A Mechanical Analysis of Program Verification Strategies

Sandip Ray (sandip@cs.utexas.edu)
Department of Computer Sciences, The Unversity of Texas at Austin
Austin, TX 78712
http://www.cs.utexas.edu/users/sandip

Warren A. Hunt Jr. (hunt@cs.utexas.edu)
Department of Computer Sciences, The Unversity of Texas at Austin
Austin, TX 78712
http://www.cs.utexas.edu/users/hunt

John Matthews (matthews@galois.com)
Galois Inc.
Beaverton, OR 97005.
http://web.cecs.pdx.edu/∼jmatthew

J Strother Moore (moore@cs.utexas.edu)
Department of Computer Sciences, The Unversity of Texas at Austin
Austin, TX 78712
http://www.cs.utexas.edu/users/moore

January 5, 2008

Abstract.

We analyze three proof strategies commonly used in deductive verification of deterministic sequential programs formalized with operational semantics. The strategies are: (i) *stepwise invariants*, (ii) *clock functions*, and (iii) *inductive assertions*. We show how to formalize the strategies in the logic of the ACL2 theorem prover. Based on our formalization, we prove that each strategy is both *sound* and *complete*. The completeness result implies that given any proof of correctness of a sequential program one can derive a proof in each of the above strategies. The soundness and completeness theorems have been mechanically checked with ACL2.

Keywords: inductive assertions, invariants, partial correctness, theorem proving, total correctness

1. Introduction

Proving the correctness of a program entails showing that if the program is initiated from a machine state satisfying a certain precondition then the state on termination satisfies some desired postcondition. Program verification has arguably been one of the most fertile application areas of formal methods, and there is a rich body of works (Goldstein and von Neumann, 1961; Turing, 1949; McCarthy, 1962; Floyd, 1967; Hoare, 1969; Manna, 1969; Dijkstra, 1975) developing mathematical theories for reasoning about sequential programs. In mechanical

theorem proving, one of the central research thrusts is to mechanize and automate such reasoning (Boyer and Moore, 1979; Gordon and Melham, 1993; Owre et al., 1992; Bertot and Castéran, 2004).

In this paper, we analyze proof strategies for deductive verification of deterministic sequential programs. We consider programs modeled with *operational semantics*. In this approach, the semantics of the programming language is formalized by an interpreter that describes the effect of each instruction on the underlying machine state.[1] We consider three commonly used proof strategies, namely (i) the use of *stepwise invariants*, (ii) the application of *clock functions* (Hunt, 1994; Bevier et al., 1989; Boyer and Moore, 1996), and (iii) reasoning based on *inductive assertions* attached to a program at cutpoints (Floyd, 1967; Hoare, 1969; King, 1969; Manna, 1969). We present formalizations of these strategies in the logic of the ACL2 theorem prover (Kaufmann et al., 2000), and mechanically derive relations between them.

In spite of the obvious importance of mathematical theories underlying program correctness, there has been relatively little research on analyzing the expressive power of such theories.[2] In particular, we have found few published works on mechanical analysis comparing different strategies used in verifying programs based on operational semantics. However, it has been informally believed that the logical guarantees provided by the strategies are different.

Our work shows that such beliefs are flawed. In particular, we prove that each of the three strategies above is both *sound* and *complete*. By *complete* we mean that given any correctness proof of a sequential program in a sufficiently expressive logic, we can derive the proof obligations involved in each of the above strategies. In fact, we show how to derive such obligations *mechanically*. These results hold for both partial and total correctness proofs. The soundness and completeness results have been proven with ACL2, and we have implemented translation routines (as Lisp macros) to translate program proofs between the strategies. The results themselves are not mathematically deep; a careful formalization of the strategies essentially leads to the results.

[1] We use "operational semantics" to denote the semantics derived from the definition of a formal interpreter in a mathematical logic. This terminology is common in program verification, in particular in mechanical theorem proving. However, as pointed out by a reviewer, this usage is somewhat dated among programming language communities, where "operational semantics" now refers to a style of specifying language semantics through inductively defined relations and functions over the program syntax. In current programming language terminology, our approach might more appropriately be referred to as "abstract machine semantics".

[2] One notable exception is the Hoare logic. There has been significant work on analysis of soundness and completeness of Hoare axioms for various programming language constructs. See Section 2.

Nevertheless, our approach clarifies the workings of the different strategies to the practitioners of program verification: without the requisite formalization, it is easy to accept apparently reasonable but flawed notions of program correctness. We discuss this point while explaining the implications of our work in Section 7.

Preliminary versions of some of the results presented in this paper have appeared in previous conference papers of the individual authors. Moore (2003a) constructs stepwise invariants from partial correctness proofs based on inductive assertions. Matthews and Vroon (2004) show how to define a clock function from an assertional termination proof. Ray and Moore (2004) show that stepwise invariants and clock functions are interchangeable for both partial and total correctness proofs. Matthews et al. (2006) construct clock functions from inductive assertions. All the previous results were discovered during our quest for an effective mechanism for applying inductive assertions on operational models. This paper unifies and substantially extends the scope of these previous results, and clarifies the theoretical underpinnings.

The proof strategies were formalized in the logic of the ACL2 theorem prover and all the theorems described here have been mechanically proven with ACL2. ACL2 is a theorem prover for an untyped first-order logic of total recursive functions; the inference rules constitute propositional calculus, equality, instantiation, and well-founded induction up to ϵ_0. The syntax of ACL2 is derived from the prefix-normal syntax of Common Lisp: to denote the application of the function f on argument x, one writes $(f\ x)$ instead of the more traditional $f(x)$. However, in this paper we avoid Lisp and adhere to the latter notation. Our presentation assumes basic familiarity with first-order logic and well-founded induction and no previous exposure to the ACL2 system. The relevant facets of the ACL2 logic are explained in passing. The reader interested in a comprehensive understanding of ACL2 is referred to the ACL2 Home Page (Kaufmann and Moore, 2006), which contains an extensive hypertext documentation together with links and references to numerous books and papers. The use of ACL2 *does* have some influence on the specific details of our formalizations. We model systems and their properties using untyped, total, recursive, first-order functions; we avoid explicit representation of infinite sets: to express "x is a natural number", we use the predicate $\mathit{natp}(x)$ (which is axiomatized in ACL2) instead of the more familiar $x \in \mathbb{N}$. However, the basic results are independent of ACL2 and can be formalized in any theorem prover supporting first-order quantification, Skolemization, and recursive function definitions.

The remainder of the paper is organized as follows. We start in Section 2 with a summary of related research on program correctness

and the use of the strategies formalized here in deductive verification projects. In Section 3, we formalize the notions of partial and total correctness of sequential programs based on operational semantics. In Section 4, we present formalizations of the three proof strategies, and derive a proof of soundness of each strategy. In Section 5, we derive completeness theorems. In Section 6 we briefly comment on the mechanization of the results in ACL2. We discuss some implications of the work in Section 7, and conclude in Section 8.

2. Related Work

In this section, we review related research on program verification. Because of the vastness of the area, we confine ourselves to work that involves the application of the strategies discussed in this paper.

McCarthy (1962) introduced the notion of operational semantics. Operational models have been extensively used in mechanical theorem proving, and are often lauded for clarity and concreteness (Greve et al., 2000). Furthermore, in executable logics, operational semantics facilitates validation of formal models by simulation against concrete design artifacts. Operational models have been particularly successful in the Boyer-Moore class of theorem provers, namely ACL2 and its predecessor, Nqthm (Boyer et al., 1995); in the next paragraph we summarize some key program verification projects in these provers. Operational models have also been used in program verification in HOL (Gordon and Melham, 1993) and PVS (Owre et al., 1992). In HOL, Homeier and Martin (1995) have developed a verification system called Sunrise with a model of a small programming language; Norrish (1998) has developed an operational formalization of C. In Isabelle, Strecker has used an operational semantics to formalize Java and the JVM (Strecker, 2002). Recently Fox (2003) has formalized an operational model of the ARM6 processor ISA in HOL. In PVS, Hamon and Rushby (2004) have used operational models to formalize statechart languages.

The Nqthm and ACL2 theorem provers have extensively used operational semantics for modeling computing systems. One of the significant verification projects with operational semantics in Nqthm involved the "CLI stack" (Bevier et al., 1989), consisting of a netlist model of a microprocessor, FM9001, written in an operationally defined Hardware Description Language called `DUAL-EVAL` (Hunt and Brock, 1992),[3] an operational semantics of its ISA, an assembly language called Piton and

[3] The original stack constituted a microprocessor model called FM8502, a 32-bit version of the 16-bit microprocessor FM8501 (Hunt, 1994). FM9001 was developed later using `DUAL-EVAL`, and the stack ported to this microprocessor.

models of an assembler, loader, and linker for Piton programs (Moore, 1996), a simple operating system (Bevier, 1987), two simple high-level languages (Young, 1988; Flatau, 1992), and simple application programs written in these languages (Wilding, 1993). Each layer of the stack was verified against the semantics of the underlying layer, and the theorems were composed to mechanically prove the correctness of high-level programs executed on the microprocessor model. Another significant project in Nqthm was Yu's (1992) verification of the binary code produced by `gcc` on the Motorola 68020 for subroutines in the Berkeley C string library. Some of the non-trivial operational machine models formalized in ACL2 include (i) the Motorola CAP digital signal processor at the pipelined architectural level and sequential microcode ISA level (Brock and Hunt, 1999), (ii) a pipelined microprocessor with interrupts and speculative execution (Sawada and Hunt, 1997), (iii) the Rockwell Collins AAMP7$^{\mathrm{TM}}$ microprocessors, and (iv) a significant subset of the JVM and its bytecode verifier (Liu and Moore, 2005).

