Inductive Assertions and Operational Semantics

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
E-mail: moore@cs.utexas.edu

Abstract. This paper shows how classic inductive assertions can be used in conjunction with an operational semantics to prove partial correctness properties of programs. The method imposes only the proof obligations that would be produced by a verification condition generator but does not require the definition of a verification condition generation. The paper focuses on iterative programs but recursive programs are briefly discussed. Assertions are attached to the program by defining a predicate on states. This predicate is then "completed" to an alleged invariant by the definition of a partial function defined in terms of the state transition function of the operational semantics. If this alleged invariant can be proved to be an invariant under the state transition function, it follows that the assertions are true every time they are encountered in execution and thus that the post-condition is true if reached from a state satisfying the pre-condition. But because of the manner in which the alleged invariant is defined, the verification conditions are sufficient to prove invariance. Indeed, the "natural" proof generates as subgoals the classical verification conditions. The invariant function may be thought of as a state-based verification condition generator for the annotated program. The method allows standard inductive assertion style proofs to be constructed directly in an operational semantics setting. The technique is demonstrated by proving the partial correctness of a simple bytecode program with respect to a pre-existing operational model of the Java Virtual Machine.

1 Summary

This paper connects two well-known approaches to program verification: operational semantics and inductive assertions. The paper shows how one can adopt the clarity and concreteness of a formal operational semantics while incurring just the proof obligations of the inductive assertion method, without writing a verification condition generator or other extra-logical tool. In particular, the formal definition of the state transition function can be used directly to generate verification conditions for annotated programs.

In this section the idea is presented in the abstract. Some details are skipped and a deliberate confusion of states with formulas is perpetrated to convey the basic idea. Subsequently, the method is applied to a particular formal operational semantics, program, annotation, mechanical theorem prover, etc., to demonstrate that the basic idea is practical.

Consider a simple one loop program π (Figure 1) that concludes with a `HALT` instruction. Assume instructions are addressed sequentially, with α being the address or label of the first instruction and γ being the address or label of the `HALT`. Let the pre- and post-conditions of the program be P and Q respectively. The arrows of Figure 1 indicate the control flow; functions f, g, and h indicate the compound state transitions along the arcs and t is the test for staying in the loop. R is the loop invariant and "cuts" the only loop. The partial correctness challenge is to prove that if P holds at α then Q holds whenever (if) control reaches γ.

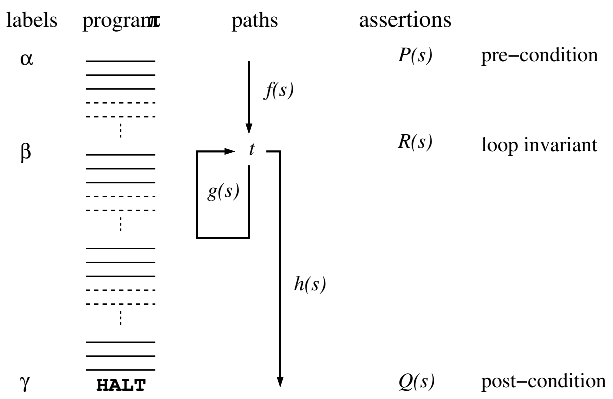

Fig. 1. The One-Loop Program π with Annotations

To give meaning to such programs with an operational semantics, one formalizes the abstract machine state and the effect of each instruction on the state. Typically the state, s, is a vector or n-tuple describing available computational resources such as environments, stacks, flags, etc. It is assumed here that the state includes a program counter, $pc\,(s)$, and the current program, $prog\,(s)$), which are used to determine the next instruction. Instructions are given meaning by defining a state transition function *step*. Typically, $step\,(s)$ is defined by considering the next instruction and transforming the state components accordingly. For example, a `LOAD` instruction might advance the program counter and push onto some stack the contents of some specified variable. More complicated instructions, such as method invocation, may affect many parts of the state. The `HALT` instruction is particularly simple; it is a no-op.

