Inductive Assertions and Operational Semantics

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Taylor Hall 2.124
Austin, Texas 78712
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, without the introduction of a verification condition generator. In particular, we show how a formal statement about the operational semantics can be deduced more or less directly from verification conditions. Both iterative and recursive programs are considered. 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 tail-recursive 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. The fact that the assertions are completed via tail-recursion means that it is unnecessary to prove that the completion process terminates. Indeed, the invariant function may be thought of as a state-based verification condition generator built in a single function definition from the state transition function of the operational semantics. The method allows standard inductive assertion style proofs to be constructed directly in an operational semantics setting. To demonstrate the technique, a pre-existing model of the Java Virtual Machine is used as the operational semantics. Partial correctness theorems about several simple Java methods are presented. These theorems are contrasted total correctness results proved by defining "clock functions" that characterize how many steps are to be executed.

1 Summary

A formal operational semantics consists of a representation of a machine state (including some program to be run) as some object in a formal logic, together with the formal definition of a state transition function, `step`. The ACL2 [6] logic is used here. Function application is denoted as in Lisp. Hence, `(step s)` denotes the state obtained by applying the state transition function to `s`. The expression `(run k s)` is defined to be the state obtained by stepping `s` `k` times.

A typical *total correctness* theorem in this setting is a formula of the form

```
(implies (P n0 s)
         (Q n0 (run (clock n0) s)))
```

Here, P is the pre-condition for the program in `s` with initial inputs `n0`. For simplicity, suppose that the pre-condition includes the requirement that the program counter (pc) of `s` be some single entry point, $\langle entry \rangle$. Q is the post-condition. Suppose it includes the requirement that the pc be at a single exit point $\langle exit \rangle$. `(Clock n0)` is defined to return the number of steps necessary to drive the program from the entry to the exit when the pre-condition is true. This is a strong total correctness theorem because it not only guarantees termination but specifies how long the program runs. Many such examples may be found in [12].

Suppose a "partial correctness" theorem is desired. Such a theorem might take the form:

```
(implies (and (P n0 s)
              (equal (pc (run k s)) ⟨exit⟩))
         (Q n0 (run k s)))
```

The theorem may be read as "if `s` satisfies the pre-condition and a `run` of arbitrary length `k` produces a state where control has reached ⟨*exit*⟩, then the post-condition is true."

Such theorems are typically proved by the inductive assertion method. This is not new. What is new is how the inductive assertion method is formally embedded into an operational semantics model.

The first step is to attach assertions at selected cut-points by defining a function such as:

```
(defun assert (n0 s)
  (cond ((equal (pc s) ⟨entry⟩) (P n0 s))
        ((equal (pc s) ⟨loop⟩)  (I n0 s))
        ...
        ((equal (pc s) ⟨exit⟩)  (Q n0 s))
        (t nil)))
```

Next, this assertion is completed to an alleged invariant on states by defining the partial function:

```
(defpun invariant (n0 s)
  (if (member (pc s) '(⟨entry⟩ ⟨loop⟩ ... ⟨exit⟩))
      (assert n0 s)
      (invariant n0 (step s))))
```

This tail-recursive definition can be admitted to the ACL2 logic soundly since Manolios and Moore [9] proved that every tail-recursive equation has an admissible total function as a witness. (The `invariant` function will not terminate if every loop is not cut.) To members of the Boyer-Moore community, the only novel idea in this paper is the observation that the assertion can be completed to an invariant in a tail-recursive way without incurring a termination proof.

The third step is to prove that `invariant` is, indeed, invariant under stepping.

```
(implies (invariant n0 s)
         (invariant n0 (step s)))
```

When `invariant` is defined as shown here, this generates the same proof obligations as a conventional "verification condition generator" (VCG) would.