In Nqthm and ACL2, program verification has typically involved *clock functions*. In this approach, one defines a function to specify for each machine state the number of steps to termination. Clock functions were involved in code proofs for the CLI stack, and Yu's verification of the C string library mentioned above. At Rockwell Collins Inc., verification of AAMP7$^{\mathrm{TM}}$ programs have used clocks. Recently, clock functions have been used in proofs of JVM bytecodes (Moore, 2003b; Liu and Moore, 2004). The method has been applied in other theorem provers as well, albeit less frequently (Wilding, 1997).

The use of *inductive assertions* involves annotating program points with formulas over program variables such that whenever control reaches an annotated point the associated assertions must hold. This notion was introduced in works of Goldstein and von Neumann (1961), and Turing (1949). These early results involved annotating *all* control points and the assertions thus corresponded essentially to defining a *stepwise invariant* of the program steps. The idea was generalized in papers by Floyd (1967) and Manna (1969), enabling assertions to be attached only at *cutpoints* such as loop tests and program entry and exit. Hoare (1969) introduced *program logics*, namely first-order logic augmented with a set of *Hoare axioms* for specifying the program semantics. The Hoare axioms provide an *axiomatic semantics* of the program, whereby an instruction is used as a predicate transformer rather than a state transformer. For a detailed overview of axiomatic semantics and program logic, see Apt's survey on "Ten Years of Hoare's Logic" (Apt, 1981); in Section 4.3, we provide a brief overview of some relevant aspects of axiomatic semantics.

Assertional reasoning methods have enjoyed much popularity in research on program verification. In practice, the method requires two trusted tools, namely (i) a *verification condition generator* (VCG) that crawls over an annotated program to generate a collection of first-order formulas (called *verification conditions*), and (ii) a theorem prover to discharge these formulas. King (1969) wrote the first mechanized VCG. VCGs have been extensively used in practical program verification projects. Some of the non-trivial projects employing this technology include the Extended Static Checker for Java (Detlefs et al., 1998), the Java Certifying Compiler (Colby et al., 2000), and the Praxis verification of Spark Ada programs (King et al., 2000; Barnes, 2003). VCGs are also the key trusted components in proof-carrying code architectures (Necula, 1997). In theorem proving, the use of assertions with an operational semantics has typically involved implementing a VCG for the target language and verifying it against the operational model. Gloess (1999) formalizes and verifies a VCG in PVS. The Sunrise system cited above includes a verified VCG for the target language in HOL; Schirmer (2005) presents a similar framework in Isabelle. Assertions have been used in Isabelle to verify pointer-manipulation programs (Mehta and Nipkow, 2003) and BDD normalization algorithms (Ortner and Schirmer, 2005). Finally, as mentioned in Section 1, in our previous work (Moore, 2003a; Matthews and Vroon, 2004; Matthews et al., 2006), we built a formal framework in ACL2 to emulate VCG reasoning using symbolic simulation via an operational semantics, without implementing a VCG. We will briefly describe this framework while formalizing assertions in Section 4.3.

Analysis of the expressive power of mathematical theories for program verification has traditionally been confined to Hoare logics. The original Hoare axioms provided proof rules formalizing a few elementary programming constructs such as assignments, conditionals, and loops. The proof rules were extended to incorporate virtually all programming constructs including goto's, coroutines, functions, data structures, and parallelism (Clint and Hoare, 1971; Hoare, 1972; Clint, 1973; Oppen and Cook, 1975; Owicki and Gries, 1976). Soundness of axiomatic semantics has often been established by appealing to an operational model of the underlying language. De Bakker (1980) provides a comprehensive treatment of the issues involved in developing axiomatic semantics of different programming constructs. The soundness theorems for these different flavors of Hoare logic were usually established by careful mathematical arguments, but the arguments were rarely mechanized in a theorem prover. One notable exception is Harrison's (1998) formalization of Dijkstra's (1978) monograph "A Discipliine of Programming" in HOL. In addition, the VCG verification work cited above has of-

ten involved mechanically checking that the formula transformations implemented by the VCG are sound with respect to the operational semantics.

In addition to soundness, there has been significant research analyzing completeness of the different proof rules for axiomatic semantics. Clarke (1976) provides a comprehensive treatment of completeness (and incompleteness) results for Hoare-like axiomatic proof systems; in addition, Apt's survey (1981) provides an interesting analysis of both soundness and completeness results. Cook (1978) presents a sound and complete assertional system for certain classes of programs. Wand (1978) shows that the axiomatic formulation of certain constructs are incomplete. Sokolowski (1977) proves the completeness of a set of axioms for total correctness.

3. Operational Semantics and Program Correctness

The use of operational semantics involves modeling the instructions in a program by specifying their effects on the state of the underlying machine. To formalize a programmming language operationally in a mathematical logic, one first models the machine states as objects in the logic. A machine state is represented as a tuple that specifies the values of all the machine components such as the registers, memory, stack, etc. The components include the program counter (pc) and the program to be executed. These two components identify the "current instruction" at any state s, which is the instruction in the program that is pointed to by the pc. The meaning of a program is specified by defining a *next state function* step: for any state s, $\mathit{step}(s)$ returns the state s' obtained by executing the current instruction in s. For instance, if the instruction is `LOAD` then s' might be obtained from s by pushing some component of s on the operand stack and advancing the pc by an appropriate amount. The function step can thus be viewed as a formal interpreter for the programming language in question.

Given step, one specifies machine executions by the function run below, which returns the state after stepping for n instructions from s. Recall that $\mathit{natp}(x)$ holds if and only if x is a natural number.

$$\mathit{run}(s,n) \triangleq \begin{cases} \mathit{run}(\mathit{step}(s), n-1) & \mathit{if}\ \mathit{natp}(n) \wedge (n > 0) \\ s & \text{otherwise} \end{cases}$$

Correctness of programs is specified with three predicates pre, post, and exit on the machine states, with the following associated meanings.

– Predicates pre and post represent the precondition and postcondition. Thus if the program of interest is a sorting program then

pre(s) might posit that some component of s contains a list l of numbers and *post*(s) might say that some (possibly the same) component contains a list of numbers that is an ordered permutation of l.

– The predicate *exit* recognizes the "terminal" states. The definition of *exit* depends on the program being verified. While analyzing *halting* programs we define *exit* to recognize the halted states of the machine: *exit*$(s) \triangleq (\mathit{step}(s) = s)$. More commonly, we analyze a *program component* and *exit* is defined to recognize the return of control from that component. For instance, if the component is a subroutine then *exit* might recognize the popping of the current call frame from the call stack.

Formally there are two notions of correctness, *partial* and *total*. Partial correctness entails showing that if, starting from a state s satisfying *pre*, the machine reaches an *exit* state, then *post* holds for the first such *exit* state. No obligation is involved if no *exit* state is reachable from s; that is, a non-terminating program is considered partially correct. Partial correctness can be expressed by the following formula.

Partial Correctness:

$$\begin{aligned}&\forall s, n : \mathit{pre}(s) \wedge \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)) \\ &\qquad \wedge (\forall m : \mathit{natp}(m) \wedge (m < n) \Rightarrow \neg \mathit{exit}(\mathit{run}(s, m))) \\ &\quad \Rightarrow \mathit{post}(\mathit{run}(s, n))\end{aligned}$$

Total correctness entails showing both partial correctness and *termination*, that is, some *exit* state is reachable from each state satisfying *pre*. Termination can be expressed by the following formula.

Termination:

$$\forall s : \mathit{pre}(s) \Rightarrow (\exists n : \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)))$$

Given an operational semantics specified by *step*, together with predicates *pre*, *post*, and *exit*, verifying a program amounts to proving **Partial Correctness** and **Termination**. We now make a brief remark on the formalization of these obligations in ACL2. Note that both the obligations involve arbitrary first-order quantification. The syntax of the ACL2 logic is quantifier-free, and all formulas are assumed to be implicitly universally quantified over all free variables. However, ACL2 provides a logical construct (called the *defchoose principle*) to enable expression of quantified formulas via Skolemization. Suppose that P is a binary predicate definable in some ACL2 theory $\mathcal{T}$. The defchoose principle allows us to extend $\mathcal{T}$ by introducing a function *exists-P-witness* with the following axiom:

$$\forall x, y : P(x, y) \Rightarrow P(x, \mathit{exists\text{-}P\text{-}witness}(x))$$

The function *exists-P-witness* is thus simply a Skolem function, and the principle is essentially a rendering of Hilbert choice in the ACL2 logic. ACL2 provides a macro called `defun-sk` to conveniently define quantified formulas via Skolemization. For instance `defun-sk` allows us to introduce the predicate *exists-P*(x) that holds if and only if there exists a y such that $P(x, y)$ by first introducing *exists-P-witness* above and then defining *exists-P* in terms of the witness as follows.

$$\textit{exists-P}(x) \triangleq P(x, \textit{exists-P-witness}(x))$$

The macro also supports universally quantified predicates by exploiting the duality between universal and existential quantification.

Quantification and Skolemization are crucial to the results of this paper. In particular, we make critical use of the fact that whenever a quantified predicate is defined, the logic introduces the corresponding Skolem witness which can be used in subsequent definitions.

4. Proof Strategies

We now discuss three strategies commonly used for proving the correctness of operationally modeled sequential programs. The strategies are (i) *stepwise invariants*, (ii) *clock functions*, and (iii) *inductive assertions*. We also provide a brief sketch of the proof of soundness for each strategy; completeness proofs are presented in the next section.