It is convenient to define an iterated step function:

$$run\,(k, s) = \begin{cases} s & \text{if } k = 0 \\ run\,(k-1, step\,(s)) & \text{otherwise} \end{cases}$$

and to make the convention that $s_k = run\,(k, s)$.

Given this operational semantics, the formalization of the partial correctness result is

Theorem: Correctness of Program π.

$$pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \wedge pc\,(s_k) = \gamma \rightarrow Q\,(s_k).$$

Proof. In an operational semantics setting, theorems such as the Correctness of Program π are proved by establishing an invariance $Inv\,(s)$ with the following three properties:

1. $Inv\,(s) \rightarrow Inv\,(step\,(s))$,
2. $pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \rightarrow Inv\,(s)$, and
3. $pc\,(s) = \gamma \wedge prog\,(s) = \pi \wedge Inv\,(s) \rightarrow Q\,(s)$.

The main theorem is then proved as follows. The inductive application of property 1 produces

4. $Inv\,(s) \rightarrow Inv\,(s_k)$.

Furthermore, instantiation of the s in property 3 with s_k produces

5. $pc\,(s_k) = \gamma \wedge prog\,(s_k) = \pi \wedge Inv\,(s_k) \rightarrow Q\,(s_k)$.

We assume no instruction in π changes the program; hence $prog\,(s) = prog\,(s_k)$. The Correctness of Program π then follows immediately from 2, 4, and 5. $\square$

Property 1, above, is problematic; it forces the user of the methodology to characterize all the states reachable from the chosen initial state. Contrast this situation with that enjoyed by the user of the inductive assertion method, where assertions are attached only to certain user-chosen cut-points, as in Figure 1. An extra-logical process, which encodes the language semantics as formula transformations, is then applied to the annotated program text to generate proof obligations or verification conditions

VC1. $P\,(s) \rightarrow R\,(f\,(s))$,
VC2. $R\,(s) \wedge t \rightarrow R\,(g\,(s))$, and
VC3. $R\,(s) \wedge \neg t \rightarrow Q\,(h(s))$.

If these formulas are proved, the user is then assured that if P holds initially then Q holds when (if) the program terminates.

To render this assurance formal, i.e., write it as a formula, one must adopt some logic of programs, i.e., a logic that allows the combination of classical mathematical expressions about numbers, sequences, vectors, etc., with program text and terminology. The resulting programming language semantics is extra-logical in the sense that it is expressed as rules of inference in a metalanguage and is not directly subject to formal analysis within the logic. In contrast, in the operational approach, the semantics is expressed within the language (typically as defined functions or relations on states), programs are objects in the logical universe, and the properties of both — programs and the semantic functions and relations – are subject to proof within the logic.

The central question of this paper is whether it is possible to have the best of both worlds: the concreteness and clarity of an operational semantics in a classical logical setting but the elegance and simplicity of an inductive assertion-style proof. The central question may be put bluntly as "Is it possible to prove the *formula* named 'Correctness of Program π,' above, directly from VC1–VC3?" The answer is "yes."

Recall that the proof of 'Correctness of Program π' required the definition of $Inv\,(s)$ satisfying properties 1–3 above. The key to constructing an inductive assertion-style proof in an operational setting is the following definition of $Inv\,(s)$.

$$Inv\,(s) \equiv \begin{cases} prog\,(s) = \pi \wedge P\,(s) & \text{if } pc\,(s) = \alpha \\ prog\,(s) = \pi \wedge R\,(s) & \text{if } pc\,(s) = \beta \\ prog\,(s) = \pi \wedge Q\,(s) & \text{if } pc\,(s) = \gamma \\ Inv\,(step\,(s)) & \text{otherwise} \end{cases}$$

The logician will immediately ask whether there exists a predicate satisfying this equivalence. The affirmative answer is provided in [8]. The logical crux of the matter is that $Inv\,(s)$ is defined with tail-recursion and there exists a satisfying and total witness for every tail-recursive equivalence. If some loop in the program is not cut, the equivalence may not uniquely define a predicate, but at least one witness exists.