The fourth step is to observe, by a trivial induction which is automatic for ACL2, that the invariant holds for arbitrary runs.

```
(implies (invariant n0 s)
         (invariant n0 (run k s)))
```

The desired partial correctness result follows from the invariance above: if a state at ⟨*entry*⟩ satifies the pre-condition and a `run` reaches a state at ⟨*exit*⟩ then the post-condition holds.

The inductive assertion method for proving programs correct is among the oldest such methods. The idea 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 [7]. McCarthy [10] made explicit an alternative approach, operational semantics, in which "the meaning of a program is defined by its effect on the state vector."

This paper concerns the combination of the two, namely, the use of inductive assertions to prove theorems about programs modeled with an operational semantics. Inductive assertions, and more generally, the establishment of state invariants, is essential in verifying partial correctness, safety, and other properties in an operational setting. Another view of this work is that it proposes a method of defining state invariants by completing assertions provided for selected cut points in a program.

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 [11].

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].

So what is new? A careful look at the earlier work reveals that 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.) The classic Floyd-Hoare inductive assertion method requires an assertion only for selected cut points; a VCG is used to produce the proof obligations by propagating the assertions through the code.

An alternative way to combine inductive assertions at selected cut points with an operational semantics in a completely 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.[1]

Another way to use inductive assertions to prove theorems in an operational setting is admit `invariant` under the definitional principle. The admission would require proving termination of the recursion, which would, in turn, require defining a measure of the distance to the next cut point and proving that it decreased under `step`. That would represent a proof burden not generally incurred by the user of a VCG.

The technique used here exploits the observation that `invariant` is tail-recursive and hence admissible without proof obligation, given the work of Manolios and Moore [9] 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 observation that the assertions at the cut points can be completed to an alleged invariant without incurring proof obligations beyond those of the classic method immediately opens the door to inductive assertion-style proofs in an operational semantics setting.

[1] One could regard `invariant`, above, as a VCG. But if so it differs from classic VCGs in two senses. First, it is state based. Second, it is trivial by comparison to conventional VCGs, because it leverages the formal definition of operational semantics.

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.

To illustrate the technique an operational semantics must be introduced. In this paper a pre-existing operational semantics for a significant fragment of the JVM [8] is used. The model is called M5 [12]. The semantics of the JVM are of no special interest in this paper. M5 was chosen because it was available and it was realistic. Writing a VCG for JVM bytecode is a serious and error-prone undertaking. The semantics of the bytecode may be gathered from [8] or by inspection of the formal model. In this paper, comments in displayed 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. In this setting, the "clock functions" mentioned earlier become "schedule functions" specifying exactly how to step the various threads to reach the desired state.

The paper describes partial correctness proofs of several simple M5 programs via the inductive assertion method. See the supporting material for technical details, including some lemmas not mentioned here. In the theorems displayed below, the hints and other pragmatic advice sometimes provided to the theorem prover by the author have been omitted. Readers interested in getting these formulas proved with ACL2 should see the supporting scripts. The paper focuses on the basic idea of inductive assertions in an operational semantics, not on how to get a particular theorem prover to prove the verification conditions.

2 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)`, 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. The proof is done without defining a schedule or clock function and without counting or caring about how many instructions are executed and without incurring any more proof overhead than had a VCG for the JVM been used.

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.

3 The Assertions at the Three Cut Points

The cut points, to which assertions will be attached, are at pcs 0 (entry), 2 (loop), and 18 (exit). The assertions themselves are captured by the following function definitions. The names of the functions are, of course, irrelevant but indicate how they will be used.

```
(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. Under that more restrictive pre-condition it would be easy to define a clock function and prove total correctness. The pre-condition used here does not require `n0` to be even. Instead, it will be proved that if the program terminates then `n0` is even: the post-condition asserts that `n0` is even. In addition the post-condition asserts that the value computed is `(halfa n0 0)`.

4 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*`?

5 Attaching the Assertions to the Code

The assertions are attached to the code by defining the following predicate.

```
(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*`. It then case splits on the pc of thread `th` and for each of pcs `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 stack at the conclusion of the program.