The soundness of a proof strategy entails showing that if the corresponding proof obligations are met then one can derive the correctness statements. Since each strategy has been extensively used for mechanized program verification such results are not surprising; we discuss the proofs mainly to provide a flavor of the reasoning involved in doing them formally. We have carried out the soundness proofs in ACL2 both to validate our formalization of the correctness statements and to facilitate implementation of macros for switching strategies. (See Section 6.)

Each strategy can be used to prove the correctness theorems; however, the proof obligations involved are different. To illustrate the methods, we use the simple program fragment shown in Figure 1. The program assigns two variables `X` and `Y` the values 0 and 10 respectively, and then executes a loop in which `X` is incremented and `Y` is decremented by 1 at each iteration; the loop terminates when `Y` is 0. We assume (i) the underlying machine state contains components `X`, `Y`, and pc, and for any state s the values of these components are given by $X(s)$, $Y(s)$, and *pc*(s) respectively, (ii) the language constructs, namely assignment, branching, and arithmetic, are encoded in the definition of the function

```
1:  X:=0
2:  Y:=10
3:  IF (Y == 0) goto 7;
4:  X:=X+1;
5:  Y:=Y-1;
6:  goto 3;
7:  ...
```

Figure 1. A Simple One-Loop Program Fragment. The number to the left of each instruction represents the corresponding pc value.

step, and (iii) the transitions given by *step* correspond to executing one instruction. The precondition and postconditions and the *exit* predicate are defined below. Here *loaded*(s) holds if s has the program fragment loaded in memory from location 1.

- *pre*$(s) \triangleq (\mathit{pc}(s) = 1 \wedge \mathit{loaded}(s))$
- *exit*$(s) \triangleq (\mathit{pc}(s) = 7)$
- *post*$(s) \triangleq (X(s) = 10)$

4.1. Stepwise Invariants

The method of *stepwise invariants* represents one approach to proving program correctness. To prove partial correctness in this method, one defines a predicate *inv* so that the following three formulas are theorems:

I1: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{inv}(s)$

I2: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{inv}(\mathit{step}(s))$

I3: $\forall s : \mathit{inv}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

Partial correctness follows from **I1**-**I3**. To prove this, we must show that for each state s from which some *exit* state is reachable, the first reachable *exit* state from s satisfies *post*. The key lemmas for the proof are shown in Figure 2. The lemmas are interpreted as follows. Let s be an arbitrary non-*exit* state satisfying *inv*, and n be a natural number such that (i) there is no *exit* state reachable from s in n or fewer steps, and (ii) the state s' reachable from s in $(n + 1)$ steps satisfies *exit*. Consider the sequence $\langle s, \mathit{step}(s), \ldots, s' \rangle$. Each non-*exit* state in the sequence satisfies *inv* (Lemma `inv-for-internal`), and in addition, the

$$\mathit{non\text{-}exit}(s,n) \triangleq \begin{cases} \neg \mathit{exit}(s) & \mathit{if}\ \neg \mathit{natp}(n) \vee (n = 0) \\ \neg \mathit{exit}(\mathit{run}(s,n)) \wedge \mathit{non\text{-}exit}(s, n-1) & \mathit{otherwise} \end{cases}$$

Lemma `inv-for-internal`
$\forall s, n : \mathit{inv}(s) \wedge \mathit{non\text{-}exit}(s,n) \Rightarrow \mathit{inv}(\mathit{run}(s,n))$

Lemma `inv-for-first-exit`
$\forall s, n : \mathit{inv}(s) \wedge \mathit{natp}(n) \wedge \mathit{non\text{-}exit}(s,n) \wedge \mathit{exit}(\mathit{run}(s,n+1)) \Rightarrow \mathit{inv}(\mathit{run}(s,n+1))$

Figure 2. Key Lemmas in the Proof of Soundness of Stepwise invariants. Following ACL2, we use `T` and `NIL` for logical true and false, and also specify formalized lemmas by names rather than numbers. Lemma `inv-for-internal` can be proven using **I2** by induction on n. Lemma `inv-for-first-exit` follows from Lemma `inv-for-internal`, **I2** and the fact that if $\mathit{non\text{-}exit}(s,n)$ holds then $\neg \mathit{exit}(\mathit{run}(s,n))$.

first *exit* state s' satisfies *inv* as well (Lemma `inv-for-first-exit`). Then by **I1** and **I3**, for each non-*exit* state s satisfying *pre*, s' satisfies *post*. Finally, if s is an *exit* state that satisfies *pre*, then by **I1** and **I3** s (which is the first reachable *exit* state from s) also must satisfy *post*. □

Total correctness additionally requires a well-foundedness restriction. From set theory, a *well-founded structure* is a pair $\langle O, \prec \rangle$ where O is a set and $\prec$ is a binary relation such that there is no infinite decreasing chain in O with respect to $\prec$ (Church and Kleene, 1937). To prove total correctness, one defines a function *measure* (called a *ranking function*) such that the following two formulas are theorems. Here $\mathit{o\text{-}p}(x)$ holds if and only if x is a member of O where $\langle O, \prec \rangle$ is a well-founded structure.

I4: $\forall s : \mathit{inv}(s) \Rightarrow \mathit{o\text{-}p}(\mathit{measure}(s))$

I5: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{measure}(\mathit{step}(s)) \prec \mathit{measure}(s)$

Total correctness follows from **I1**-**I5**. To prove termination, assume by way of contradition that no *exit* state is reachable from a state s satisfying *pre*. By **I1** and **I2**, each state in the sequence $\langle s, \mathit{step}(s), \ldots \rangle$ satisfies *inv*. Thus, by **I4** and **I5**, the sequence forms an infinite decreasing chain on O with respect to the relation $\prec$, violating well-foundedness. □

We take O to be the set of ordinals below ϵ_0, which is axiomatized in ACL2. Ordinals in ACL2 are represented in Cantor Normal Form (Manolios and Vroon, 2003), and there are two predicates, (i) *o-p*

$$\mathit{inv}(s) \triangleq \begin{cases} \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 1 \\ X(s) = 0 \wedge \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 2 \\ X(s) + Y(s) = 10 \wedge \mathit{natp}(Y(s)) \wedge \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 3 \\ X(s) + Y(s) = 10 \wedge \mathit{posp}(Y(s)) \wedge \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 4 \\ X(s) + Y(s) = 11 \wedge \mathit{posp}(Y(s)) \wedge \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 5 \\ X(s) + Y(s) = 10 \wedge \mathit{natp}(Y(s)) \wedge \mathit{loaded}(s) & \text{if } \mathit{pc}(s) = 6 \\ X(s) = 10 & \text{if } \mathit{pc}(s) = 7 \\ \texttt{NIL} & \text{otherwise} \end{cases}$$

Figure 3. A Single-step Invariant for the One-Loop Program. Here `NIL` denotes logical false, and the predicate *posp* is defined as: $\mathit{posp}(x) \triangleq \mathit{natp}(x) \wedge (x > 0)$

to recognize (the representation of) ordinals, and (ii) the relation $\prec$ (called `o<`) axiomatized to be an irreflexive total order on ordinals.

A key attraction of stepwise invariants is that none of the proof obligations requires reasoning about more than a single step. Nevertheless, coming up with an appropriate definition of *inv* is widely acknowledged to be complicated and tedious (Shankar, 1997; Manna and Pnueli, 1995). To illustrate the difficulty, consider proving partial correctness of the program in Figure 1. To satisfy **I2**, *inv* must characterize every possible pc value: to infer that when the control is at pc value 7 the value of `X` is 10, we must know that at pc value 3 the sum `X+Y` must be 10 and `X` and `Y` are both at least 0. Figure 3 shows one possible definition of *inv*.

The complexity of the stepwise invariants approach increases substantially for a practical machine model such as a model of the JVM. For instance, method invocation in the JVM must deal with method resolution with respect to the object on which the method is invoked, as well as effects on many components of the machine state including the call frames of the caller and the callee, the thread table, the heap, and the class table; defining *inv* to hold along a transition involving such operations is non-trivial. Finally, for total correctness proofs, defining the function *measure* involves analogous complications.

4.2. Clock Functions

Clock functions provide another method for proving program correctness. In this method, one defines a function mapping each state s to a natural number which specifies the number of transitions to reach

the first *exit* state from s. For proving total correctness, we define the function *clock* so that the following formulas are theorems:

C1: $\forall s:\ \mathit{pre}(s) \Rightarrow \mathit{natp}(\mathit{clock}(s))$

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

C3: $\forall s:\ \mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

C4: $\forall s:\ \mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

C1-C4 guarantee total correctness. By **C1** and **C3**, for each state s satisfying *pre*, there exists some natural number m, namely $\mathit{clock}(s)$, such that $\mathit{run}(s, m)$ satisfies *exit*. Finally, **C1**, **C2** and **C3** guarantee that the state $\mathit{run}(s, \mathit{clock}(s))$ is the first *exit* state reachable from s, and, by **C4**, this state satisfies *post*. □

To express only partial correctness, we weaken **C1**, **C3** and **C4** to their "primed" versions below, requiring that the obligations be satisfied for a state s only if some *exit* state is reachable from s.

C1′: $\forall s, n:\ \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{natp}(\mathit{clock}(s))$

C3′: $\forall s, n:\ \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

C4′: $\forall s, n:\ \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

Partial correctness follows from **C1′**, **C2**, **C3′**, and **C4′**. This is trivial since even with the weakening, if, for a state s satisfying *pre*, there exists a natural number n such that $\mathit{run}(s, n)$ satisfies *exit*, then there still exists a natural number m, namely $\mathit{clock}(s)$, such that after m steps the first *exit* state is reached and this state satisfies *post*. □

What is complicated about clock functions? Consider total correctness. By **C1-C3**, we know that for each state s satisfying *pre*, $\mathit{clock}(s)$ is the minimum number of transitions to reach the first *exit* state. But this number is the precise characterization of the program's time complexity! On the other hand, total correctness with stepwise invariants did not seem to require characterizing the time complexity although it involved an argument showing that the program eventually terminates.