$Inv\,(s)$ clearly has properties 2 and 3. It therefore remains only to prove property 1. As will become apparent, the proof that $Inv\,(s)$ has property 1 will *generate the verification conditions as subgoals.* To drive this home, we describe the process by which the proof is constructed rather than merely the formulas produced. Recall Figure 1. Successive steps from a state s with pc α eventually produce the state $f\,(s)$ with pc β. Similarly, if t, then successive steps from a state s with pc β produce $g\,(s)$ with pc β, and if $\neg t$, then successive steps from a state s with pc β produce $h\,(s)$ with pc γ. Furthermore, repeated symbolic expansion and simplification of the *step* function produce the transformations described by f, g, and h.

Theorem: Property 1.

$$Inv\,(s) \rightarrow Inv\,(step\,(s))$$

Proof. Consider the cases on $pc\,(s)$ as used in the definition of *Inv*.

Case: $pc\,(s) = \alpha$. The hypothesis, $Inv\,(s)$ may be simplified to $prog\,(s) = \pi \wedge P\,(s)$. Consider the conclusion, $Inv\,(step(s))$. Symbolic simplification of $step\,(s)$, given $pc\,(s) = \alpha$ and $prog\,(s) = \pi$, produces a symbolic state s' with $pc\,(s') = \alpha + 1$. For program π either $\alpha + 1$ is β or it is none of the cut points α, β or γ. In the latter case, $Inv\,(step\,(s)) \equiv Inv\,(s') \equiv Inv\,(step\,(s'))$ and stepping continues until β is reached at state $f\,(s)$. Hence, $Inv\,(step\,(s)) \equiv R\,(f\,(s'))$ (since $prog\,(f\,(s)) = \pi$). Thus, this case simplifies to the goal

$$pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \rightarrow R\,(f\,(s)).$$

This is just VC1 (with two now-irrelevant hypotheses, given traditional assertions P and R).

Case: $pc\,(s) = \beta$. The hypothesis $Inv\,(s)$ simplifies to $prog\,(s) = \pi \wedge R\,(s)$. Then the symbolic simplification of $step\,(s)$ in the conclusion produces a bifurcated symbolic state whose program counter depends on test t. Repeated expansions of the definition of Inv on both branches of the state eventually reach states $g\,(s)$ and $h\,(s)$ at which Inv is defined. The results are VC2 and VC3, respectively.

Case: $pc\,(s) = \gamma$. The hypothesis $Inv\,(s)$ simplifies to $prog\,(s) = \pi \wedge Q\,(s)$. But the $step\,(s)$ in the conclusion simplifies to s because the instruction at γ in π is the no-op `HALT`. Hence, $Inv\,(s) \equiv Inv\,(step(s))$ and this case is trivial (propositionally true independent of the assertions).

Case: otherwise. Since $pc\,(s)$ is not one of the cut-points, $Inv\,(s) \equiv Inv\,(step(s))$ by definition of Inv and this case is also trivial.
$\square$

Hence, if the verification conditions VC1–VC3 have been proved, the proof of property 1, the step-wise invariance of *Inv*, involves no assertion-specific reasoning. More interestingly, given the definition of *Inv*, the proof *generates* the verification conditions by symbolic expansion of the operational semantics' state transition function.

Practically speaking this means that with a mechanical theorem prover and a formal operational semantics one can enjoy the benefits of the inductive assertion method without writing a verification condition generator or other extra-logical tools to do formula transformations.

Another practical ramification of this paper is that it provides a simple means to define a step-wise invariant given only the assertions at the cut points. Step-wise invariants are frequently needed in operational semantics-based proofs of safety and liveness properties.

2 Related Work and Discussion

McCarthy [9] made explicit the notion of operational semantics, in which "the meaning of a program is defined by its effect on the state vector."