6 The Nugget: Defining the Invariant

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

This is accomplished by using a tail-recursive "definition" introduced without a termination proof obligation under the `defpun` utility of [9]. The assertions are tested at the three cut points and all other statements inherit the invariant of the next statement.

```
(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))))
```

Had `defun` been used instead of `defpun`, a termination proof would be required. An appropriate measure would be the distance to the next cut point. The termination proof would be tantamount to proving that all loops were cut — a proof obligation not incurred by the user of a VCG.

After defining `flat-inv` above a technical lemma is proved that forces ACL2 to expand calls of `flat-inv` if the state is poised at some pc other than `0`, `2`, or `18`. See `flat-inv-make-state-opener` in the script for the details. If all the loops have been cut, this opening will stop and a "verification condition" will emerge. If some loop is not cut, the ACL2 rewriter will not terminate — a situation exactly comparable to what would happen in the classic approach.

7 Proofs

Here is the key theorem. It is proved without further guidance given the three verification conditions `VC1`, `VC2`, and `VC3`.

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

The proof is given below. However, it is described in terms of the process that generates it: the mechanical manipulation of the definitions and goal above.

Proof. Expand the definition of `flat-inv` in the hypothesis. This produces four subgoals: the pc is at `0`, `2`, `18`, or somewhere else. In the last case, the hypothesis becomes the conclusion and there is nothing more to prove. In the case where the pc is `18`, the `step` is a no-op because the instruction executed is `HALT` so the conclusion becomes the hypothesis. There are thus two non-trivial cases.

Case: pc = `0`. Then the hypothesis becomes `(flat-pre-condition n0 n)`. The `step` in the conclusion expands, symbolically executing the first instruction, and produces a state with pc `1` and with `0` on top of the stack. There is no assertion attached to pc `1` and so `flat-inv` performs another `step`, executes another instruction symbolically, deposits the `0` into the 1^{st} local, and produces a state with pc `2`. There is an assertion here. When applied to the state it becomes `(flat-loop-invariant n0 n 0)`. This is `VC1` and is thus proved.

Case: pc = `2`. The hypothesis becomes `(flat-loop-invariant n0 n a)`. The step produces a state with pc `3`. Since no assertion is found there, another `step` is taken, symbolically executing the `IFEQ` instruction. This produces two possible successor states, depending on whether `n` is `0`. Symbolic stepping continues on both paths until an assertion is reached. One of the paths produces `VC2` and the other produces `VC3`.

Q.E.D.

Note that the proof process described above actually *generates* the verification conditions. Thus, it is not actually necessary to identify and prove them separately. Defining `flat-assertion` and `flat-inv` as shown, and then attacking `flat-inv-step`, is exactly equivalent to generating and proving the verification conditions but produces a theorem about the operational semantics.

Having proved the invariance of `flat-inv` under `step` the next theorem in the "methodology" 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 a clock function.

8 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.

On the JVM, method invocation pushes a new stack frame on the invocation stack. 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.[2]

To prevent the machine from running "past" the return of interest, define

```
(defun run-to-return (sched th d0 s)
  (cond ((endp sched) s)
        ((<= d0 (sdepth (call-stack th s)))
         (run-to-return (cdr sched) th d0 (step (car sched) s)))
        (t s)))
```

which runs a state `s` with schedule `sched` until the depth of the invocation stack of thread `th` is less than `d0`. If that condition is never satisfied, the state is run until the schedule is exhausted. This function is easily related to `run`.

Using `run-to-return` instead of `run` in the main theorem, returns can be dealt with via inductive assertions. Let `*half-prog*` be the bytecode that results from replacing the last instruction in `*flat-prog*` by `(IRETURN)`, which returns one `int` value to the caller.

Here is the assertion function for `*half-program*`. It is comparable to `flat-assertion` in its program-specific content but contains supplementary material to handle features of the JVM, including the invocation stack.