Consider the use of clock functions to prove total correctness of our running example. We show one possible definition of *clock* in Figure 4. The function is defined by analyzing the control structure of the program as follows. The function $\mathit{lpc}(s)$, which determines the number of instructions executed from a state s poised at the start of loop (pc value 3) until the loop terminates, is either 0 (if the loop is not taken), or the sum of the number of instructions (namely, 4) in one loop iteration

$$
\begin{array}{lcl}
\mathit{lp\text{-}taken}(s) & \triangleq & (\mathit{pc}(s) = 3) \wedge \mathit{natp}(Y(s)) \wedge (Y(s) > 0) \wedge \mathit{loaded}(s) \\
\mathit{lpc}(s) & \triangleq & \begin{cases} 0 & \mathit{if}\ \neg \mathit{lp\text{-}taken}(s) \\ 4 + \mathit{lpc}(\mathit{run}(s, 4)) & \text{otherwise} \end{cases} \\
\mathit{clock}(s) & \triangleq & 2 + \mathit{lpc}(\mathit{run}(s, 2)) + 1
\end{array}
$$

Figure 4. Clock Function for Verifying the One-Loop Program.

together with, recursively, the number of instructions to be executed after one iteration from s until loop termination; finally, *clock*(s) computes the number of instructions executed from program invocation to *exit* by summing (i) the number of instructions from invocation to loop entry, (ii) the number of instructions during loop iteration, and (iii) the number of instructions from loop termination to *exit*. Thus to define *clock* we only focus on the critical machine states (called *cutpoints*) such as those for loop entry, and program invocation and exit rather than each step as needed for stepwise invariants.

The complexity of clock functions in practice involves the use of these definitions to formally prove **C1**-**C4**. Note that the definition of *lpc* above is recursive; thus, we must ensure that the definition is consistent. One way of ensuring consistency is to show that the recursion is well-founded; this amounts to the proof that under the conditions governing recursive calls (namely, when *lp-taken*(s) holds), the value of the Y component of s (a positive natural number if the recursive call is invoked) decreases along each recursive call. This is essentially the argument for the termination of the loop (and hence the program) itself. Furthermore, for verifying correctness of programs with loops, one normally needs to prove lemmas characterizing the updates performed during the execution of the loops; such lemmas need to be proven by induction on the number of loop iterations.

4.3. Inductive Assertions

The third proof strategy we consider is *inductive assertions*. Stepwise invariants require attaching assertions to each program point; clocks require defining a function that counts the number of instructions to the first *exit* state. Inductive assertions attempt to facilitate verification by eliminating the need for both these exercises. Instead, the users annotate a program with assertions at program cutpoints. The goal is to prove that whenever control reaches a cutpoint the associated assertions must hold. To guarantee this, one typically uses a program

$$\mathit{cut}(s) \triangleq ((\mathit{pc}(s) = 1) \vee (\mathit{pc}(s) = 3) \vee (\mathit{pc}(s) = 7))$$

$$\mathit{a}(s) \triangleq \begin{cases} \mathtt{T} & \mathit{if}\ \mathit{pc}(s) = 1 \\ X(s) + Y(s) = 10 \wedge \mathit{natp}(Y(s)) & \mathit{if}\ \mathit{pc}(s) = 3 \\ X(s) = 10 & \mathit{if}\ \mathit{pc}(s) = 7 \\ \mathtt{NIL} & \text{otherwise} \end{cases}$$

$$\mathit{assert}(s) \triangleq \mathit{a}(s) \wedge (((\mathit{pc}(s) = 1) \vee (\mathit{pc}(s) = 3)) \Rightarrow \mathit{loaded}(s))$$

Figure 5. Definitions of *cut* and *assert* for the One-Loop Program.

called a *verification condition generator* (VCG) (King, 1969) in addition to a theorem prover. A VCG "crawls" over the annotated program to generate a collection of formulas (called *verification conditions*) which are checked for validity by the theorem prover. Roughly, the guarantee provided upon successful validity check can be stated as follows. "Let p be any non-*exit* cutpoint satisfying *assert*. Let q be the next cutpoint encountered in an execution from p. Then *assert*(q) must hold." Thus, if in addition (i) each initial state s (that is, those for which *pre*(s) holds) is a cutpoint satisfying *assert*, (ii) each *exit* state is a cutpoint, and (iii) for each *exit* state s *assert*(s) logically implies *post*(s), then the first *exit* state reachable from any initial state must satisfy *post*. Finally, for termination one also needs to associate a well-founded ranking function *rank* with each cutpoint p and show that if q is the next subsequent cutpoint from p then *rank*$(q) \prec$ *rank*(p).

Figure 5 shows the definitions of predicates *cut* (which characterize the set of cutpoints) and *assert* for proving partial correctness of our running example. Ignoring the "boiler-plate" predicate *loaded*(s), the only non-trivial assertion is the loop invariant specified for pc value 3.

We briefly digress here to say a few words about axiomatic semantics in order to relate the above with traditional VCG work. When using assertional methods with a VCG one does not use assertions over machine states but over the *program variables*: instead of writing $X(s)+Y(s) = 10$ as above, we will write $\mathtt{X}+\mathtt{Y} = 10$. A VCG encodes the semantics of the programming language as predicate transformations over program variables. The transformations are written in the form $\{\mathcal{P}\}\sigma\{\mathcal{Q}\}$ (where $\mathcal{P}$ and $\mathcal{Q}$ are predicates over the program variables and σ is a statement in the language) and are read as: "If $\mathcal{P}$ holds before the execution of statement σ then $\mathcal{Q}$ holds afterwards." Such

triples (also called *Hoare triples*) form the *axiomatic semantics* of the language. The following is the axiom of assignment.

– $\{\mathcal{P}\}$`x := a`$\{\mathcal{Q}\}$ holds if $\mathcal{P}$ is obtained by replacing all occurrences of `x` in $\mathcal{Q}$ by `a`.

A VCG explores the control structure of an annotated program using the axiomatic semantics. Each control path leading from one cutpoint to the next produces one verification condition: the path along program counter values $1 \longrightarrow 2 \longrightarrow 3$ will produce the verification condition $\mathtt{T} \Rightarrow (0 + 10) = 10$. These conditions are then discharged by a theorem prover. Note that each loop must contain at least one cutpoint (typically inserted at the loop test) so that the exploration terminates (Floyd, 1967; Manna, 1969).

To formalize inductive assertions, we need the notion of "next cutpoint". For that purpose, we use the following definition:

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

Note that the above definition is partial; if no cutpoint is reachable from s, then the recursion does not terminate and the equation does not specify the value of $\mathit{csteps}(s, i)$. ACL2 normally requires that recursive definitions be terminating. However, the crucial observation here is that the definition is *tail-recursive*. Manolios and Moore (2003) show how such definitions can be consistently introduced in ACL2. In general, tail-recursive definitions can be introduced in a logic that supports recursive definitions and a Skolem choice function; the choice function is used to exhibit a total function that witnesses the definition when the recursion does not terminate. Given the definition above we can prove that if some cutpoint is reachable from s in j steps then $\mathit{csteps}(s, i)$ returns $(i + j)$. We postpone the proof of this statement to the next section, where (in Figure 6) we present an analogous theorem about a similarly defined tail-recursive function.

Using *csteps*, we now formalize the notion of the next reachable cutpoint as follows. Fix a state $\mathbf{d}$ such that $\mathit{cut}(\mathbf{d}) \Leftrightarrow (\forall s : \mathit{cut}(s))$; the state $\mathbf{d}$ can be defined using defchoose. Then $\mathit{nextc}(s)$ returns the first reachable cutpoint from s if any, else $\mathbf{d}$:

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

We now formalize verification conditions. Conditions **V1**-**V5** specify the obligations for partial correctness. 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: $\forall s, n : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{step}(s)))$

Partial correctness follows from **V1**-**V5**. The proof is analogous to that for stepwise invariants, but here we focus only on cutpoints. First observe that if s is not an *exit* state and some *exit* state is reachable from s then, by **V3** and the definition of *nextc*, some *exit* state is reachable from $\mathit{nextc}(\mathit{step}(s))$. Now, by **V1**, the initial states (those satisfying *pre*) satisfy *assert*. By **V5** and the observation above, it follows that if a state s satisfies *assert*, it follows that every reachable cutpoint from s up to (and, by **V3**, including) the first reachable *exit* state s' satisfies *assert*. Then from **V4**, we infer that s' satisfies *post*. □

For total correctness, we additionally attach a well-founded ranking function at cutpoints. This is formalized by the additional proof obligations **V6** and **V7** below, together with the necessary strengthening of **V5** to **V5′**. The strengthening ensures (from **V2**) that for each cutpoint p there *does* exist a subsequent next cutpoint satisfying *assert*.