The inductive assertion method for proving programs correct was implicitly used by von Neumann and Goldstine in [3] and made explicit in the classic papers by Floyd [2] and Hoare [4]. The first mechanized verification condition generator, which generates proof obligations from code and attached assertions, was written by King [6]. Hoare, of course, rendered the inductive assertion method formal by introducing a logic of programs. From the practical perspective most program logics are mechanized with two trusted tools, a formula generator, here called a VCG, and a theorem prover. It is not uncommon for the VCG to include not just language semantics as formula transformers but also some logical simplification (i.e., theorem proving) to keep the generated proof obligations manageable.

This paper contains one apparently novel idea: a step-wise invariant can be defined from the inductive assertions using the state-transition function. One may think of this as a methodology for obtaining a state-based verification condition generator from an operational semantics. By doing it on a per program basis the method avoids the need to generate or trust extra-logical tools.

The use of inductive assertions in conjunction with a formal operational semantics to prove partial correctness results mechanically is not new. Robert S. Boyer and the author developed it for their *Analysis of Programs* course at the University of Texas at Austin as early as 1983. In that class, an operational semantics for a simple procedural language in Nqthm [1] was defined and the course explored program correctness proofs that combined operational semantics with inductive assertions. These proofs motivated the exploration of total versus partial correctness, Hoare logics, and verification condition generation. For an Nqthm proof script illustrating the use of inductive assertions in an operational semantics setting, see [10].

A recent example of the use of assertions to prove theorems about a program modeled operationally may be found in [13], where a safety property of a non-terminating multi-threaded Java system is proved with respect to an operational semantics for the Java Virtual Machine [12].

However, in the earlier work the invariant explicitly included an assertion for every value of the *pc*. (The invariant must recognize every reachable state and so must handle every *pc*; the issue is whether it does so explicitly or implicitly.)

An alternative way to combine inductive assertions at selected cut points with an operational semantics in a classical formal setting is to formalize and verify a VCG with respect to the operational semantics. In [5], for example, an HOL proof of the correctness of a VCG for a simple procedural language is described. The work includes support for mutually recursive procedures. Formal proofs of the verification conditions could, in principle, be used with the theorem stating the correctness of the VCG, to derive a property stated operationally. But the method described here does not require the definition of a VCG much less a proof of its correctness.

Logically speaking, a crucial aspect of the novel idea here is that the step-wise invariant is defined using tail recursion. The admission of a new function or predicate symbol via recursive definition is generally handled by a definitional principle that insures the existence (and often the uniqueness) of the defined concept. In many logics, this requires a termination proof. Admitting *Inv* under such a definitional principle would require a measure of the distance to the next cut point and a proof that the distance decreases under *step*. That imposes a proof burden not generally incurred by the user of the inductive assertion method. (Every loop must be cut for the inductive assertion method to be effective; the question is whether that must be proved formally or merely demonstrated by the successful generation of the verification conditions.)

The technique used here exploits the observation that *Inv* is tail-recursive and hence admissible without proof obligation, given the work of Manolios and Moore [8] in which it was proved that every tail-recursive equation may be witnessed by a total function. The tail-recursive function may not be uniquely defined by the equation — this occurs if insufficient cut points are chosen. Such a failure is manifested by an infinite loop in the process of generating/proving the step invariance. This is the same behavior a VCG user would experience in the analogous situation.

The technique here is similar in spirit to one used by Pete Manolios [private communication] to attack the 2-Job version of the Apprentice problem [13]. There, he defined the reachable states of the Apprentice problem as all the states that could be reached from certain states by the execution of a fixed maximum number of steps.

3 A Demonstration of the Method

To illustrate the technique a mechanized formal logic and an operational semantics must be introduced. In this paper we use the ACL2 logic [?]. In this logic, function application is denoted as in Lisp, e.g., $run(k, s)$ is written `(run k s)`.