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

```
(defun half-assertion (n0 d0 th s)
  (cond
   ((< (sdepth (call-stack th s)) d0)                    ; See note 1.
    (let ((value (top (stack (top-frame th s)))))
      (flat-post-condition n0 value)))
   (t
    (let ((n (nth 0 (locals (top-frame th s))))
          (a (nth 1 (locals (top-frame th s))))
          (stack (stack (top-frame th s))))
      (and (equal (sdepth (call-stack th s)) d0)          ; See note 2.
           (equal (program (top-frame th s)) *half-prog*)
           (equal (sync-flg (top-frame th s)) 'UNLOCKED)  ; See note 3.
           (case (pc (top-frame th s))
             (0 (flat-pre-condition n0 n))                ; See note 0.
             (2 (flat-loop-invariant n0 n a))
             (18 (let ((value (top (stack (top-frame th s)))))
                   (flat-post-condition n0 value)))
             (otherwise nil)))))))
```

Note 0: The assertions are exactly those used in the flat example. Note 1: This clause asserts that when control returns from depth `d0`, the exit assertion (also found for pc `18`) is true. Think of `d0` as the depth of the call stack while control is in the program in question. Note 2: This conjunct asserts that when control is in `*half-prog*` the call stack has length `d0`. Recursive methods require an inequality here and that is illustrated later. Note 3: Return from "synchronized" methods on the JVM release the monitor on a certain instance object in the heap. The conjunct here asserts that the `*half-prog*` method is not synchronized and thus the heap is not affected by return. Otherwise, this function is `flat-assertion`.

The invariant for `*half-prog*` is exactly analogous to what was shown for `*flat-prog*`, with one more case to handle the return from the method.

```
(defpun half-inv (n0 d0 th s)
  (if (or (< (sdepth (call-stack th s)) d0)
          (equal (pc (top-frame th s)) 0)
          (equal (pc (top-frame th s)) 2)
          (equal (pc (top-frame th s)) 18))
      (half-assertion n0 d0 th s)
    (half-inv n0 d0 th (step th s))))
```

The invariance theorem is also analogous:

```
(defthm half-inv-step
  (implies (and (integerp d0)
                (< 1 d0)
                (<= d0 (sdepth (call-stack th s)))
                (half-inv n0 d0 th s))
           (half-inv n0 d0 th (step th s))))
```

However it requires that `d0` a positive integer (guaranteeing that a call frame is below that for the program in question – this is the frame to which control will return when the `IRETURN` is executed) and that `d0` not exceed the depth of the call stack. The proof of this theorem proceeds exactly as before, with one additional but trivial verification condition: the assertion at the `IRETURN` insures the assertion for the caller's frame.[3]

From the invariance theorem it follows trivially that:

```
(defthm half-inv-run-to-return
  (implies (and (mono-threadedp th sched)
                (integerp d0)
                (< 1 d0)
                (half-inv n0 d0 th s))
           (half-inv n0 d0 th (run-to-return sched th d0 s))))
```

and hence

```
(defthm half-main
  (let ((s1 (run-to-return sched th (sdepth (call-stack th s0)) 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)) *half-prog*)
                  (equal (sync-flg (top-frame th s0)) 'unlocked)
                  (< 1 (sdepth (call-stack th s0)))
                  (mono-threadedp th sched)
                  (< (sdepth (call-stack th s1))
                     (sdepth (call-stack th s0))))
             (and (evenp n0)
                  (equal (top (stack (top-frame th s1)))
                         (halfa n0 0))))))
```

The final theorem captures partial correctness for `*half-prog*`: Suppose thread `th` of state `s0` satisfies the pre-condition for `*half-prog*`, namely, `n0` is a positive `int`, the pc is `0`, the first local is `n0`, the program is `*half-prog*`, the frame is not synchronized, and the call stack depth exceeds `1`. Such a frame might be produced by calling a method whose body consists of the bytecodes in `*half-prog*`. Suppose `sched` is an mono-threaded schedule on `th` and is of arbitrary length. Let `s1` be the result of running `s0` with `sched` "until" the return from depth `d0`. There is no guarantee that such a `return` occurs. However, the last hypothesis above supposes that it does.

