Verification Condition Generation via Theorem Proving

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 to convert (i) an operational semantics for a given machine language, and (ii) an off-the-shelf theorem prover, into a high assurance verification condition generator (VCG). Given a program annotated with assertions at cutpoints, we show how to use the theorem prover directly on the operational semantics to generate verification conditions analogous to those produced by a custom-built VCG. Thus no separate VCG is necessary, and the theorem prover can be employed both to generate and to discharge the verification conditions. The method handles both partial and total correctness. It is also compositional in that the correctness of a subroutine needs to be proved once, rather than at each call site. The method has been used to verify several machine-level programs using the ACL2 theorem prover.

1 Introduction

Operational semantics has emerged as a popular approach for formal modeling of complex computing systems. In this approach, a program is modeled by defining an interpreter that specifies the effect of executing its instructions on the states of the underlying machine. Unfortunately, traditional code proofs based on operational models have been tedious and complex, requiring the user to define global invariants which are preserved on each transition or a *clock function* that precisely characterizes the number of machine steps to termination [1, 2].

Research in program verification has principally focused on assertional reasoning [3, 4]. Here a program is annotated with assertions at cutpoints. From these annotations, one derives a set of formulas or *verification conditions*, which guarantee that whenever program control reaches a cutpoint the associated assertions hold. Assertional methods generally rely on (i) a *verification condition generator* (VCG) to generate verification conditions from an annotated program, and (ii) a theorem prover to discharge these conditions.

In this paper, we present a method for verifying deterministic sequential programs, using operational semantics, that inherits the benefits of the assertional methods. Given an annotated program and an operational semantics, we show how to configure a theorem prover to emulate a VCG for generating (and discharging) the verification conditions.

In this section, we first provide a brief overview of operational models and assertional proof approaches to establish the relevant background. We then discuss our contributions in greater detail.

1.1 Background

In *operational semantics*, a program is modeled by its effects on the underlying machine state. A state is viewed as a tuple of values of all machine variables like the program counter (pc), registers, memory, etc. One defines a transition function $\mathit{next} : S \to S$ where S is the set of states: for a state s, $\mathit{next}(s)$ returns the state after executing one instruction from s. Executions are modeled by a function $\mathit{run} : S \times \mathbb{N} \to S$ which returns the state after n transitions from s.

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

Correctness is formalized with three predicates *pre*, *post*, and *exit*, on set S. Predicates *pre* and *post* are the preconditions and postconditions, and *exit* specifies the "final states"; when verifying a program component, *exit* is defined to recognize the return of control from that component. There are two notions of correctness, *partial* and *total*. Partial correctness involves showing that for any state s satisfying *pre*, the predicate *post* holds at the first *exit* state reachable from s (if some such state exists). Total correctness involves showing both partial correctness and *termination*, that is, the machine starting from a state s satisfying *pre* eventually reaches an *exit* state. Partial correctness and termination are formalized as follows:

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

Several deductive techniques have been devised to facilitate proofs of the above statements. One method is to define a global invariant *inv* satisfying **I1**-**I3** below:

I1: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{inv}(s)$
I2: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{inv}(\mathit{next}(s))$
I3: $\forall s : \mathit{inv}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

Partial correctness follows from **I1**-**I3**. By **I1** and **I2**, any state reachable from a *pre* state s up to (and including) the first *exit* state p satisfies *inv*; **I3** then guarantees $\mathit{post}(p)$. For total correctness, one also defines a function $\mathit{rank} : S \to W$ where W is well-founded under some ordering $\prec$, and shows **I4** below. Well-foundedness guarantees termination.

I4: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{next}(s)) \prec \mathit{rank}(s)$.

Another approach is to use *clock functions*. A clock function $\mathit{clock} : S \to \mathbb{N}$ satisfies conditions **C1**-**C3** below:

C1: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$
C2: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$
C3: $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow (\mathit{clock}(s) \leq n)$

C1-**C3** imply total correctness: for every *pre* state s, there exists an n, namely $\mathit{clock}(s)$, such that $\mathit{run}(s, n)$ is an *exit* state, guaranteeing termination. To express only partial correctness, one weakens **C1** and **C2** by adding the predicate $(\exists n : \mathit{exit}(\mathit{run}(s, n)))$ as a conjunct in the antecedents. It is known [5] that global invariants and clock functions have the same logical strength in that a correctness proof in one method can be mechanically transformed into the other.