For the demonstration we choose a pre-existing operational semantics for a significant fragment of the JVM [7] is used. The model is called M5 [12] and it was chosen simply because it was available and it was realistic. The JVM is a good example of an abstract machine that is sufficiently complicated that writing a VCG for it a serious and error-prone undertaking.

The semantics of the JVM bytecode may be gathered from [7] or by inspection of the formal model. In this paper, comments in our bytecode explain the language. Roughly speaking, the JVM is a stack machine in which each method invocation allocates a new stack frame which is popped upon return. Each frame contains a stack for the computation of intermediate results by the bytecode of the method. The most primitive native arithmetic is 32-bit twos complement, here called "`int` arithmetic" after the Java term for such integers. In `int` arithmetic, overflow is not signaled; adding one to the most positive integer produces the most negative integer. M5 models this and many other aspects of Java, including the creation of instance objects in the heap, the invocation of static, special, and virtual methods, the creation of multiple threads, and synchronization via monitors. The later details are not exposed in this paper, with one exception: the `step` function for M5 takes two arguments instead of just one: `(step th s)` is the state obtained by stepping thread `th` in state `s`. The `run` function, instead of taking the number of steps, takes a list of thread identifiers, called a schedule, and steps those threads sequentially.

The paper describes a partial correctness proof of a simple M5 program via the inductive assertion method. See [11] for a long version of the paper with other examples.

4 An Iterative Program

Below is an M5 program that decrements its first local, informally called `n`, by 2 and iterates until the result is 0. On each iteration it adds 1 to its second local variable, here called `a`, which is initialized to 0. Thus, the method computes `n`/2, henceforth written `(/ n 2)`, when `n` is even. It does not terminate when `n` is odd.

The program is slightly simpler to deal with if it is assumed that `n` is a non-negative `int`. The program actually terminates for even negative `int`s, because Java's `int` arithmetic wraps around: the most negative `int`, `-2147483648`, is even and when it is decremented by 2 it becomes the most positive even, `2147483646`. For simplicity, the program concludes with the fictitious `HALT` instruction, which stops the machine. The program constant below is named `*flat-prog*` because it does not return to a caller but stops the machine. Method invocation is discussed later in the paper.

```
(defconst *flat-prog*
  '((ICONST_0)      ; 0
    (ISTORE_1)      ; 1             a := 0
    (ILOAD_0)       ; 2  top of loop:
    (IFEQ 14)       ; 3             if n=0, goto 17
    (ILOAD_1)       ; 6
    (ICONST_1)      ; 7
    (IADD)          ; 8
    (ISTORE_1)      ; 9             a := a+1
    (ILOAD_0)       ;10
    (ICONST_2)      ;11
    (ISUB)          ;12
    (ISTORE_0)      ;13             n := n-2
    (GOTO -12)      ;14             goto top of loop
    (ILOAD_1)       ;17             push a
    (HALT)))        ;18
```

Let the initial value of `n` be `n0`. The goal is to prove that if `n0` is a non-negative `int` and control reaches `pc` 18, then `n0` is even and `(/ n 2)` is on the stack. That is, if the program halts the initial input must have been even and the final answer is half that input.

Rather than deal with integer division during the code proof, the following function is introduced. The decision to use this function rather than algebraic expressions to express the properties of the code is independent of the decision to express the properties with inductive assertions.

```
(defun halfa (n a)
  (if (zp n)
      a
    (halfa (- n 2) (int-fix (+ a 1)))))
```

Here, `int-fix` returns the integer represented by the low-order 32-bits of its argument and thus implements `int` wrap-around. The inductive assertion method will be used to establish that if the program terminates it will leave `(halfa n0 0)` on the stack. A second theorem, independent of the code, establishes that `(halfa n0 0)` is `(/ n 2)` under certain conditions. Such decomposition of code proofs into “algorithm” and “requirements” is standard in the ACL2 community and independent of whether inductive assertions are being used. It is possible, of course, to mix the two via inductive assertions about division or multiplication by two.