Then the conclusion is that `n0` is even and the top item on the caller's operand stack is `(halfa n0 0)` (aka `(/ n0 2)`).

9 Recursive Methods

To handle recursive methods the assertion must include a description of the invocation stack down to the external caller. To illustrate the elaborations necessary to handle this, consider the bytecode generated for the following Java by the Sun `javac` compiler.

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

[3] The assertion at the `IRETURN` may be eliminated entirely but this presentation makes the assertion more closely resemble that for `*flat-prog*`.

```
    else return 1;
  }
```

That bytecode, in M5 notation, is

```
(defconst *fact-def*
  '("fact" (INT) NIL
    (ILOAD_0)                          ;;;  0
    (IFLE 12)                          ;;;  1
    (ILOAD_0)                          ;;;  4
    (ILOAD_0)                          ;;;  5
    (ICONST_1)                         ;;;  6
    (ISUB)                             ;;;  7
    (INVOKESTATIC "Demo" "fact" 1)     ;;;  8
    (IMUL)                             ;;; 11
    (IRETURN)                          ;;; 12
    (ICONST_1)                         ;;; 13
    (IRETURN)))                        ;;; 14
```

Note that the constant above includes the bytecode for `fact` but also includes other information from the class table entry for `fact`. The body of `fact` is actually the `cdddr` of this constant.

The following function characterizes an invocation stack of internal calls of `fact`. Note that when the `INVOKESTATIC` instruction at pc 8 is executed, the pc is advanced to `11` in the caller's frame and "then" the new frame is pushed.

```
(defun fact-caller-framesp (cs n0 k)
  (cond ((zp k) t)
        ((and (equal (pc (top cs)) 11)
              (equal (program (top cs)) (cdddr *fact-def*))
              (equal (sync-flg (top cs)) 'UNLOCKED)
              (intp (nth 0 (locals (top cs))))
              (equal (+ n0 (- k)) (- (nth 0 (locals (top cs))) 1))
              (equal (nth 0 (locals (top cs)))
                     (top (stack (top cs)))))
         (fact-caller-framesp (pop cs) n0 (- k 1)))
        (t nil)))
```

This predicate checks that the top `k` frames on the invocation stack `cs` ("call stack") are internal calls of `fact`. Namely, the pc is `11`, the program is that for `fact`, the frames are unsynchronized, the first local is an `int`, the first local is related to `n0` appropriately given its depth in the call stack, and the first local is also on top of the operand stack.

The assertion for `fact` is then written as follows. This looks complicated but contains a lot of "boilerplate" common to all recursive JVM methods. The expression `(! n)` below denotes the mathematical factorial function applied to `n` and is unbounded (unlike `fact`, which uses `int` arithmetic).

```
(defun fact-assertion (n0 d0 th s)
  (cond
   ((< (sdepth (call-stack th s)) d0)
    (equal (top (stack (top-frame th s)))
           (int-fix (! n0))))
   (t
    (let ((n (nth 0 (locals (top-frame th s)))))
```

```
(and (equal (program (top-frame th s)) (cdddr *fact-def*))
     (equal (lookup-method "fact" "Demo" (class-table s))
            *fact-def*)
     (equal (sync-flg (top-frame th s)) 'UNLOCKED)
     (intp n0)
     (intp n)
     (<= 0 n)
     (<= n n0)
     (equal (sdepth (call-stack th s)) (+ d0 (- n0 n)))
     (fact-caller-framesp (pop (call-stack th s)) n0 (- n0 n))
     (case (pc (top-frame th s))
       (0 t)
       ((12 14) (equal (top (stack (top-frame th s)))
                       (int-fix (! n))))
       (otherwise nil)))))))
```