Assertional methods are based on annotating a program with assertions at certain control points called *cutpoints* that typically include loop tests and program entry and exit [3, 6]. To formalize this, assume that we have two predicates *cut* and *assert*, where *cut* recognizes the cutpoints and *assert* specifies the assertions at each cutpoint. Commonly *cut* is a predicate on the pc values but might occasionally involve other state components. A VCG generates a set of *verification conditions* from the annotated program, which are verified using a theorem prover. The guarantee provided by the process is informally stated as: "Let p be a non-exit *cut* state satisfying *assert*. Let q be the next *cut* state in an execution from p. Then $\mathit{assert}(q)$ must hold." Thus, if (i) initial (*i.e.*, *pre*) and *exit* states are cutpoints, (ii) *pre* implies *assert*, and (iii) *assert* implies *post* at *exit*, then the first *exit* state reachable from a *pre* state satisfies *post*. Finally, for termination, one also defines a ranking function $\mathit{rank} : S \rightarrow W$, where W is a well-founded set, and shows that for any non-exit cutpoint p satisfying *assert*, if q is the next cutpoint, then $\mathit{rank}(q) \prec \mathit{rank}(p)$. Notice that both assertions and ranking functions are attached to cutpoints rather than to every state.

1.2 Contributions of this Paper

Operational semantics and assertional methods have complementary strengths. Operational models have been lauded for clarity and concreteness [1, 7], and facilitate the validation of formal models by simulation [7, 8]. However, performing code proofs with such models is cumbersome: defining an appropriate global invariant or clock function requires understanding of the effect of *each* transition on the machine state [1, 9, 2]. Assertional methods factor out verification complexity by restricting user focus to cutpoints, but require a VCG which must be trusted. A VCG encodes the language semantics as formula transformations. Most VCGs also perform on-the-fly simplifications to keep the generated formulas manageable. Implementing a practical VCG, let alone ensuring its correctness by verifying it against an operational semantics, is non-trivial [10].

In this paper, we present a technique to integrate assertional methods with operational semantics that is suitable for use with general-purpose theorem proving and does not depend on a trusted VCG. As in assertional reasoning, the user annotates the program at cutpoints. However, instead of implementing a VCG we show how to configure the theorem prover to generate verification conditions by symbolic simulation on the operational model. The result is a high assurance program verifier with an off-the-shelf theorem prover as the only trusted

component. The method handles both partial and total correctness, and recursive procedures. It is also compositional; subroutines can be verified separately rather than at every call site. The method has been mechanized in the ACL2 theorem prover [11], and used to reason about several machine-level programs. The basic approach (*i.e.*, without composition) has also been formalized in the Isabelle theorem prover [12].

The rest of the paper is organized as follows. We present the basic approach in Section 2. In Section 3, we discuss compositionality and means for handling recursive procedures. In Section 4, we present illustrative applications of the method. We discuss related work in Section 5 and conclude in Section 6.

2 Basic Methodology

Assume that we have defined *next*, *pre*, *post*, *exit*, *cut*, and *assert*, as described in Section 1.1. Consider the following function *csteps*:

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

If j is the minimum number of transitions to a cutpoint from state s, then $\mathit{csteps}(s,i)$ returns $i+j$; the recursion does not terminate if no cutpoint is reachable. Generally, defining a recursive function requires showing that the recursion terminates. However, if the definition is tail-recursive as above, then it is admissible in theorem provers whose logics support Hilbert's choice operator; the defining axiom can be witnessed by a total function that returns an arbitrary constant when the recursion does not terminate [13][12, §9.2.3].

We now formalize the notion of "next cutpoint". Fix a state $\mathbf{d}$ such that $\mathit{cut}(\mathbf{d}) \Leftrightarrow (\forall s : \mathit{cut}(s))$. State $\mathbf{d}$ can be defined with a choice operator. Then $\mathit{nextc}(s)$ returns the first reachable cutpoint from s if any, else $\mathbf{d}$:

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

With these definitions, we formalize verification conditions as formulas **V1**-**V5**. Notice that the formulas involve obligations only about assertions at cutpoints.

V1: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{assert}(s)$
V2: $\forall s : \mathit{assert}(s) \Rightarrow \mathit{cut}(s)$
V3: $\forall s : \mathit{exit}(s) \Rightarrow \mathit{cut}(s)$
V4: $\forall s : \mathit{assert}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$
V5: $\forall s : \mathit{assert}(s) \wedge \neg\mathit{exit}(s) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{next}(s)))$

The formulas imply partial correctness. To prove this, we define function *esteps* to count the number of transitions up to the first *exit* state, and *nexte* that returns the first reachable exit point. Note that *esteps* is tail-recursive.

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

$$\mathit{nexte}(s) \triangleq \mathit{run}(s, \mathit{esteps}(s,0))$$

We can take $\mathit{esteps}(s, 0)$ as the definition of a generic clock function. Partial correctness now follows from Theorem 1.