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

V6: $\forall s : \mathit{assert}(s) \Rightarrow \mathit{o\text{-}p}(\mathit{rank}(s))$

V7: $\forall s : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{nextc}(\mathit{step}(s))) \prec \mathit{rank}(s)$

Total Correctness follows from **V1**-**V4**, **V5′**, **V6**-**V7**. It suffices to show that some *exit* state is reachable from each cutpoint p. By **V2** and **V5′**, for each cutpoint p, the state $\mathit{nextc}(\mathit{step}(p))$ is a subsequent cutpoint reachable from p. But by well-foundedness, **V6**, and **V7**, one of these cutpoints must be an *exit* state, proving the claim. □

Note that it is possible to have minor variations of the above verification conditions as follows. (1) We can remove the restriction that the cutpoints include initial states by refining **V1** to be $\mathit{pre}(s) \Rightarrow \mathit{assert}(\mathit{nextc}(s)$. (2) We can remove **V2** and use the conjunct $\mathit{cut}(s) \wedge \mathit{assert}(s)$ instead of $\mathit{assert}(s)$ in the antecedents of **V4**, **V5**, **V5′** **V6**, and **V7**. (3) We can remove **V3** and use the disjunct $\mathit{cut}(s) \vee \mathit{exit}(s)$ instead of $\mathit{cut}(s)$ in **V2**. We ignore such variations in the rest of this presentation.

Conditions **V5** (resp., **V5′**) and **V7** might involve multiple transitions. For instance, **V5** says that if a non-*exit* cutpoint satisfies the

assertions then so does the next subsequent cutpoint. Traditionally, a VCG simplifies these conditions to a first-order proof obligation as suggested by our trivial example. However, in our previous work (Matthews et al., 2006), we have built a framework in which a theorem prover (ACL2) itself can be configured to discharge them by symbolic simulation. The basic idea is to prove the following two theorems, which are both trivial:

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 treat **SSR1** and **SSR2** as conditional rewrite rules. For a symbolic state s, they rewrite the term $\mathit{nextc}(s)$ to either s or $\mathit{nextc}(\mathit{step}(s))$; in the latter case we expand the definition of *step*, perform possible simplifications, and apply the rules again on the resulting term. An attempt to prove **V5** thus causes ACL2 to symbolically simulate the program from each cutpoint satisfying *assert* until another cutpoint is reached, followed by checking if the new state also satisies *assert*. The process mimics a forward VCG, but generates (and discharges) verification conditions on a case-by-case basis.

The formalization of inductive assertions has strong parallels with stepwise invariants. One can view the assertions as stepwise invariants over a *big step semantics*, where a big step transits from one cutpoint to the next. Note that there is one minor difference between the two approaches in the "persistence condition" for partial correctness (namely, **I2** vs. **V5**). **I2** requires that for *each* non-*exit* state s satisfying *inv*, $\mathit{step}(s)$ must satisfy *inv*; for a cutpoint s satisfying *assert*, **V5** requires the next subsequent cutpoint to also satisfy *assert* only if some *exit* state is reachable from s. The difference reflects the disparate design choices involved in formalizing each method. For stepwise invariants, the goal is to develop obligations that do not involve more than a single machine step. However, for inductive assertions, the above weakening is *logically necessary* to support composition. Let P be a procedure calling a subroutine Q, and assume that Q has been proven partially correct. Then, on encountering an invocation of Q during the verification of P we want symbolic simulation to skip past Q, inferring its postcondition at its exitpoint. However, since the correctness proof of Q is *partial* we can only justify the inference under the hypothesis that Q eventually reaches an exitpoint. We can then use the antecedent "some exitpoint of P is reachable" (by noting that when a caller exits, all its callees also exit) from the weakened obligation for P to relieve this hypothesis and continue symbolic simulation past the exitpoint of Q.

There is an important difference between our formalization and a traditional VCG approach. Recall that the assertions in axiomatic se-

mantics involve *program variables*. However, since our approach is based on an operational semantics, we permit any predicate on the *machine state* that is expressible in the logic. Furthermore, we can freely use arbitrary quantified first-order formulas; the only practical requirement is that there are enough theorems about the function and predicate symbols involved to ascertain the validity of the assertions at any (symbolic) state encountered during symbolic simulation. For instance, in our framework, we have used predicates such as the following:

$$\mathit{excut}(s) \triangleq (\exists n : \mathit{cut}(\mathit{run}(s, n)))$$

Although the predicate is not computable, ACL2 can "propagate" it through any symbolic state with this assertion attached, by using the following theorems as rewrite rules.

$$\begin{array}{lll} \forall s: & \mathit{cut}(s) & \Rightarrow \ \mathit{excut}(s) = \mathtt{T} \\ \forall s: & \neg\mathit{cut}(s) & \Rightarrow \ \mathit{excut}(s) = \mathit{excut}(\mathit{step}(s)) \end{array}$$

5. Completeness of Proof Strategies

We now turn our attention to proving that each of the three proof strategies above is complete. By completeness, we mean that given any proof of correctness of a program we can mechanically construct a corresponding proof in any of the strategies. The completeness results do not depend on the structure or style of the alleged original proof; given that the obligation **Partial Correctness** is a theorem (together with **Termination** for total correctness proofs) for some operational model defined by *step* and corresponding predicates *pre*, *post*, and *exit*, we define appropriate functions and predicates that meet the obligations necessary for each strategy.

We start with *clock functions*. The function *clock* below satisfies the relevant obligations of a clock function proof.

$$\begin{array}{lcl} \mathit{esteps}(s, i) & \triangleq & \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{esteps}(\mathit{step}(s), i+1) & \text{otherwise} \end{cases} \\ \mathit{clock}(s) & \triangleq & \mathit{esteps}(s, 0) \end{array}$$

Assuming that **Partial Correctness** is provable, we now show that the conditions **C1′**, **C2**, **C3′**, and **C4′** are provable for this definition. The key lemma `esteps-characterization` is shown in Figure 6. The lemma can be interpreted as follows. Let s be an arbitrary state and i be a natural number, and assume that there is some *exit* state reachable

Lemma `esteps-characterization`

$$\forall s, n, i : \mathit{exit}(\mathit{run}(s, n)) \wedge \mathit{natp}(i) \Rightarrow$$
$$\textbf{let rslt} \leftarrow \mathit{esteps}(s, i) \textbf{ in}$$
$$\textbf{let stps} \leftarrow (\textbf{rslt} - i) \textbf{ in}$$
$$\mathit{natp}(\textbf{rslt}) \wedge$$
$$\mathit{natp}(\textbf{stps}) \wedge$$
$$\mathit{exit}(\mathit{run}(s, \textbf{stps})) \wedge$$
$$(\mathit{natp}(n) \Rightarrow (\textbf{stps} \leq n))$$

Figure 6. Key lemma about *esteps*. Lemma `esteps-characterization` is proven by induction on n. For the base case, note that if $\neg \mathit{natp}(n) \vee (n = 0)$ holds then the lemma is trivial. For the induction hypothesis we assume the instance of the formula under the substitution $[s \leftarrow \mathit{step}(s), n \leftarrow (n-1), i \leftarrow (i+1)]$. The induction step follows from the definition of *esteps*.

from s in n steps. Let (i) **rslt** be the value returned by $\mathit{esteps}(s, i)$, and (ii) **stps** be the difference $(\textbf{rslt} - i)$. Then **rslt** and **stps** are natural numbers, executing the machine for **stps** times from s results in an *exit* state, and **stps** is less or equal to n. Thus, if there is some n such that $\mathit{run}(s, n)$ is an *exit* state then $\mathit{clock}(s)$ returns a natural number that counts the number of steps to the first reachable *exit* state from s, satisfying **C1′**, **C2**, and **C3′**. Finally, from **Partial Correctness**, if s satisfies *pre*, then $\mathit{run}(s, \mathit{clock}(s))$ must satisfy *post*, proving **C4′**. □

The reader familiar with ACL2 will note that since *esteps* has been defined with tail-recursion (and might not always terminate), there can be no induction scheme associated with the recursive structure of this definition. The proof of Lemma `esteps-characterization` requires an explicit induction scheme with the extra parameter n, namely the number of steps to reach some *exit* state from s if such a state exists.

Finally, we now prove the stronger total correctness obligations **C1**, **C3**, and **C4** as follows assuming that both **Partial Correctness** and **Termination** are provable. Let $n(s)$ be the Skolem witness for the existential predicate in the **Termination** formula. By **Termination** and the properties of Skolemization, we know that for a state s satisfying *pre*, $\mathit{run}(s, n(s))$ satisfies *exit*. The obligations **C1**, **C3**, and **C4** now follow by instantiating the variable n in **C1′**, **C3′**, and **C4′** with $n(s)$. □

We now consider *stepwise invariants*. From the results above, we can assume without loss of generality that the correctness proof has been translated to a proof involving clock functions using the definitions of

esteps and *clock*. We define the relevant invariant *inv* below.

$$\begin{aligned} \mathit{inv}(s) \triangleq\ & (\exists p, m :\ \mathit{pre}(p) \wedge \\ & \mathit{natp}(m) \wedge \\ & (s = \mathit{run}(p, m)) \wedge \\ & ((\exists \alpha : \mathit{exit}(\mathit{run}(p, \alpha))) \Rightarrow (m \leq \mathit{clock}(p)))) \end{aligned}$$

The definition can be read as follows. A state s satisfies *inv* if s is reachable from some *pre* state p and the path from p to s contains no *exit* state, except perhaps for s itself.

