verify partial and total correctness in a unified framework.

6 Conclusion and Future Work

We have presented a method for verifying sequential programs modeled using operational semantics. The method requires no VCG implementation or invariant. Instead, the user annotates the program with assertions and ranking functions at cutpoints, exactly as in assertional reasoning. Symbolic simulation is used to generate and prove the verification conditions, which are then traded for a correctness theorem by automatically generating a tail-recursive clock. Both partial and total correctness are handled uniformly. The approach is attractive for several reasons. First, the use of operational models affords validation with respect to actual programs via simulation, and specification of correctness in a general-purpose mathematical logic. Secondly, assertional reasoning factors out the complexity of operational details and enables the user to focus on key program assertions. Indeed, it is these operational details that often make the task of verifying programs on realistic machine models daunting. For example, an inductive invariant that "persists" along a method invocation on the JVM needs to consider side effects to many state components like the heap, thread table, and class table. In our approach, the only manual assistance required is the definition of the assertions and ranking functions. Third, implementing an efficient reliable VCG for any realistic language, let alone its verification, is a substantial enterprise. Further, a VCG needs to implement several theorem proving techniques to generate manageable formulas. By directly generating verification conditions in the logic of the theorem prover, we afford reuse and extension of existing theorems and proof strategies. Finally, since we generate correctness theorems uniformly in terms of clock functions, it is possible to mechanically compose proofs of different program components [6].

We plan to apply this method to verify programs on realistic machine models. A key target application is the high-assurance *certifying compiler* being developed at Galois Connections Inc. to compile programs in Cryptol$^{\text{TM}}$ specification language [26] into object code for the Rockwell Collins AAMP7$^{\text{TM}}$ microprocessor. The goal is to generate, in addition to object code, a proof to certify that the generated code implements the Cryptol semantics of the source program; we plan to apply this approach to automatically produce verification conditions for showing correctness of the object code with respect to an existing formalization of AAMP7 in ACL2. We have also implemented the method in the Isabelle theorem prover [4], although we have not yet applied it to any examples.

We have assumed in this paper that the programs are deterministic, that is, the next state of the machine can be specified as a *function* of the current state. In non-deterministic programs that arise, for example, in asynchronous distributed systems, one specifies the "possible next states" which are defined by a *transition relation.* Our method cannot be directly used for verification of such systems. In future work we plan to represent non-deterministic systems using a linear-time path semantics; our method should then be applicable.

Acknowledgements We thank Jared Davis, Dave Greve, David Hardin, Matt Kaufmann, Robert Krug, Jeff Lewis, Hanbing Liu, Pete Manolios, Mark Shields, Matt Wilding, Thomas Wahl, and Bill Young, for comments, suggestions, and advice.

References

1. Greve, D., Wilding, M., Hardin, D.: High-Speed, Analyzable Simulators. In Kaufmann, M., Manolios, P., Moore, J.S., eds.: Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers (2000) 89–106

2. Boyer, R.S., Moore, J.S.: Mechanized Formal Reasoning about Programs and Computing Machines. In Veroff, R., ed.: Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, MIT Press (1996) 141–176
3. Manolios, P., Moore, J.S.: Partial Functions in ACL2. Journal of Automated Reasoning **31** (2003) 107–127
4. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher Order Logics. Volume 2283 of LNCS. Springer-Verlag (2002)
5. Moore, J.S.: Inductive Assertions and Operational Semantics. In Geist, D., ed.: CHARME 2003. Volume 2860 of LNCS., Springer-Verlag (2003) 289–303
6. Ray, S., Moore, J.S.: Proof Styles in Operational Semantics. In: FMCAD 2004. LNCS 3312, Springer-Verlag (2004) 67–81
7. Matthews, J., Vroon, D.: Partial Clock Functions in ACL2. In Kaufmann, M., Moore, J.S., eds.: 5th ACL2 Workshop. (2004)
8. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
9. Kaufmann, M., Manolios, P., Moore, J.S., eds.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
10. (`http://www.cs.utexas.edu/users/sandip/`)
11. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12** (1969) 576–583
12. Dijkstra, E.W.: Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. Communications of the ACM **18** (1975) 453–457
13. Manolios, P., Vroon, D.: Algorithms for Ordinal Arithmetic. In: 19th International Conference on Automated Deduction. LNAI, Springer-Verlag (2003) 243–257
14. Moore, J.S.: Proving Theorems about Java and the JVM with ACL2. In Broy, M., Pizka, M., eds.: Models, Algebras, and Logic of Engineering Software, Amsterdam, IOS Press (2003) 227–290
15. McCarthy, J.: Towards a Mathematical Science of Computation. In: Proceedings of the Information Processing Congress. Volume 62., North-Holland (1962) 21–28
16. Goldstein, H.H., J. von Neumann: Planning and Coding Problems for an Electronic Computing Instrument. In: John von Neumann, Collected Works, Volume V, Pergamon Press, Oxford (1961)
17. Turing, A.M.: Checking a Large Routine. In: Report of a Conference on High Speed Automatic Calculating Machine, University Mathematical Laboratory, Cambridge, England (1949) 67–69
18. Floyd, R.: Assigning Meanings to Programs. In: Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs. Volume XIX., Providence, Rhode Island, American Mathematical Society (1967) 19–32
19. King, J.C.: A Program Verifier. PhD thesis, Carnegie-Melon University (1969)
20. Moore, J.S.: Piton: A Mechanically Verified Assembly Language. Kluwer Academic Publishers (1996)
21. Yu, Y.: Automated Proofs of Object Code for a Widely Used Microprocessor. PhD thesis, University of Texas at Austin (1992)
22. Strecker, M.: Formal Verification of a Java Compiler in Isabelle. In Voronkov, A., ed.: CADE 2004. LNCS 2392, Springer-Verlag (2002) 63–77
23. Hamon, G., Rushby, J.: An Operational Semantics for Stateflow. In: FASE 2004. LNCS 2984, Springer-Verlag (2004) 229–243
24. Mehta, F., Nipkow, T.: Proving Pointer Programs in Higher Order Logic. In Baader, F., ed.: CADE 2003. LNAI 2741, Springer-Verlag (2003) 121–135
25. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (1998)
26. (`http://www.cryptol.net`)