5 The Assertions at the Three Cut Points

The cut points, to which assertions will be attached, are at program counters 0 (α), 2 (β), and 18 (γ). The assertions themselves, called P, R, and Q in the earlier treatment, are captured by the following function definitions. The names of the functions are, of course, irrelevant but indicate how they will be used. In the earlier treatment it was convenient to make these functions of state; here they are functions of the initial input `n0` and the relevant state components, namely `n` and `a`.

```
(defun flat-pre-condition (n0 n)
  (and (equal n n0)
       (intp n0)
       (<= 0 n0)))
(defun flat-loop-invariant (n0 n a)
  (and (intp n0)
       (<= 0 n0)
       (intp n)
       (if (and (<= 0 n)
                (evenp n))
           (equal (halfa n a)
                  (halfa n0 0))
         (not (evenp n)))
       (iff (evenp n0) (evenp n))))
(defun flat-post-condition (n0 value)
  (and (evenp n0)
       (equal value (halfa n0 0))))
```

The details of the assertions are not germane to this paper. The assertions are typical inductive assertions for such a program. They are complicated primarily because of Java’s `int` arithmetic. `Halfa` tracks the behavior of the program only as long as `n` stays non-negative. Things would be simpler if the pre-condition required that `n0` be even or if the post-condition did not assert that `n0` is even. These assertions were chosen to illustrate that operational semantics could be used to address partial correctness of non-terminating programs including the characterization of when termination occurs.

6 Verification Conditions

Given `*flat-prog*`, the informal attachment of the three assertions to the chosen cut points, and a VCG for the JVM, the following verification conditions would be produced.

```
(defthm VC1                                       ; entry to loop
  (implies (flat-pre-condition n0 n)
           (flat-loop-invariant n0 n 0)))
(defthm VC2                                       ; loop to loop
  (implies (and (flat-loop-invariant n0 n a)
                (not (equal n 0)))
           (flat-loop-invariant n0
                                (int-fix (- n 2))
                                (int-fix (+ 1 a)))))
(defthm VC3                                       ; loop to exit
  (implies (and (flat-loop-invariant n0 n a)
                (equal n 0))
           (flat-post-condition n0 a)))
```

These are easily proved. The challenge is: how can these three theorems be used to verify a partial correctness result for `*flat-prog*`?

7 Attaching the Assertions to the Code

In the earlier treatment of the method, the invariant conjoined each assertion with $prog(s) = \pi$. Here we introduce an intermediate function to do this and also to name relevant components of the state.

```
(defun flat-assertion (n0 th s)
  (let ((n (nth 0 (locals (top-frame th s))))
        (a (nth 1 (locals (top-frame th s)))))
    (and (equal (program (top-frame th s)) *flat-prog*)
         (case (pc (top-frame th s))
           (0  (flat-pre-condition n0 n))
           (2  (flat-loop-invariant n0 n a))
           (18 (let ((value (top (stack (top-frame th s)))))
                 (flat-post-condition n0 value)))
           (otherwise nil)))))
```

The `let` identifies parts of the JVM state of interest: the 0^{th} local of thread `th`, called `n`, and the 1^{st} local of thread `th`, called `a`. It requires that the program being executed by the thread be `*flat-prog*` ("π"). It then case splits on the `pc` of thread `th` and for program counters `0`, `2`, and `18` makes an assertion about `n`, `a`, and `n0`. The variable symbol `value` at the post-condition is bound to the value on top of the operand stack of the relevant thread at the conclusion of the program.

8 The Nugget: Defining the Invariant

The nugget in this paper is how the assertions, attached to selected cut points, are completed into a step-wise invariant on states.