Theorem 1. *Suppose conditions* **V1**, **V3**-**V5** *hold. Let state s and natural number n be such that* $\mathit{pre}(s)$ *and* $\mathit{exit}(\mathit{run}(s, n))$. *Then* $\mathit{esteps}(s, 0) \leq n$, $\mathit{exit}(\mathit{nexte}(s))$, *and* $\mathit{post}(\mathit{nexte}(s))$.

Proof sketch: $\mathit{esteps}(s, 0) \leq n$ and $\mathit{exit}(\mathit{nexte}(s))$ hold since $\mathit{esteps}(s, 0)$ returns the number of steps to the first reachable *exit* state, if one exists. If *assert* holds for a cutpoint p, then by **V5** *assert* holds for every cutpoint reachable from p until (and, by **V3**, including) the first *exit* state. Since a *pre* state satisfies *assert* (by **V1**), the first *exit* state reachable from a *pre* state satisfies *assert*. Now $\mathit{post}(\mathit{nexte}(s)))$ follows from **V4**. □

For termination, we also need a well-founded *rank* over cutpoints. **V6** below formalizes the corresponding proof obligation. By Theorem 2, total correctness follows from **V1**-**V6**.

V6: $\forall s : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{nextc}(\mathit{next}(s))) \prec \mathit{rank}(s)$

Theorem 2. *Suppose* **V1**-**V6** *hold, and let s satisfy* *pre*. *Then* $\mathit{exit}(\mathit{nexte}(s))$ *and* $\mathit{post}(\mathit{nexte}(s))$ *hold.*

Proof sketch: To prove $\mathit{exit}(\mathit{nexte}(s))$, it suffices to show that some *exit* state is reachable from each *pre* state s. By **V1**, **V2**, and **V5**, for every non-exit cutpoint p reachable from s, there exists a subsequently reachable cutpoint p'. But, by **V6** and well-foundedness of $\prec$, eventually one of these cutpoints must be an *exit* state. Then $\mathit{post}(\mathit{nexte}(s))$ follows from $\mathit{exit}(\mathit{nexte}(s))$ and Theorem 1. □

We now discuss how the verification conditions are discharged for a concrete program. The non-trivial conditions are **V5** and **V6**, which involve relation between two consecutive cutpoints. To automate their verification, we use theorems **SSR1** and **SSR2** below, which are trivial consequences of the definition of *nextc*.

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

We use **SSR1** and **SSR2** as conditional rewrite rules oriented left to right. For any symbolic state s, the rules rewrite the term $\mathit{nextc}(s)$ to either s or $\mathit{nextc}(\mathit{next}(s))$ depending on whether s is a cutpoint, in the latter case causing a symbolic expansion of the definition of *next* possibly with auxiliary simplifications, and applying the rules again on the resulting term. Proofs of **V5** and **V6** thus cause the theorem prover to symbolically simulate the program from each cutpoint satisfying *assert* until the next cutpoint is reached, at which point we check if the new state satisfies assertions. The process mimics a "forward" VCG, but generates and discharges the verification conditions on a case-by-case basis.

3 Composing Correctness Statements

The basic method above did not treat subroutines compositionally. Consider verifying a procedure P that invokes a subroutine Q. Symbolic simulation from a cutpoint of P might encounter an invocation of Q, resulting in symbolic execution of Q. Thus subroutines have been treated as if they were in-lined. We often prefer to separately verify Q, and use its correctness theorem for verifying P. We now extend the method to afford such composition.

We will uniformly use the symbols P and Q to refer to invocations of the caller and callee respectively. We also use a subscript to distinguish between predicates about P and Q when necessary, for example referring to the postcondition for P as post_P.

For composition, it is convenient to extend the notion of *exit* states as follows. We define a predicate in_P to characterize states which are poised to execute an instruction in P or one of its callees. Then define $\mathit{exit}_P(s) \triangleq \neg \mathit{in}_P(s)$. Thus, exit_P recognizes *any* state that does not involve execution of P (or any subroutine), not just those that return control from P. Note that this does not change the notion of the *first* *exit* state from P. With this view, we add the new verification condition **CC** below, stating that no cutpoint of P is encountered during the execution of Q. The condition will be used in the proofs of additional rules **SSR3** and **SSR3′** that we define later, which are necessary for composition.