When control exits from depth `d0`, the assertion is that the top item on the operand stack of the caller is the `int-fix` of `(! n0)` (the twos-complement integer denoted by the low-order 32 bits of the mathematical factorial of the input). While in the `fact` method, the assertion checks that the program is that for `fact`, the bytecode continues to be found in the class table, the frame is unsynchronized, `n0` and `n` (the first local) are `int`s such that $0 \leq$ `n` $\leq$ `n0`, the call stack depth is appropriately related to `d0`, `n0` and `n`, and the appropriate number of frames below this one are internal calls of `fact`. Finally, the specific cut points chosen for `fact` are the entry, pc `0`, and the two exits, pcs `12` and `14`. The pre-condition is vacuous (all has already been said). The post-condition is that the top of the stack contains the `int-fix` of `(! n)`.

It is not necessary to introduce explicitly such functions as `pre-condition`, `loop-invariant`, and `post-condition`. The assertions may be written "inline" in the `fact-assertion` function, as done above.

The invariant is defined with `defpun` exactly analogously to previous examples. The invariance theorem is also analogous.

```
(defthm fact-inv-step
  (implies (and (integerp d0)
                (< 1 d0)
                (<= d0 (sdepth (call-stack th s)))
                (fact-inv n0 d0 th s))
           (fact-inv n0 d0 th (step th s))))
```

The four verification conditions proved are interesting. The first considers the path from the entrance at pc `0` through the "base case" exit at pc `14`. The second considers the path from the entrance at pc `0` through the `INVOKESTATIC` instruction at pc `8`, to the recursive entrance in the frame pushed by the `INVOKESTATIC`. The third and fourth consider the paths from the two exits (at pcs `12` and `14`) to the exits in caller's frame. That is, one condition stays in the current frame, one condition relates the current frame to the newly pushed one upon recursive method invocation, and the other two relate (both exits from) the current frame to the caller's frame. The proof actually generates more than four cases because the assertions are inlined and cause case splits. This is akin to optimization and simplification built into the VCG.

The `run-to-return` theorem is exactly analogous to that shown for `*half-prog*` and is proved automatically.

```
(defthm fact-inv-run-to-return
```

```
  (implies (and (mono-threadedp th sched)
                (integerp d0)
                (< 1 d0)
                (fact-inv n0 d0 th s))
           (fact-inv n0 d0 th (run-to-return sched th d0 s))))
```

Finally, the main theorem is the desired partial correctness theorem for an `int` valued recursive `fact` method.

```
(defthm fact-main
  (let ((s1 (run-to-return sched th (sdepth (call-stack th s0)) s0)))
    (implies (and (intp n0)
                  (<= 0 n0)
                  (equal (pc (top-frame th s0)) 0)
                  (equal (locals (top-frame th s0)) (list n0))
                  (equal (program (top-frame th s0))
                         (cdddr *fact-def*))
                  (equal (sync-flg (top-frame th s0)) 'unlocked)
                  (equal (lookup-method "fact" "Demo" (class-table s0))
                         *fact-def*)
                  (< 1 (sdepth (call-stack th s0)))
                  (mono-threadedp th sched)
                  (< (sdepth (call-stack th s1))
                     (sdepth (call-stack th s0))))
             (equal (top (stack (top-frame th s1)))
                    (int-fix (! n0))))))
```

It may be read as follows. Let thread `th` of state `s0` satisfy the pre-conditions of the `fact` method. Suppose the depth of the call stack exceeds `1` so that a caller is known to be below the current frame. Run `s0` with an arbitrary mono-threaded schedule until it returns or the schedule is exhausted; call the final state `s1`. Suppose a return actually happened. Then the top of the operand stack in the caller's frame will contain `(int-fix (! n0))`.

Again, no instructions were counted and termination was not proved.

10 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.

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. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.
7. J. C. King. *A Program Verifier.* PhD thesis, Carnegie-Mellon University, 1969.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, Boston, MA., 1999.
9. 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.
10. 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.
11. 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.
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.