The invariant is introduced with the `defpun` ("define partial function") utility of [8]. The assertions are tested at the three cut points and all other statements inherit the invariant of the next statement. This definition is analogous to that for *Inv* in the abstract treatment, except that the invariant also takes the initial input, `n0`, and the identifier of the relevant thread, `th`.

```
(defpun flat-inv (n0 th s)
  (if (or (equal (pc (top-frame th s)) 0)
          (equal (pc (top-frame th s)) 2)
          (equal (pc (top-frame th s)) 18))
      (flat-assertion n0 th s)
    (flat-inv n0 th (step th s))))
```

9 Proofs

Here is the key theorem, called "property 1 of *Inv*" or the step-wise invariant theorem.

```
(defthm flat-inv-step
  (implies (flat-inv n0 th s)
           (flat-inv n0 th (step th s))))
```

As noted earlier, the proof attempt generates the verification conditions (with a few extra hypotheses about the program counter and current program). If ACL2's data base already contains the theorems `VC1`–`VC3`, those theorems are used to complete the proof of `flat-inv-step`. If the verification conditions have not already been proved, the proof attempt here generates and proves them.

Central to the process is the symbolic simplification of state expressions under the state transition function `step`.

Having proved the invariance of `flat-inv` under `step` the next theorem in the mechanized "methodology" corresponds to property 4 of the earlier proof of the Correctness of Program π. is trivial. The theorem states that `flat-inv` is invariant under arbitrarily long runs of the thread in question.

```
(defthm flat-inv-run
  (implies (and (mono-threadedp th sched)
                (flat-inv n0 th s))
           (flat-inv n0 th (run sched s))))
```

where

```
(defun mono-threadedp (th sched)
  (if (endp sched)
```

```
    t
  (and (equal th (car sched))
       (mono-threadedp th (cdr sched))))).
```

Proof of `flat-inv-run` is trivial by induction and appeal to `flat-inv-step`.

Thus, if the initial state has `pc 0` and satisfies the pre-condition, and, after some arbitrary mono-threaded run, a state with `pc 18` is reached, then it satisfies the post-condition, namely, `n0` is even and the answer is `(halfa n0 0)`. Formally this can be written as follows.

```
(defthm flat-main
  (let ((s1 (run sched s0)))
    (implies (and (intp n0)
                  (<= 0 n0)
                  (equal (pc (top-frame th s0)) 0)
                  (equal (locals (top-frame th s0)) (list n0 any))
                  (equal (program (top-frame th s0)) *flat-prog*)
                  (mono-threadedp th sched)
                  (equal (pc (top-frame th s1)) 18))
             (and (evenp n0)
                  (equal (top (stack (top-frame th s1)))
                         (halfa n0 0))))))
```

This is proved by using the instance of `flat-inv-run` obtained by letting `s` be `s0`.

`Flat-main` is essentially the goal, except it characterizes the answer as `(halfa n0 0)`. If `(/ n0 2)` were preferred, either a separate proof relating `(halfa n0 0)` to `(/ n0 2)` could be performed, or the assertions could be stated in terms of division in the first place. In any case, this issue is independent of the use of inductive assertions.

Notice what has been accomplished. `Flat-main` is a partial correctness theorem about a JVM program, formalized with an operational semantics. The creative part of the proof consisted of the definition of the three assertions. The proof of the key lemma, `flat-inv-step`, generated (and requires the proof of) the classic verification conditions just as though a VCG for the JVM were available. But no VCG was defined. The proof does not establish termination of the code under the pre-conditions but does characterize necessary conditions to reach the `HALT` statement. Finally, neither the theorem nor the proof involved counting instructions or defining what is called a "clock function" in the Boyer-Moore community.

10 Method Invocation and Return

The `HALT` instruction in the previous program is fictitious but handy. Stepping the machine while on a `HALT` leaves the machine at the `HALT`. Thus, the invariance of the exit assertion is easy to prove once the exit is reached. In realistic code, the machine does not halt but returns control to the caller and non-trivial stepping continues. A useful inductive assertion methodology must deal with call and return. This paper does not discuss call and return in detail; see [11].