CC: $\forall s : \mathit{cut}_P(s) \Rightarrow \mathit{exit}_Q(s)$

Another key ingredient for composition is the formalization of *frame conditions* necessary to prove that P can continue execution after Q returns. A postcondition specifying that Q correctly performs its desired computation is not sufficient to guarantee this. For instance, Q, while correctly computing its return value, might corrupt the call stack preventing P from executing on return. To account for this, post_Q needs to characterize the global effect of executing Q, that is, specify how *each* state component is affected by the execution of Q. However, such global characterization of the effect of Q might be difficult. In practice, we require that post_Q is strong enough such that for any state s satisfying exit_Q and post_Q we can infer the control flow for continuing execution of P. For instance, if Q updates some "scratch space" which is irrelevant to the execution of P, then post_Q need not characterize such update. Then we prove the additional symbolic simulation rule **SSR3** (resp., **SSR3′**) below, which (together with **SSR1** and **SSR2**) affords compositional reasoning about total (resp., partial) correctness of P assuming that Q has been proven totally (resp., partially) correct. Here $\mathit{excut}_P(s) \triangleq \mathit{cut}_P(\mathit{nextc}_P(s))$.

SSR3: $\forall s : \mathit{pre}_Q(s) \Rightarrow \mathit{nextc}_P(s) = \mathit{nextc}_P(\mathit{nexte}_Q(s))$
SSR3′: $\forall s : \mathit{pre}_Q(s) \wedge \mathit{excut}_P(s) \Rightarrow \mathit{nextc}_P(s) = \mathit{nextc}_P(\mathit{nexte}_Q(s))$

Proof sketch: We only discuss **SSR3** since the proof of **SSR3′** is similar. By **CC** and the definition of esteps_Q, if s satisfies pre_Q and $n < \mathit{esteps}_Q(s, 0)$, then $\mathit{run}(s, n)$ does not satisfy cut_P. Hence the next cut_P state after s is the same

as the next cut_P state after the first exit_Q state reachable from s. The rule now follows from the definitions of *nextc* and *nexte*. □

We prioritize rule applications so that **SSR1** and **SSR2** are tried only when **SSR3** (resp., **SSR3′**) cannot be applied during symbolic simulation. Therefore, if Q has been proven totally correct and if a non-cutpoint state s encountered during symbolic simulation of P satisfies pre_Q, then **SSR3** "skips past" the execution of Q; otherwise we expand the transition function via **SSR2** as desired.

We need one further observation to apply **SSR3′** for composing partial correctness proofs. Note that **SSR3′** has the hypothesis $\mathit{excut}_P(s)$. To apply the rule, we must therefore know for a symbolic state s satisfying pre_Q whether some subsequent cutpoint of P is reachable from s. However, such a cutpoint, if one exists, can only be encountered *after* s. The solution is to observe that for partial correctness we can weaken the verification condition **V5** to **V5′** below. For a cutpoint s satisfying assertions, **V5′** requires the next subsequent cutpoint to satisfy the assertion only if some such cutpoint is reachable.

V5′: $\forall s : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \wedge \mathit{excut}(\mathit{next}(s)) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{next}(s)))$

V5′ allows us to assume $\mathit{excut}_P(\mathit{next}(s))$ for any non-exit cutpoint s of P. Now let b be some pre_Q state encountered during symbolic simulation. We must have previously encountered a non-exit cutpoint a of P such that there is no cutpoint between $\mathit{next}_P(a)$ and b. Assuming $\mathit{excut}_P(\mathit{next}(a))$ we can infer $\mathit{excut}_P(b)$ by the definitions of *excut* and *nextc*, enabling application of **SSR3′**.

Note that while we used the word "subroutine" for presentation, our treatment does not require P or Q to be subroutines. One can mark *any* program block by defining an appropriate predicate *in*, verify it separately, and use it to compositionally reason about programs that invoke it. In practice, we separately verify callees that (i) contain one or more loops, and (ii) are invoked several times, possibly by several callers. If Q is a straight-line procedure with complicated semantics, for instance some complex initialization code, we skip composition and allow symbolic simulation of P to emulate in-lining of Q.

We now turn to recursive procedures. So far we have considered the scenario where Q has been verified *before* P. This is not valid for recursive programs where P and Q are invocations of the same procedure. Nevertheless, we can still *assume* the correctness of Q while reasoning about P. The soundness of the assumption is justified by well-founded induction on the number of machine steps needed to reach the first *exit* state for P, and the fact that recursive invocations of P execute in fewer steps than P itself.

We end the description of the method with a note on its mechanization in ACL2. Observe that the proofs of Theorems 1 and 2, the symbolic simulation rules, and the justification for applying induction for recursive procedures above, do not depend on the actual definitions of *next*, *pre*, *post*, etc., but merely on conditions **V1**-**V5**, **V5′**, and **CC**. Thus we can verify concrete programs by instantiating the correctness theorems with the corresponding functions for the concrete machine model. In ACL2, we make use of a derived rule of inference called *functional instantiation* [14], which enables instantiation of theorems

about constrained functions with concrete functions satisfying the constraints. In particular, we have used constrained functions *pre*, *post*, *next*, etc., axiomatized to satisfy the verification conditions, and mechanically derived the remaining theorems and rules. This allows us to automate assertional reasoning on operational models by implementing a macro which performs steps 1-4 below.