We now derive **I1**-**I3** from the definition of *inv*. For **I1** note that given a state s satisfying *pre* we can prove $\mathit{inv}(s)$ by choosing the existentially quantified variables p and m to be s and 0 respectively. For **I3**, let s be an *exit* state satisfying *inv* and let $p(s)$ and $m(s)$ be the Skolem witnesses for p and m respectively. Then by Lemma `esteps-characterization` and definitions of *inv* and *clock*, $p(s)$ satisfies *pre* and s is the first *exit* state reachable from $p(s)$; **I3** follows from **Partial Correctness**. Finally, to prove **I2**, we assume that *inv* holds for some non-*exit* state s, and show that *inv* holds for $\mathit{step}(s)$. For this, we must determine a state q satisfying *pre* and a natural number n such that $\mathit{step}(s) = \mathit{run}(q, n)$; furthermore, if there is an *exit* state reachable from $\mathit{step}(s)$ then $n \leq \mathit{clock}(q)$. Let $p(s)$ and $m(s)$ be the Skolem witnesses for $\mathit{inv}(s)$ as above. Then we choose q to be $p(s)$ and n to be $(m(s) + 1)$. Note that by the definition of *inv*, we have $s = \mathit{run}(p(s), m(s))$; thus by definition of *run*, $\mathit{step}(s) = \mathit{run}(p(s), m(s)+1)$. Finally, if some *exit* state is reachable from $\mathit{step}(s)$ then it is also reachable from s. Since s is not an *exit* state, by Lemma `esteps-characterization` and definition of *clock*, we know (i) $\mathit{natp}(\mathit{clock}(p(s)))$ and (ii) $m(s) < \mathit{clock}(p(s))$. Thus $(m(s) + 1) \leq \mathit{clock}(p(s))$, proving **I2**. □

For total correctness, we define the following *measure*:

$$\mathit{measure}(s) \triangleq \mathit{clock}(s)$$

We can now prove **I4** and **I5**. Let s be an arbitrary state satisfying *inv*. Then as above, s must be reachable from some state $p(s)$ satisfying *pre*, and there is no *exit* state in the path from $p(s)$ to s. By **Termination**, some *exit* state is reachable from $p(s)$; thus, some *exit* state is reachable from s. Now, by Lemma `esteps-characterization` and definition of *clock*, $\mathit{clock}(s)$ is a natural number (and hence an ordinal), proving **I4**. Furthermore, for *any* state s, $\mathit{clock}(s)$ gives the number of transitions to the first reachable *exit* state. Thus if s is not an *exit* state

and an *exit* state is reachable from s (by *inv* and **Termination**), then $\mathit{clock}(\mathit{step}(s)) < \mathit{clock}(s)$, proving **I5**. □

We now consider inductive assertions. Obviously, this strategy would reduce to stepwise invariants if each state were a cutpoint. However, our goal is to attach assertions (and ranking functions) to a collection of cutpoints *given a priori*. We use the following definitions for *assert* and *rank*. Here *inv* is the same predicate that we used in proving completeness of stepwise invariants above. Note that the definition of *assert* contains a case split to account for **V2**.

$$\begin{array}{lcl} \mathit{assert}(s) & \triangleq & \begin{cases} \mathit{inv}(s) & \text{if } \mathit{cut}(s) \\ \texttt{NIL} & \text{otherwise} \end{cases} \\ \mathit{rank}(s) & \triangleq & \mathit{clock}(s) \end{array}$$

The proof of completeness of assertional reasoning is analogous to that for stepwise invariants. The only non-trivial lemma necessary for partial correctness requires establishing the followng. "Suppose that a non-*exit* cutpoint s satisfies *assert* and let s' be $\mathit{nextc}(\mathit{step}(s))$. Then if some *exit* state is reachable from s' there exists a state p and a natural number m such that $s' = \mathit{run}(p, m)$ and $m \leq \mathit{clock}(p)$." We exhibit such p and m as follows. Assume $p(s)$ and $m(s)$ to be the Skolem witnesses for $\mathit{inv}(s)$ as in the proof of completeness of stepwise invariants; note from above that *assert* is defined in terms of *inv*. We then take p to be $p(s)$ and m to be $(m(s) + 1 + \mathit{csteps}(\mathit{step}(s), 0))$. The proof thus reduces to showing $(m(s) + 1 + \mathit{csteps}(\mathit{step}(s), 0)) \leq \mathit{clock}(p(s))$ for each non-*exit* cutpoint s. We prove this by first showing that there is no intermediate cutpoint between s and s'; this follows from the definition of *csteps* and is derived by induction analogous to Lemma `esteps-characterization` in Figure 6 but using *csteps* instead of *esteps*. Thus since some *exit* state is reachable from s it must also be reachable from s'. The lemma now follows from the definitions of *esteps* and *clock*. Finally, for total correctness, we note that since there is no intermediate cutpoint between s and s' the number of steps to the first *exit* state (which is what the function *clock* counts) must be less for s' than for s. □

Our results above establish that if one can prove the correctness of an operationally formalized program *in any manner*, then one can mechanically derive the proof obligations of each strategy. However, the results should be regarded with a caveat. They do not imply that in practice one technique might not be easier than the other. For instance, manually writing a stepwise invariant for each pc value *is* more tedious than attaching assertions only at cutpoints. Also, the functions and predicates that we used to prove the completeness theorems might not

be directly used to verify a program from scratch. For instance, the *clock* function we used essentially runs the program until some *exit* state is reached; as we saw in Section 4.2, using a clock function in practice requires a careful reflection of the control structure so that properties about loops can be proven by induction. However, our results perhaps *do* indicate that the difficulty in practical code proofs stems from the inherent complexity in reasoning about complex computing systems rather than the nuances of a particular proof style.

6. Remarks on Mechanization

The proofs of soundness and completeness discussed above are independent of the details of the operational model (that is, the formal definition of *step*), the precondition, and the postcondition. In ACL2, we carry out the reasoning in the abstract, essentially formalizing the proof sketches outlined in the last two sections. The relevant abstraction is achieved using *encapsulation*.

ACL2 allows axiomatization of new function symbols via so-called *extension principles*. The most common extension principle is the *definitional principle* which permits introduction of functions with total (recursive) definitions; the return value of such a function is specified for *all* possible values of its arguments. In contrast, the *encapsulation principle* introduces functions axiomatized only to satisfy some constraints; for example, we can introduce a unary function *foo* axiomatized only to return a natural number. For consistency, the user must exhibit some function (called the *witness*) satisfying the constraints; for *foo*, one witness is the constant function that always returns 1.

Since the only axioms about an encapsulated function are the constraints, any theorem provable about such a function is also provable for other functions satisfying the constraints. More precisely, call the conjunction of the constraints on f the formula ϕ. For any formula ψ let $\hat{\psi}$ be the formula obtained by replacing the function symbol f by the function symbol f'. A derived rule of inference called *functional instantiation* specifies that for any theorem θ one can derive the theorem $\hat{\theta}$ provided one can prove $\hat{\phi}$ as a theorem (Boyer et al., 1991). In the example of *foo* above, if we can prove $\forall x : \mathit{bar}(\mathit{foo}(x))$ for some predicate *bar* and if h is any function that provably returns a natural number, then functional instantiation can be used to derive $\mathit{bar}(h(x))$.

We use encapsulation to reason about proof strategies as follows. Consider the soundness theorem for the partial correctness of stepwise invariants. We encapsulate functions *step*, *pre*, *post*, *exit*, and *inv* to satisfy **I1**-**I3**, and derive the formula **Partial Correctness**. For the

completeness proof we do the opposite, namely encapsulate *pre*, *step*, and *post* constrained to satisfy partial correctness and derive **I1**-**I3**.

We note that the mechanical proofs are not trivial, principally because ACL2 provides limited automation in reasoning about quantified formulas. The proofs therefore require significant manual guidance and care needs to be taken in formalizing concepts such as "the first reachable *exit* state from s". However, the basic structure of the proofs follow the high-level descriptions provided in the preceding two sections.

The generic nature of the proofs enables us to mechanically switch strategies by functional instantiation. For instance, let *step-c*, *pre-c*, *post-c*, *exit-c*, and *inv-c* be the functions involved in a stepwise invariant (partial correctness) proof of some program. We can derive the corresponding clock function proof by the following two steps.

1. Derive **Partial Correctness** by functionally instantiating the soundness theorem for stepwise invariant; the instantiation substitutes the concrete functions *step-c* for *step*, etc. ACL2 must prove that the concrete functions satisfy the constraints. But the constraints are **I1**-**I3** which are what have been proven for the concrete functions.

2. Derive the clock obligations by generating the *clock* described in Section 5, and functionally instantiating the completeness theorem. The constraint is exactly the statement of partial correctness for the concrete functions which has been proven in Step 1.

We have developed a macro in ACL2 to automate the above steps for switching between any two of the strategies. Given a correctness proof for a program in a particular style and a keyword specifying the target strategy, the macro performs the above steps, generating the requisite functions and functionally instantiating the generic theorems.

7. Discussion

The soundness and completeness theorems are not conceptually deep, once the strategies are formalized. The only non-trivial insight in the derivations is the understanding that quantification and Skolemization can be used *in the logic* to define functions characterizing the set of reachable states and the number of steps to termination. In spite of the simplicity, however, several researchers have been surprised when shown the completeness theorems (before seeing the proofs) precisely because of the apparent dissimilarities in the workings of the different proof strategies. For instance, *clock functions* were considered to be significantly different from stepwise invariants and assertional reasoning. Indeed, clock functions had often been criticized before on the

grounds that they require reasoning about efficiency of the program when "merely" a correctness theorem has been desired.[4]

Our work also provides the important foundational basis for building deductive strategies for program verification. In spite of significant research on program correctness, it is still surprisingly easy to create apparently reasonable but flawed proof strategies. To illustrate this, we consider a strategy used by Manolios and Moore (2003) for reasoning sequential programs via symbolic rewriting. The strategy involves defining a function *stepw* as follows via tail-recursion:

$$\mathit{stepw}(s) \triangleq \begin{cases} s & \mathit{if}\ \mathit{halted}(s) \\ \mathit{stepw}(\mathit{step}(s)) & \text{otherwise} \end{cases}$$

The function induces the following equivalence relation $\hookrightarrow$ on states:[5]

$$(s_0 \hookrightarrow s) \triangleq (\mathit{stepw}(s_0) = \mathit{stepw}(s))$$

The relation $\hookrightarrow$ has nice algebraic properties; for instance, the following formula is a theorem:

$$\forall s : s \hookrightarrow \mathit{step}(s)$$

Let $\mathit{modify}(s)$ be the desired modification to s after execution of the program: if the program is a JVM method computing the factorial, then $\mathit{modify}(s)$ might involve popping the current call frame off the call stack and storing the factorial of the argument involved in the invocation at the appropriate local of the call frame of the caller. Manolios and Moore use the following notion to relate s with $\mathit{modify}(s)$.

$$\forall s : \mathit{pre}(s) \Rightarrow (s \hookrightarrow \mathit{modify}(s)) \tag{1}$$

Obligation 1 seems apparently reasonable, and substantial automation can be achieved in proving this statement for operationally modeled sequential programs. In particular, using the theorem $\forall s : s \hookrightarrow \mathit{step}(s)$ and the fact that $\hookrightarrow$ is transitive, the statements can be proven and composed by symbolic simulation. However, does the above theorem imply that the program under inspection is (partially) correct in general? The answer is "no". To see this, first consider a program that never terminates. Such a program is, by definition, partially correct.