On the JVM, method invocation pushes a new stack frame on the invocation stack of the active thread. Abstractly, that frame may be thought of as containing the bytecode for the newly invoked method with initial `pc 0`. The new frame contains an initially empty "operand stack" for intermediate results. When certain return instructions are executed, the topmost item, v, on the operand stack is removed, the invocation stack is popped, and v is pushed onto the operand stack of the caller.[1]

To deal with call and return via inductive assertions, two changes are made to the "methodology" described above. First, instead of using `run` to run the state a certain number of steps, the new function `run-to-return` is introduced, which runs a certain number of steps or until the state returns from the call depth, `d0`, at which the run was started. Second, the assertion function is changed so that the post-condition is asserted if the call depth is less than `d0`.

To deal with recursive methods, one must characterize the stack of frames created by previous recursive calls so that `return`s produce states in which continued symbolic evaluation is possible.

11 Conclusion

This paper has demonstrated that inductive assertion style proofs can be carried out in an operational semantics framework, without producing a verification condition generator or incurring proof obligations beyond those produced by such a tool. The key insight is that assertions attached to cut points in a program can be propagated by a tail-recursive function to create an alleged invariant. The proof that the alleged invariant is invariant under the state transition function produces the standard verification conditions. The invariance result can then be traded in for a partial correctness result stated in terms of the operational semantics, without requiring the construction of clocks or the counting of instructions.

No verification condition generator need be constructed. Given an operational semantics it is possible, more or less immediately, to perform inductive assertion style proofs of partial correctness theorems.

The process of proving the step-wise invariance of the completed assertions "naturally" produces the verification conditions. To be more precise, the proof obligations produced correspond exactly to the verification conditions with some additional hypotheses about the location of the program counter and the identity of the program being analyzed.

This situation is attractive for three reasons. First, writing a verification condition generator for a realistic programming language like JVM bytecode is error-prone. For example, method invocation involves complicated non-syntactic issues like method resolution with respect to the object on which the method is

[1] Some forms of return implement `void` methods and return no v to the caller.

invoked, as well as side-effects to many parts of the state including, possibly, the call frames of both the caller and the callee, the thread table (in the event that a thread is started), the heap (in the event of a synchronized method locking the object upon which it is invoked), and the class table (in the event of dynamic class loading). Coding this all in terms of formula transformation instead of state transformation is difficult. Second, when completed, the semantics of the language is encoded in the VCG process rather than as sentences in a logic. This encoding of the semantics makes it difficult to inspect. In our approach, the semantics is expressed explicitly in the logic so that it can be inspected. Indeed, it is possible to prove theorems about the semantics (not just theorems about programs under the semantics). Finally, realistic VCGs contain simplifiers used to keep the generated proof obligations simple. These simplifiers are just theorems provers and must be trusted. In our approach, only one theorem prover is involved. It must be trusted but that trusted engine derives the verification conditions from the operational semantics and the user-supplied assertions.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.
2. R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, volume XIX, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967.
3. H. H. Goldstine and J von Neumann. *Planning and Coding Problems for an Electronic Computing Instrument.* Pergamon Press, Oxford, 1961.
4. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.
5. P. Homeier and D. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.
6. J. C. King. *A Program Verifier.* PhD thesis, Carnegie-Mellon University, 1969.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, Boston, MA., 1999.
8. P. Manolios and J S. Moore. Partial functions in acl2. Technical Report `http:/-/www.cs.utexas.edu/users/moore/publications/defpun/% -index.html`, Computer Sciences, University of Texas at Austin, 2001.
9. John McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.
10. J S. Moore. An nqthm formalization of a small machine. Technical Report ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/nqthm-1992/examples/basic/-small-machine.events, Computational Logic, Inc., May 1991.
11. J S. Moore. Inductive assertions and operational semantics – long version. Technical report, Department of Computer Sciences, University of Texas at Austin, 2003.
12. J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy, editor, *Lecture Notes of the Marktoberdorf 2002 Summer School.* Springer, LNCS, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.
13. J S. Moore and G. Porter. The apprentice challenge. *ACM TOPLAS*, 24(3):1–24, May 2002.