1. Mechanically generate concrete versions of the functions *csteps*, *nextc*, *esteps*, etc., for the given operational semantics.
2. Functionally instantiate the generic symbolic simulation rules **SSR1**, **SSR2**, and **SSR3** (resp., **SSR3′**), and the justification for recursive procedures.
3. Use symbolic simulation to prove the verification conditions.
4. Derive correctness by functionally instantiating Theorems 1 and 2.

4 Applications

In this section, we discuss applications of the method in verification of concrete programs. All the examples presented have been verified in ACL2 using the macro mentioned above. We start with an assembly language Fibonacci program on a simple machine model called TINY [8]. The subsequent examples are JVM bytecodes compiled from Java for an operational model of the JVM in ACL2 called M5 [2]. The details of TINY or M5 are irrelevant to this paper; we chose them since they are representative of operational machine models in ACL2, and their formalizations were accessible to us.

4.1 Fibonacci Implementation on TINY

TINY is a stack-based 32-bit processor developed at Rockwell Collins Inc [8]. The Fibonacci program shown in Fig. 1 is the result of compiling the standard iterative implementation for this machine. TINY represents memory as a linear address space. The two most recently computed values of the Fibonacci sequence are stored in addresses `20` and `21`, and the loop counter `n` is maintained on the stack. TINY performs 32-bit integer arithmetic. Given a number k the program computes $\mathit{fix}(\mathit{fib}(k))$, where $\mathit{fix}(n)$ returns the low-order 32 bits of n, and *fib* is the mathematical Fibonacci function defined below:

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

The *pre*, *post*, and *exit* predicates for the verification of the Fibonacci program[4] are shown in Fig. 2, and the assertions at the different cutpoints in Fig. 3. They are fairly traditional. The key assertion is the loop invariant which specifies that the numbers at addresses `20` and `21` are $\mathit{fix}(\mathit{fib}(k-\mathtt{n}))$ and $\mathit{fix}(\mathit{fib}(k-\mathtt{n}-1))$

[4] Functions *pre* and *post* here take an extra argument k while our generic proofs used unary functions. This is admissible since one can functionally instantiate constraints with concrete functions having extra arguments, as long as such arguments do not affect the parameters (in this case s) involved in the constraints [14].

```
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 computing the nth Fibonacci sequence. The numbers to the left of each instruction is the pc value for the loaded program. High-level pseudo-code is shown at the extreme right. The `add` instruction at pc value `129` removes `0` from the top of stack; this trick is necessary since TINY has no `DROP` instruction.

respectively, where `n` is the loop count stored at the top of the stack when the control reaches the loop test. For partial correctness, no further user input is necessary. Symbolic simulation proves the standard verification conditions.

For total correctness, we additionally use the function *rank* below that maps the cutpoints to the well-founded set of ordinals below ϵ_0.

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

Here ω is the first infinite ordinal, and $\cdot_o$ and $+_o$ represent ordinal multiplication and addition. Informally, *rank* is a lexicographic ordering of the loop count and the difference between the location `*halt*` and *pc*(s).

4.2 Recursive Factorial Implementation on the JVM

Our next example involves JVM bytecodes for a recursive implementation of the factorial program (Fig. 4). We use an operational model of the JVM called M5, developed at the University of Texas [2]. M5 defines the semantics for 138 JVM instructions, and supports invocation of static, special, and virtual methods, inheritance rules for method resolution, multi-threading, and synchronization via monitors. The bytecodes in Fig. 4 are produced from the Java implementation by disassembling the output of `javac` and can be executed with M5.

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

Fig. 2. Predicates *pre*, *post*, and *exit* for the Fibonacci program. Here $\mathit{pc}(s)$ and $\mathit{tos}(s)$ return the program counter and top of stack at state s, and *fib-loaded* holds at state s if the program in Fig. 1 is loaded in the memory starting at location `*start*`.

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

Fig. 3. Assertions for the Fibonacci program

The example is an entertaining illustration of our treatment of recursion. With the exception of the recursive call, the procedure involves a straight line code. Thus we only need to specify the precondition and the postcondition. The precondition posits that the state s is poised to start executing the bytecodes for `fact` on argument k; the postcondition specifies that the return state pops the top frame from the call stack and stores $\mathit{fix}(\mathit{fact}(k))$ on the frame of the caller where *fact* is the mathematical factorial function. No further annotation is necessary. When symbolic simulation reaches the state in which the recursive call is invoked, it skips past the call (inferring the postcondition for the recursive call) and continues until the procedure exits. This stands in stark contrast to all the previously published ACL2 proofs of the method [2, 15], which require complex assertions to characterize each recursive frame in the call stack.

4.3 CBC-mode Encryption and Decryption

Our third example is a more elaborate proof of functional correctness of a Java program implementing encryption and decryption of an unbounded array of bits. By *functional correctness*, we mean that the composition of encryption and decryption yields the original plaintext. Functional correctness of cryptographic protocols has received considerable attention recently in formal verification [16, 17]. We refer the reader to Schneier [18] for an overview of cryptosystems.

Cryptographic protocols use a *block cipher* that encrypts and decrypts a fixed-size *block* of bits. We use blocks of 128 bits. Encryption and decryption of large data streams additionally require the following operations.

– A *mode of operation* extends the cipher from a single block to arbitrary block sequences. We use *Cipher Block Chaining* (CBC), which 'xor's a plaintext block with the previous ciphertext in the sequence before encryption.

```
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 Bytecodes for the Factorial Method

- *Padding* expands a bit sequence to one which is a multiple of a block length, so as to apply a block cipher; *unpadding* drops the padding during decryption.
- *Blocking* involves transforming an array of bits to an array of blocks for use by CBC encryption; *unblocking* is the obvious inverse.

Our Java implementation performs the following sequence of operations on an unbounded bit-array: (i) padding, (ii) blocking, (iii) CBC encoding, (iv) CBC decoding, (v) unblocking, and (vi) unpadding. It follows Slind and Hurd's HOL model of the operations [16], adapted for bit arrays. However, we do *not* implement a practical block cipher; our cipher 'xor's a 128-bit block with a key based on a fixed key schedule. The program constitutes about 300 lines of Java (with 18 subroutines), which compiles to 600 bytecodes.

We verify both partial and total functional correctness. The precondition specifies that the class table containing the routines is loaded and the current call frame contains a reference to an array a of bits in the heap; the postcondition requires that the array on termination is the same as a. Using ACL2 libraries on arithmetic and arrays, the only non-trivial user inputs necessary for the proofs are the loop invariants for the associated procedures. Furthermore, the only property of the block cipher used in reasoning about the CBC methods is that the encryption and decryption of 128-bit blocks are inverses. Thus, it is now possible to independently prove this invertibility property for a practical block cipher, and "plug in" the cipher to obtain a proof of the corresponding unbounded bit-array encryption.

5 Related Work

Operational semantics was introduced by McCarthy [19], and has since been used extensively for mechanical verification of complex programs. In particular, ACL2 and its predecessor Nqthm have used such models extensively [1, 2, 8, 20]. Operational models have also been used in Isabelle/HOL to formalize Java and the JVM [21], and in PVS to model state chart languages [22].

The notion of assertions was used by Goldstein and von Neumann [23], and Turing [24], and made explicit in the classic works of Floyd [3], Manna [6], Hoare [4], and Dijkstra [25]. King [26] wrote the first mechanized VCG. VCGs have been used extensively in practice, for example in the Extended Static Checker for Java (ESC/Java) [27], the Java certifying compiler [10], and the Praxis verification of Spark programs [28]. Several researchers have commented on the complexity of a practical VCG [29, 30]. There has also been significant research verifying VCGs via theorem proving [31–33]. In the context of theorem proving, assertions have also been used to verify C programs in HOL [34], and reason about pointers and BDD normalization algorithms in Isabelle [35, 36].

This work is influenced by two earlier efforts in ACL2 by the individual authors, namely Moore [15] and Matthews and Vroon [37], to emulate VCG reasoning with a theorem prover. Moore defines a tail-recursive predicate *inv* such that the proof of invariance of *inv* reduces to showing that each cutpoint satisfies assertions. However, since the definition of *inv* is tied to assertions, the method cannot be used to reason about ranking functions (and hence termination). Matthews and Vroon prove termination by directly characterizing cutpoints, but conflate assertions and cutpoints in a single predicate. Thus symbolic simulation can skip past cutpoints not satisfying assertions, and partial correctness cannot be inferred. Neither method handles composition or recursive procedures. Our work can be viewed as a unification and substantial extension of these efforts.

There are parallels between our work and research on proof-carrying code (PCC) [38]. VCGs are the key trusted components in PCCs. Similar to our work, foundational PCC research [39] ensures reliability of verification condition generation by relying only on a general-purpose theorem prover and the operational semantics of a machine language. However, while PCCs focus on automatic proofs of fixed safety properties (such as type and memory safety), our approach is geared towards verifying functional program correctness which requires more general-purpose assertions. We achieve this by using the simplification mechanisms of a theorem prover to automate verification condition generation.

An early implementation of our ACL2 macro is currently distributed with ACL2. Several researchers have personally communicated to the authors independent endeavors applying and extending the method. At Galois Connections Inc., Pike has applied the macro to verify programs on the Rockwell Collins AAMP7$^{\mathrm{TM}}$ processor [40]. At the National Security Agency, Legato has used it to verify an assembly language multiplier for the Mostek 6502 microprocessor. At Rockwell Collins Inc., Hardin *et al.* are independently extending the method and using it for AAMP7 and JVM code verification [41]. Fox has formalized the method in HOL4 and is applying it on ARM assembly language programs.

6 Summary and Conclusion

We have presented a method to apply assertional reasoning for verifying sequential programs based on operational semantics, that is suitable for use in mechanical theorem proving. Symbolic simulation is used for generating and discharging

verification conditions, which are then traded for the correctness theorem by automatically generating a tail-recursive clock. Partial and total correctness are handled uniformly. The method is compositional in that individual procedures can be verified component-wise to prove the correctness of their composition. It also provides a natural treatment of recursive procedures.

The method unifies the clarity and concreteness of operational semantics with the abstraction provided by assertional methods without requiring the implementation (or verification) of a VCG for the target language. To understand why implementing a VCG for a realistic programming language is difficult, consider the method invocation instruction of the JVM. This instruction involves method resolution with respect to the object on which the method is invoked, and side effects on many parts of the states such as the call frames, heap (for synchronized methods), and the class table (for dynamic methods). Encoding such operations as predicate transformation instead of state transformation is non-trivial. Furthermore, most VCGs perform on-the-fly formula simplifications to generate manageable verification conditions. As a measure of the complexity, the VCG for the Java certifying compiler ran to about 23000 lines of C in 2001 [10]. In our approach, only one trusted tool, namely an off-the-shelf theorem prover, is necessary, while still inheriting the benefits of a VCG; the requisite simplifications are performed with the full power of the theorem prover.

Note however, that practical VCGs may implement substantial static analysis on the control flow of the program. For instance, the VCG for ESC/Java performs static analysis to elide assertions from join points of conditionals without incurring exponential case blow-up [29]. To emulate them with a theorem prover, the simplification engine and lemma libraries must be powerful enough to encode such transformations. ACL2 provides a *meta reasoning* facility [42], allowing the user to augment its native simplification heuristics. We are investigating its use to encode the analysis performed by a practical VCG.

We are working on making our ACL2 macro more efficient and applying it to verify high-assurance programs on realistic machine models. A target application is the *verifying compiler* being developed at Galois Connections and Rockwell Collins, Inc. to compile programs in the Cryptol$^{\text{TM}}$ language into code for the AAMP7$^{\text{TM}}$ processor [43]. The goal is to generate, in addition to object code, a proof to certify that the code implements the source program semantics, and our macro can be used with the existing ACL2 model of the AAMP7 [40] to generate the requisite verification conditions.

Acknowledgements This work has been supported in part by DARPA and the NSF under Grant no. CNS-0429591. Eric Smith and Matt Kaufmann provided many insightful suggestions, and Joe Hurd found an error in an earlier draft of this paper. We thank our colleagues at the National Security Agency, Galois Connections Inc., Rockwell Collins Inc., and the University of Texas, for using our ACL2 macro and providing feedback. The anonymous referees also provided substantial comments that have improved the presentation of the paper.

References

1. 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
2. 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
3. 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
4. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12** (1969) 576–583
5. Ray, S., Moore, J.S.: Proof Styles in Operational Semantics. In: FMCAD 2004. LNCS 3312, Springer-Verlag (2004) 67–81
6. Manna, Z.: The Correctness of Programs. JCSS **3** (1969) 119–127
7. Oheimb, D.v., Nipkow, T.: Machine-checking the Java Specification: Proving Type-Safety. In Alves-Foss, J., ed.: Formal Syntax and Semantics of Java. Volume 1523 of LNCS. Springer (1999) 119–156
8. 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
9. Shankar, N.: Machine-Assisted Verification Using Theorem Proving and Model Checking. In Broy, M., Schieder, B., eds.: Mathematical Methods in Program Development. Volume 158 of NATO ASI Series F: Computer and Systems Science. Springer (1997) 499–528
10. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A Certifying Compiler for Java. In: ACM SIGPLAN 2000 conference on Programming language design and implementation. (2000) 95–107
11. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
12. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher Order Logics. Volume 2283 of LNCS. Springer-Verlag (2002)
13. Manolios, P., Moore, J.S.: Partial Functions in ACL2. Journal of Automated Reasoning **31** (2003) 107–127
14. Boyer, R.S., Goldshlag, D., Kaufmann, M., Moore, J.S.: Functional Instantiation in First Order Logic. In Lifschitz, V., ed.: Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press (1991) 7–26
15. Moore, J.S.: Inductive Assertions and Operational Semantics. In Geist, D., ed.: CHARME 2003. Volume 2860 of LNCS., Springer-Verlag (2003) 289–303
16. Slind, K., Hurd, J.: Applications of polytypism in theorem proving. In Basin, D., Wolff, B., eds.: 16th International Conference on Theorem Proving in Higher Order Logics. LNCS 2978 (2003) 103–119
17. Toma, D., Borrione, D.: Formal verification of a SHA-1 circuit core using ACL2. In Hurd, J., Melham, T., eds.: TPHOLS 2005. Springer LNCS 3603 (2005) 326–341
18. Schneier, B.: Applied Cryptography (2nd ed.): Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc. (1995)
19. McCarthy, J.: Towards a Mathematical Science of Computation. In: Proceedings of the Information Processing Congress. Volume 62., North-Holland (1962) 21–28

20. Yu, Y.: Automated Proofs of Object Code for a Widely Used Microprocessor. PhD thesis, University of Texas at Austin (1992)
21. Strecker, M.: Formal Verification of a Java Compiler in Isabelle. In Voronkov, A., ed.: CADE 2004. LNCS 2392, Springer-Verlag (2002) 63–77
22. Hamon, G., Rushby, J.: An Operational Semantics for Stateflow. In: FASE 2004. LNCS 2984, Springer-Verlag (2004) 229–243
23. 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)
24. 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
25. Dijkstra, E.W.: Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. Communications of the ACM **18** (1975) 453–457
26. King, J.C.: A Program Verifier. PhD thesis, Carnegie-Melon University (1969)
27. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended Static Checking for Java. Technical Report 159, Compaq Systems Research Center (1998)
28. King, S., Hammond, J., Chapman, R., Pryor, A.: Is Proof More Cost-Effective Than Testing? IEEE Transactions on Software Engineering **26** (2000) 675–686
29. Flanagan, C., Saxe, J.B.: Avoiding Exponential Explosion: Generating Compact Verification Conditions. In: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages . (2001) 193–205
30. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93** (2005) 281–288
31. Homeier, P., Martin, D.: A Mechanically Verified Verification Condition Generator. The Computer Journal **38** (1995) 131–141
32. Gloess, P.Y.: Imperative Program Verification in PVS. Technical report, École Nationale Supérieure Électronique, Informatique et Radiocommunications de bordeaux (1999)
33. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In Baader, F., Voronkov, A., eds.: LPAR 2004. Volume 3452 of LNAI., Springer (2005) 398–414
34. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (1998)
35. Mehta, F., Nipkow, T.: Proving Pointer Programs in Higher Order Logic. In Baader, F., ed.: CADE 2003. LNAI 2741, Springer-Verlag (2003) 121–135
36. Ortner, V., Schirmer, N.: Verification of bdd normalization. In Hurd, J., Melham, T., eds.: TPHOLS 2005. Springer LNCS 3603 (2005) 261–277
37. Matthews, J., Vroon, D.: Partial Clock Functions in ACL2. In Kaufmann, M., Moore, J.S., eds.: 5th ACL2 Workshop. (2004)
38. Necula, G.C.: Proof-Carrying Code. (In: POPL 1997) 106–119
39. Appel, A.W.: Foundational Proof-Carrying Code. In: LICS 2001. (2001) 247–258
40. Greve, D., Richards, R., Wilding, M.: A Summary of Intrinsic Partitioning Verification. In Kaufmann, M., Moore, J.S., eds.: 5th ACL2 Workshop. (2004)
41. Hardin, D., Smith, E.W., Young, W.D.: A Robust Machine Code Proof Framework for Highly Secure Applications. In Manolios, P., Wilding, M., eds.: 6th ACL2 Workshop. (2006)
42. Hunt Jr., W.A., Kaufmann, M., Krug, R.B., Moore, J.S., Smith, E.W.: Meta Reasoning in ACL2. In Hurd, J., Melham, T., eds.: TPHOLS 2005. Springer LNCS 3603 (2005) 373–384
43. Pike, L., Shields, M., Matthews, J.: A Verifying Core for a Cryptographic Language Compiler. In Manolios, P., Wilding, M., eds.: 6th ACL2 Workshop. (2006)