[4] This criticism has been rarely written in print, but usually expressed in conference question-answer sessions. However, there has been a nagging feeling that clock functions require more work. The absence of published criticism and the presence of this "nagging feeling" have both been confirmed by an extensive literature search and private communications with some of the authors of other theorem provers.

[5] The actual symbol they use is `==` instead of $\hookrightarrow$. We use the latter to avoid confusion between this relation and equality.

However, note that for any s such that no terminating state is reachable from s, the return value of $\mathit{stepw}(s)$ is not defined by the above defining equation; thus, obligation 1 cannot be established. A stronger objection is that $s \hookrightarrow s'$ may be proved for any two states that reach the same *halted* state; there is no requirement or implication that s' is reachable from s. Consider a pathological machine in which all halting computations collapse to the same final state. Then, for any two states s and s' poised to execute two (perhaps completely different) halting programs, we can prove $s \hookrightarrow s'$. Thus any two halting programs are "equivalent", and any terminating program can be proven "correct" in the sense of being equivalent to any desired halting specification.

The lesson from above is that it is imperative to validate proof strategies against a formal, clearly specified notion of correctness. We should note that the machine used in Manolios and Moore (2003) was not pathological, and the programs analyzed by Manolios and Moore can also be proven correct with respect to our formalization.

Finally, an important consequence of our results is the possibility of mixing strategies for verifying different program components. One strategy might be easier or more intuitive than the other in a practical scenario. For example, consider two procedures: (1) initialization of a Binary Search Tree (BST), and (2) insertion of a sequence of elements in an already initialized BST. Assume that in either case the postcondition specifies that a BST structure is produced. A typical approach for verifying (1) is to define a clock that specifies the number of steps required by the initialization procedure, and then prove the result by symbolic simulation; definition of a stepwise invariant is cumbersome, requiring a precise characterization of the portion of the tree initialized upon execution of each instruction. On the other hand, a stepwise invariant proof might be more natural for verifying (2), by showing that each insertion preserves the tree structure. As a broader example, note that clock functions have been widely used in ACL2 for program verification. On the other hand, as mentioned before, we have developed a compositional verification framework emulating inductive assertions via symbolic simulation; as mentioned in the preceding section, that framework has been used to verify non-trivial programs with substantial automation. With the formal proofs of correspondence and our macros, we can now use that framework while reusing the previous ACL2 proofs based on clock functions.

8. Summary and Conclusion

We have formalized three proof strategies commonly employed in mechanical verification of programs modeled using operational semantics. We have shown how to mechanically derive the proof obligations for each strategy (in a logic allowing first-order quantification and arbitrary tail-recursive definitions) from any proof of correctness of a program. The results hold for both partial and total correctness. We have implemented macros to switch proof strategies in ACL2. We have also illustrated how it is possible, in absence of such a formal framework, to develop apparently reasonable but flawed proof strategies.

We do not advocate one proof strategy over another. Our goal is to enable practitioners working in program verification to go "back and forth" between the different strategies; thus one can focus on reasoning about a program component using the strategy most suitable, independent of other components. This ability is particularly important in theorem proving, where the user needs to guide the theorem prover during a proof search. Our results and macros free the user from adhering to a monolithic strategy for any one program component solely for compatibility with proofs done for other components.

Our results also indicate that the complications in program verification arise not from the specific proof styles used but from the inherent complexity involved in reasoning. Note that if there exists *any* proof of correctness of a program, we can mechanically generate the obligations for each proof strategy. Thus the core creative reasoning involved does not change by simply switching from one strategy to another. On a positive note, this indicates that techniques for automating code proofs in one strategy are also perhaps likely to carry over to the others.

Finally, our work emphasizes the utility of quantification in an inductive theorem prover, in particular ACL2. While ACL2 has always supported full first-order quantification via Skolemization, its expressive power has gone largely unnoticed among the users, the focus being on constructive, recursive definitions. The chief reasons for this focus are that recursive definitions are amenable to efficient executability and play to the strength of the theorem prover in doing automatic well-founded induction. These are certainly useful qualities. However, we have often found it instructive to be able to reason about a generic model of which different concrete systems are instantiations. Quantifiers are useful for reasoning about such generic models. We contend that the principal reason why the connections between the different proof strategies discussed here has gone unnoticed so far in the ACL2 community, in spite of each strategy being used extensively, is the disinclination of the ACL2 users to reason in terms of generic models

and quantified predicates. We have also found quantification useful in other circumstances, for example in formalizing weakest preconditions and in reasoning about pipelined machines (Ray and Hunt, 2004).

Acknowledgements

This material is based upon work partially supported by DARPA and the National Science Foundation under Grant No. CNS-0429591 and by the National Science Foundation under Grant No. ISS-0417413. Matt Kaufmann has provided numerous insights, including a comment that started us on track for the results of this paper in the first place. Additionally, Kaufmann and Robert Krug read an earlier draft of the paper and made several useful suggestions. Larry Paulson confirmed the previously existing nagging doubts about the complexity of *clock function* proofs. Finally, the anonymous referees provided several expository comments that have improved the quality of this article.

References

Apt, K. R.: 1981, 'Ten Years of Hoare's Logic: A Survey — Part I'. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)* **3**(4), 431–483.

Barnes, J.: 2003, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley.

Bertot, Y. and P. Castéran: 2004, *Interactive Theorem Proving and Program Development*. Springer-Verlag.

Bevier, W. R.: 1987, 'A Verified Operating System Kernel'. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin.

Bevier, W. R., W. A. Hunt, Jr., J. S. Moore, and W. D. Young: 1989, 'An Approach to System Verification'. *Journal of Automated Reasoning* **5**(4), 411–428.

Boyer, R. S., D. Goldshlag, M. Kaufmann, and J. S. Moore: 1991, 'Functional Instantiation in First Order Logic'. In: V. Lifschitz (ed.): *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. pp. 7–26.

Boyer, R. S., M. Kaufmann, and J. S. Moore: 1995, 'The Boyer-Moore Theorem Prover and Its Interactive Enhancement'. *Computers and Mathematics with Applications* **29**(2), 27–62.

Boyer, R. S. and J. S. Moore: 1979, *A Computational Logic*. New York, NY: Academic Press.

Boyer, R. S. and J. S. Moore: 1996, 'Mechanized Formal Reasoning about Programs and Computing Machines'. In: R. Veroff (ed.): *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. pp. 141–176.

Brock, B. and W. A. Hunt, Jr.: 1999, 'Formal Analysis of the Motorola CAP DSP'. In: *Industrial-Strength Formal Methods*. Springer-Verlag.

Church, A. and S. C. Kleene: 1937, 'Formal Definitions in the Theory of Ordinal Numbers'. *Fundamenta Mathematicae* **28**, 11–21.

Clarke, E. M.: 1976, 'Completeness and Incompleteness of Hoare-like Axiom Systems'. Ph.D. thesis, Cornell University.

Clint, M.: 1973, 'Program Proving: Coroutines'. *Acta Informatica* **2**, 50–63.

Clint, M. and C. A. R. Hoare: 1971, 'Program Proving: Jumps and Functions'. *Acta Informatica* **1**, 214–224.

Colby, C., P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline: 2000, 'A Certifying Compiler for Java'. In: *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*. pp. 95–107.

Cook, S. A.: 1978, 'Soundness and Completeness of an Axiom System for Program Verification'. *SIAM Journal of Computing* **7**(1), 70–90.

de Bakker, J. W.: 1980, *Mathematical Theory of Program Correctness*. Prentice-Hall.

Detlefs, D. L., K. R. M. Leino, G. Nelson, and J. B. Saxe: 1998, 'Extended Static Checking for Java'. Technical Report 159, Compaq Systems Research Center.

Dijkstra, E. W.: 1975, 'Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs'. *Language Hierarchies and Interfaces* pp. 111–124.

Dijkstra, E. W.: 1978, *A Discipline of Programming*. Prentice-Hall.

Flatau, A. D.: 1992, 'A Verified Language Implementation of an Applicative Language with Dynamic Storage Allocation'. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin.

Floyd, R.: 1967, 'Assigning Meanings to Programs'. In: *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, Vol. XIX. Providence, Rhode Island, pp. 19–32.

Fox, A. C. J.: 2003, 'Formal Specification and Verification of ARM6'. In: D. A. basin and B. Wolff (eds.): *Proceedings of the* 16*th International Conference on Theorem Proving in Higher-Order Logics (TPHOLS 2003)*, Vol. 2758 of *LNCS*. pp. 25–40.

Gloess, P. Y.: 1999, 'Imperative Program Verification in PVS'. Technical report, École Nationale Supérieure Électronique, Informatique et Radiocommunications de bordeaux. See URL `http://dept-info.labri.u.bordeaux.fr/~imperative/index.html`.

Goldstein, H. H. and J. von Neumann: 1961, 'Planning and Coding Problems for an Electronic Computing Instrument'. In: *John von Neumann, Collected Works, Volume V*.

Gordon, M. J. C. and T. F. Melham (eds.): 1993, *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press.

Greve, D., M. Wilding, and D. Hardin: 2000, 'High-Speed, Analyzable Simulators'. In: M. Kaufmann, P. Manolios, and J. S. Moore (eds.): *Computer-Aided Reasoning: ACL2 Case Studies*. Boston, MA, pp. 89–106.

Hamon, G. and J. Rushby: 2004, 'An Operational Semantics for Stateflow'. In: M. Wermelinger and T. Margaria (eds.): *Proceedings of the* 7*th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, Vol. 2984 of *LNCS*. pp. 229–243.

Harrison, J.: 1998, 'Formalizing Dijkstra'. In: J. Grundy and M. Newer (eds.): *Proceedings of the* 11*th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1998)*, Vol. 1479 of *LNCS*. pp. 171–188.

Hoare, C. A. R.: 1969, 'An Axiomatic Basis for Computer Programming'. *Communications of the ACM* **12**(10), 576–583.

Hoare, C. A. R.: 1972, 'Proof of Correctness of Data Representations'. *Acta Informatica* **1**, 271–281.

Homeier, P. and D. Martin: 1995, 'A Mechanically Verified Verification Condition Generator'. *The Computer Journal* **38**(2), 131–141.

Hunt, Jr., W. A.: 1994, *FM8501: A Verified Microprocessor*, Vol. 795 of *LNAI*. Springer-Verlag.

Hunt, Jr., W. A. and B. Brock: 1992, 'A Formal HDL and Its Use in the FM9001 Verification'. In: C. A. R. Hoare and M. J. C. Gordon (eds.): *Mechanized Reasoning and Hardware Design*. Englewood Cliffs, NJ, pp. 35–48.

Kaufmann, M., P. Manolios, and J. S. Moore: 2000, *Computer-Aided Reasoning: An Approach*. Boston, MA: Kluwer Academic Publishers.

Kaufmann, M. and J. S. Moore: 2006, 'ACL2 home page'. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

King, J. C.: 1969, 'A Program Verifier'. Ph.D. thesis, Carnegie-Melon University.

King, S., J. Hammond, R. Chapman, and A. Pryor: 2000, 'Is Proof More Cost-Effective Than Testing?'. *IEEE Transactions on Software Engineering* **26**(8), 675–686.

Liu, H. and J. S. Moore: 2004, 'Java Program Verification via a JVM Deep Embedding in ACL2'. In: K. Slind, A. Bunker, and G. Gopalakrishnan (eds.): *Proceedings of the* 17*th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, Vol. 3233 of *LNCS*. Park City, Utah, pp. 184–200.

Liu, H. and J. S. Moore: 2005, 'Executable JVM model for analytical reasoning: A study'. *Science of Computer Programming* **57**(3), 253–274.

Manna, Z.: 1969, 'The Correctness of Programs'. *Journal of Computer and Systems Sciences* **3**(2), 119–127.

Manna, Z. and A. Pnueli: 1995, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag.

Manolios, P. and J. S. Moore: 2003, 'Partial Functions in ACL2'. *Journal of Automated Reasoning* **31**(2), 107–127.

Manolios, P. and D. Vroon: 2003, 'Algorithms for Ordinal Arithmetic'. In: F. Baader (ed.): *Proceedings of the* 19*th International Conference on Automated Deduction (CADE 2003)*, Vol. 2741 of *LNAI*. Miami, FL, pp. 243–257.

Matthews, J., J. S. Moore, S. Ray, and D. Vroon: 2006, 'Verification Condition Generation Via Theorem Proving'. In: M. Hermann and A. Voronkov (eds.): *Proceedings of the* 13*th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, Vol. 4246 of *LNCS*. pp. 362–376.

Matthews, J. and D. Vroon: 2004, 'Partial Clock Functions in ACL2'. In: M. Kaufmann and J. S. Moore (eds.): 5*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*. Austin, TX.

McCarthy, J.: 1962, 'Towards a Mathematical Science of Computation'. In: *Proceedings of the Information Processing Congress*, Vol. 62. pp. 21–28.

Mehta, F. and T. Nipkow: 2003, 'Proving Pointer Programs in Higher Order Logic'. In: F. Baader (ed.): *Proceedings of the* 19*th International Conference on Automated Deduction (CADE 2003)*, Vol. 2741 of *LNAI*. pp. 121–135.

Moore, J. S.: 1996, *Piton: A Mechanically Verified Assembly Language*. Kluwer Academic Publishers.

Moore, J. S.: 2003a, 'Inductive Assertions and Operational Semantics'. In: D. Geist (ed.): *Proceedings of the* 12*th International Conference on Correct Hardware Design and Verification Methods*, Vol. 2860 of *LNCS*. pp. 289–303.

Moore, J. S.: 2003b, 'Proving Theorems about Java and the JVM with ACL2'. In: M. Broy and M. Pizka (eds.): *Models, Algebras, and Logic of Engineering Software*. pp. 227–290.

Necula, G. C.: 1997, 'Proof-Carrying Code'. In: *Proceedings of the* 24*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*. pp. 106–119.

Norrish, M.: 1998, 'C Formalised in HOL'. Ph.D. thesis, University of Cambridge.

Oppen, D. C. and S. A. Cook: 1975, 'Proving Assertions about Programs that Manipulate Data Structures'. In: *Proceedings of the* 7*th Annual ACM Symposium on Theory of Computing (STOC 1975)*. pp. 107–116.

Ortner, V. and N. Schirmer: 2005, 'Verification of BDD Normalization'. In: J. Hurd and T. Melham (eds.): *Proceedings of the* 16*th International Conference on Theorem Proving in Higher-Order Logics (TPHOLS 2005)*, Vol. 3603 of *LNCS*. pp. 261–277.

Owicki, S. S. and D. Gries: 1976, 'Verifying Properties of Parallel Programs: An Axiomatic Approach'. *Communications of the ACM* **19**(5), 279–285.

Owre, S., J. M. Rushby, and N. Shankar: 1992, 'PVS: A Prototype Verification System'. In: D. Kapoor (ed.): *11th International Conference on Automated Deduction (CADE)*, Vol. 607 of *LNAI*. pp. 748–752.

Ray, S. and W. A. Hunt, Jr.: 2004, 'Deductive Verification of Pipelined Machines Using First-Order Quantification'. In: R. Alur and D. A. Peled (eds.): *Proceedings of the* 16*th International Conference on Computer-Aided Verification (CAV 2004)*, Vol. 3114 of *LNCS*. Boston, MA, pp. 31–43.

Ray, S. and J. S. Moore: 2004, 'Proof Styles in Operational Semantics'. In: A. J. Hu and A. K. Martin (eds.): *Proceedings of the* 5*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, Vol. 3312 of *LNCS*. Austin, TX, pp. 67–81.

Sawada, J. and W. A. Hunt, Jr.: 1997, 'Trace Table Based Approach for Pipelined Microprocessor Verification'. In: O. Grumberg (ed.): *Proceedings of the* 9*th International Conference on Computer-Aided Verification (CAV 1997)*, Vol. 1254 of *LNCS*. pp. 364–375.

Schirmer, N.: 2005, 'A Verification Environment for Sequential Imperative Programs in Isabelle/HOL'. In: F. Baader and A. Voronkov (eds.): *Proceedings of the* 11*th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, Vol. 3452 of *LNAI*. pp. 398–414.

Shankar, N.: 1997, 'Machine-Assisted Verification Using Theorem Proving and Model Checking'. In: M. Broy and B. Schieder (eds.): *Mathematical Methods in Program Development*, Vol. 158 of *NATO ASI Series F: Computer and Systems Science*. Springer-Verlag, pp. 499–528.

Sokolowski, S.: 1977, 'Axioms for Total Correctness'. *Acta Informatica* **9**, 61–71.

Strecker, M.: 2002, 'Formal Verification of a Java Compiler in Isabelle'. In: A. Voronkov (ed.): *Proeedings of the* 18*th International Conference on Automated Deduction (CADE 2002)*, Vol. 2392 of *LNCS*. pp. 63–77.

Turing, A. M.: 1949, 'Checking a Large Routine'. In: *Report of a Conference on High Speed Automatic Calculating Machine*. University Mathematical Laboratory, Cambridge, England, pp. 67–69.

Wand, M.: 1978, 'A New Incompleteness Result for Hoare's System'. *Journal of the ACM* **25**(1), 168–175.

Wilding, M.: 1993, 'A Mechanically Verified Application for a Mechanically Verified Environment'. In: C. Courcoubetis (ed.): *Proceedings of the* 5*th International Conference on Computer-Aided Verification (CAV 1993)*, Vol. 697 of *LNCS*. pp. 268–279.

Wilding, M.: 1997, 'Robust Computer System Proofs in PVS'. In: C. M. Holloway and K. J. Hayhurst (eds.): 4*th NASA Langley Formal Methods Workshop*.

Young, W. D.: 1988, 'A Verified Code Generator for a Subset of Gypsy'. Technical Report 33, Computational Logic Inc.
Yu, Y.: 1992, 'Automated Proofs of Object Code for a Widely Used Microprocessor'. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin.