Proof Styles in Operational Semantics

Sandip Ray and J Strother Moore

Department of Computer Sciences,
University of Texas at Austin
{sandip, moore}@cs.utexas.edu
http://www.cs.utexas.edu/users/{sandip, moore}

Abstract. We relate two well-studied methodologies in deductive verification of operationally modeled sequential programs, namely the use of *inductive invariants* and *clock functions*. We show that the two methodologies are equivalent and one can mechanically transform a proof of a program in one methodology to a proof in the other. Both partial and total correctness are considered. This mechanical transformation is compositional; different parts of a program can be verified using different methodologies to achieve a complete proof of the entire program. The equivalence theorems have been mechanically checked by the ACL2 theorem prover and we implement automatic tools to carry out the transformation between the two methodologies in ACL2.

1 Background

This paper is concerned with relating strategies for deductive verification of sequential programs modeled operationally in some mathematical logic. For operational models, verifying a program is tantamount to characterizing the "initial" and "final" states of the machine executing the program and showing that every "execution" of the program starting from an initial state leaves the machine in a final state satisfying some desired "postcondition".

Deductive verification of sequential programs has traditionally used one of two reasoning strategies, namely the *inductive invariant* approach [1], and the *clock functions* or *direct* approach [2] respectively. While both the strategies guarantee correctness, to our knowledge no formal analysis has been performed on whether the theorems proved using one strategy are in any sense stronger than the other. However, it has been informally believed that the two strategies are fundamentally different and incompatible.

This paper analyzes the relation between these two strategies. We show that the informal beliefs are flawed in the following sense: In a sufficiently expressive logic, a correctness proof of a program in one strategy can be mechanically transformed into a proof in the other strategy. The result is not mathematically deep; careful formalization of the question essentially leads to the answer. But the question has been asked informally so often that we believe a formal answer is appropriate. Further, this equivalence enables independent verification of components of a program to obtain a proof of the composite whole. The

equivalence has been mechanically checked by the ACL2 theorem prover and such transformation tools implemented in ACL2.

To provide the relevant background, we first summarize the operational approach to modeling and reasoning about sequential programs, and describe the two strategies. We then discuss the contributions of this paper in greater detail. For ease of understanding, we adhere to traditional mathematical notations in this section. We later show how the concepts are made precise in the ACL2 logic.

1.1 Operational Program Models

Operational semantics involves characterization of program instructions by their effects on the states of the underlying machine. For simplicity, assume that a program is a sequence of instructions, and a state of the machine is a tuple describing the values registers, memory, stack, and so on. For every state s, let $pc(s)$ and $prog(s)$ denote the values of two special components in state s, the program counter and the current program respectively. These two components specify the "next instruction" executed by the machine at state s, which is the instruction in $prog(s)$ that is pointed to by $pc(s)$.

Meaning is assigned to an instruction by specifying, for every state s and every instruction i, the effect of executing i on s. This is formalized by a function $effect : S \times I \to S$, where S is the set of states, and I is the set of instructions. If the instruction is a `LOAD` its effect might be to push the contents of some specific variable on the stack and advance the program counter by some specific amount.

A special predicate *halting* characterizes the final states. A state s of M is *halting* if s is poised to execute an instruction i whose effect on s is a no-op, that is, $effect(s, i) = s$. Most programming languages provide explicit instructions like `HALT` whose *effect* on every state s is a no-op. In such cases, the machine halts when the instruction pointed to by the pc is the `HALT` instruction.

To reason about such operationally modeled programs, it is convenient to define a "next state function" $step : S \to S$. For every state s in S, the function $step(s)$ is the state produced as follows. Consider the instruction i in $prog(s)$ that is pointed to by $pc(s)$. Then $step(s)$ is defined to be $effect(s, i)$. Further, one defines the following iterated step function:

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

Program correctness is formalized by two predicates on the set S, namely a specified precondition *pre* characterizing the "initial" states, and a desired postcondition *post* characterizing the "final" states. In case of a sorting program, *pre* might specify that some machine variable contains a list l of integers, and *post* might specify that some (possibly the same) machine variable contains a list l' of integers that is an ordered permutation of l.

– **Partial Correctness:** Partial correctness involves showing that if, starting from a state that satisfies *pre*, the machine ever reaches a *halting* state, then *post* holds for such a *halting* state. Nothing is claimed if the machine does

not reach a *halting* state. Partial correctness can be formally expressed as the following formula:
$\forall s, n : pre(s) \wedge halting(run(s, n)) \Rightarrow post(run(s, n))$

- **Total Correctness:** Total correctness involves showing, in addition to partial correctness, that the machine, starting from a state satisfying *pre*, eventually halts:
$\forall s : pre(s) \Rightarrow (\exists n : halting(run(s, n)))$

1.2 Inductive Invariants

Inductive invariants constitute one strategy for proving the correctness theorems. The idea is to define a predicate that (i) is implied by the precondition, (ii) persists along every *step*, and (iii) implies the postcondition in a *halting* state. Thus, predicate *inv* is defined on the set S of states with the following properties:

1. $\forall s : pre(s) \Rightarrow inv(s)$,
2. $\forall s : inv(s) \Rightarrow inv(step(s))$, and
3. $\forall s : inv(s) \wedge halting(s) \Rightarrow post(s)$.

Then we can prove that for every state s satisfying *inv* and for every natural number n, $run(s, n)$ satisfies *inv*. This follows from property 2 by induction on n. The proof of partial correctness then follows from properties 1 and 3.

Total correctness is proved by a "well-foundedness" argument. A *well-founded structure* is a pair $\langle W, \prec \rangle$ where W is a set and $\prec$ is a partial order on the elements of W, such that there are no infinitely decreasing chains in W with respect to $\prec$. One defines a mapping $m : S \rightarrow W$, where $\langle W, \prec \rangle$ is well-founded, and proves the following property, in addition to 1, 2, and 3 above.

4. $\forall s : inv(s) \wedge \neg halting(s) \Rightarrow m(step(s)) \prec m(s)$.

The termination proof in the total correctness statement now follows from properties 2 and 4 as follows. Assume that the machine does not reach a *halting* state starting from some state s, such that $pre(s)$ holds. By property 2, each state in the sequence $\langle s, step(s), step(step(s)) \ldots \rangle$ satisfies *inv*. Then, by property 4, the sequence $\langle m(s), m(step(s)), m(step(step(s))) \ldots \rangle$ forms an infinite descending chain on W with respect to $\prec$. However, by well-foundedness, no infinitely descending chain can exist on W, leading to a contradiction.

An advantage of *inductive invariants* is that all the conditions involve only single steps of the program. The proofs are typically dispatched by case analysis without resorting to induction, once the appropriate *inv* is defined. However, the definition of *inv* is often cumbersome, since by condition 2, *inv* needs to be preserved along *every* step of the execution.

1.3 Clock Functions

A direct approach to proving total correctness is the use of *clock functions*. Roughly, the idea is to define a function that maps every state s satisfying *pre*, to a natural number that specifies the number of *steps* required to reach a *halting* state from s. Formally, $clock : S \rightarrow \mathbb{N}$ has the following two properties:

1. $\forall s : pre(s) \Rightarrow halting(run(s, clock(s)))$
2. $\forall s : pre(s) \Rightarrow post(run(s, clock(s)))$

Total correctness follows from these properties as follows. Termination proof is obvious, since for every state s satisfying *pre*, there exists an n, namely $clock(s)$, such that $run(s, n)$ is halting. Further, since by definition of *halting*, running from a *halting* state does not change the state, the state $run(s, clock(s))$ uniquely specifies the *halting* state reachable from s. By property 2 of *clock*, the state also satisfies *post*, showing correctness.

For specifying partial correctness, one weakens the properties 1 and 2 above so that $run(s, clock(s))$ satisfies *halting* and *post* only if a halting state is reachable from s. This can be achieved by adding the predicate $(\exists n : halting(run(s, n))$ as a conjunct in the antecedent of each property. The partial correctness theorem follows using exactly the correctness argument for total correctness.

Proofs involving *clock functions* typically require induction on the length of the execution. However, the definition of *clock* follows the control flow of the program [2, 3]; a user familiar with the branches and loops of a program can often define *clock* with relative ease, and the definition of *clock* provides a hint on the induction to be used in proving the correctness theorems.

1.4 Contributions of this Paper

Both *inductive invariants* and *clock functions* guarantee the same correctness theorems. However, the arguments used by the two strategies are different. The question, then, arises whether the theorems proved using one strategy are in any sense stronger than the other.

Why does one suspect that one strategy might be stronger than the other? Consider the total correctness proofs using the two strategies. In the *clock functions* approach, the function $clock(s)$ gives for every state s satisfying *pre*, the exact number of *steps* required to reach a *halting* state from s. One normally defines *clock* so that $clock(s)$ is the *minimum* number of *steps* required to reach a halting state from s. But that number is a precise characterization of the time complexity of the program! The *inductive invariant* proof, on the other hand, does not appear to require reasoning about time complexity, although it requires showing that the program eventually terminates.

Use of *inductive invariants* is a popular method for program verification. However, in the presence of a formally defined operational semantics, *clock functions* have been found useful. This method has been widely used in Boyer-Moore community, especially in ACL2 and its predecessor, Nqthm, to verify specialized architectures or machine codes [4–6]. Note that relatively few researchers outside this community have used *clock functions*; the reason is that relatively few researchers have pursued code-level mechanized formal proofs with respect to operational semantics. Operational semantics has been largely used by Nqthm and ACL2 since it permits the use of a general-purpose theorem prover for first-order recursive functions. Criticisms for *clock functions* have been typically expressed informally in conference question-answer sessions for the same reason:

given that no extant system supported code proofs for the specialized language presented, there was no motivation for comparing *clock functions* to other styles, but there was a nagging feeling that the approach required more work, namely reasoning about complexity when "merely" a correctness result is desired. The absence of written criticism of *clock functions* and the presence of this "nagging feeling" have been confirmed by an extensive literature search and discussions with authors of other theorem provers.

In this paper, our goal is to clarify relations between *inductive invariants* and *clock functions*. We show by mechanical proof that in the context of program verification, the two styles are equivalent in the sense that a proof in one style in one can be mechanically transformed into a proof in the other.

Besides showing the logical connection between the two proof styles, the equivalence theorems have an important practical implication: our results enable mechanical composition of proofs of different components of a program verified using different styles. Notwithstanding the logical equivalence of the two strategies as shown in this paper, one style might be simpler or more natural to derive "from scratch" than the other in a specific context. As an 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 desired 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 sufficient *inductive invariant* is cumbersome and requires a detailed understanding of the semantics of the different instructions. On the other hand, an *inductive invariant* proof might be more natural for verifying (2), by showing that each insertion in the sequence preserves the tree structure. However, traditional verification of a sequential composition of the two procedures (initialization followed by insertion) has had to adhere to a single style for both the procedures, often making proofs awkward and difficult. The results of this paper now allow verification of each component in the style most suitable for the component alone, by a trivial and well-known observation that *clock functions* can be naturally composed over different components.

Our equivalence theorems have been mechanically checked by the ACL2 theorem prover. Note that ACL2 (or indeed, any theorem prover) is not critical for proving the equivalence. ACL2 is used merely as a mechanized formal logic in deriving our proofs. However, since ACL2 routinely uses both strategies to verify operationally modeled programs, our theorems and the consequent proof transformation tools we implement, are of practical value in simplifying ACL2 proofs of large-scale programs. Our work can be easily adapted to any other mechanized logic like HOL [7] or PVS [8], that is expressive enough to specify arbitrary first-order formulas, and analogous tools for proof transformation can be implemented for theorem provers in such logics.

The remainder of this paper is organized as follows. In Section 2, we briefly discuss rudiments of the ACL2 logic. In Section 3, we formalize the two proof styles in ACL2 and discuss the mechanical proof of their equivalence. In Sec-

tion 4, we elaborate the framework to allow composition of proof strategies. In Section 5, we describe two macros for translation between proof strategies in ACL2. Finally, in Section 6, we discuss related work and provide some concluding remarks. The ACL2 proof scripts for all the theorems described here are available from the home page of the first author and will be distributed with the next version of the theorem prover. Note that although we adhere to the formal notation of ACL2 in the description of our theorems, this paper assumes no significant previous exposure to the ACL2 logic, and only a basic familiarity with Lisp.

2 The ACL2 Logic

In this section, we briefly describe the ACL2 logic. This provides a formal notational and reasoning framework to be used in the rest of the paper. Full details of the ACL2 logic and its theorem proving engine can be found in [9, 10].

ACL2 is essentially a first-order logic of recursive functions. The inference rules constitute propositional calculus with equality and instantiation, and well-founded induction up to ϵ_0. The language is an applicative subset of Common Lisp; instead of writing $f(a)$ as the application of function f to argument a, one writes `(f a)`. Terms are used instead of formulas. For example, the term:

```
(implies (natp i) (equal (nth i (update-nth i v l)) v))
```

represents a basic fact about list processing in the ACL2 syntax. The syntax is quantifier-free; formulas may be thought of as universally quantified over all free variables. The term above specifies the statement: "For all i, v and l, if i is a natural number, then the i-th element of the list obtained by updating the i-th element of l by v is v."

ACL2 provides axioms to reason about Lisp functions. For example, the following axiom specifies that the function `car` applied to the `cons` of two arguments, returns the first argument of `cons`.

Axiom:
```
(equal (car (cons x y)) x)
```

Theorems can be proved for axiomatically defined functions in the ACL2 system. Theorems are proved by the `defthm` command. For example, the command:

```
(defthm car-cons-for-2 (equal (car (cons x 2)) x))
```

directs the theorem prover to prove that for every `x`, the output of the function `car` applied to the `cons` of `x` and the constant `2`, returns `x`.

ACL2 provides three *extension principles* that allow the user to introduce new function symbols and axioms about them. The extension principles constitute (i) the *definitional principle* to introduce total functions, (ii) the *encapsulation principle* to introduce constrained functions, and (iii) the *defchoose principle* to introduce Skolem functions. We briefly sketch these principles here. See [11] for a detailed description of these principles along with soundness arguments.

Definitional Principle: The *definitional principle* allows the user to define new total functions in the logic. For example, the following form defines the factorial function `fact` in ACL2.

```
(defun fact (n) (if (zp n) 1 (* n (fact (- n 1)))))
```

The effect is to extend the logic by the following *definitional axiom*:

Definitional Axiom:
```
(fact n) = (if (zp n) 1 (* n (fact (- n 1))))
```

Here `(zp n)` returns `nil` if `n` is a positive natural number, and otherwise `T`. To ensure consistency, ACL2 must prove that the recursion terminates [12]. In particular, one must exhibit a "measure" m that maps the set of arguments in the function to some set W, where $\langle W, \prec \rangle$ forms a well-founded structure. The proof obligation, then, is to show that on every recursive call, this measure "decreases" according to relation $\prec$. ACL2 axiomatizes a specific well-founded structure, namely the set of ordinals below ϵ_0: membership in this set is recognized by an axiomatically defined predicate `e0-ordinalp`, and a binary relation `e0-ord-<` is axiomatized in the logic as an irreflexive partial order in the set.

Encapsulation Principle: The *encapsulation principle* allows the extension of the ACL2 logic with partially defined constrained functions. For example, the command below introduces a function symbol `foo` with the constraint that `(foo n)` is a natural number.

```
(encapsulate (((foo *) => *))
  (local (defun foo (n) 1))
  (defthm foo-returns-natural (natp (foo n))))
```

Consistency is ensured by showing that some (total) function exists satisfying the alleged constraints. In this case, the constant function that always returns `1` serves as such "witness". The effect is to extend the logic by the following *encapsulation axiom* corresponding to the constraints. Notice that the axiom does not specify the value of the function for every input.

Encapsulation Axiom:
```
(natp (foo n))
```

For a constrained function f the only axioms known are the constraints. Therefore, any theorem proved about f is also valid for a function f' that also satisfies 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'. Then, a derived rule of inference, *functional instantiation* specifies that for any theorem θ one can derive the theorem $\hat{\theta}$ provided one can prove $\hat{\phi}$ as a theorem. In the example, since the constant `10` satisfies the constraint for `foo`, if `(bar (foo n))` is provable for some function `bar`, functional instantiation can be used to prove `(bar 10)`.

Defchoose Principle: The *defchoose principle* allows introduction of Skolem functions in ACL2. To understand this principle, assume that a function symbol `P` of two arguments has been introduced in the ACL2 logic. Then the form:

```
(defchoose exists-y-witness y (x) (P x y))
```

extends the logic by the following axiom:

Defchoose Axiom:
```
(implies (P x y) (P x (exists-y-witness x)))
```

The axiom states that *if* there exists some `y` such that `(P x y)` holds, then `(exists-y-witness x)` returns such a `y`. Nothing is claimed about the return value of `(exists-y-witness x)` if there exists no such `y`. This provides the power of first-order quantification in the logic. For example, we can define a function `exists-y` such that `(exists-y x)` is true if and only if there exists some `y` satisfying `(P x y)`. Notice that the theorem `exists-y-suff` below is an easy consequence of the defchoose and definitional principles.

```
(defun exists-y (x) (P x (exists-y-witness x)))
(defthm exists-y-suff (implies (P x y) (exists-y x)))
```

ACL2 provides a construct `defun-sk` that makes use of the defchoose principle to introduce explicit quantification. For example, the form:

```
(defun-sk exists-y (x) (exists y (P x y)))
```

is merely an abbreviation for the following forms:

```
(defchoose exists-y-witness y (x) (P x y))
(defun exists-y (x) (P x (exists-y-witness x)))
(defthm exists-y-suff (implies (P x y)  (exists-y x)))
```

Thus `(exists-y x)` can be thought of specifying as the first-order formula: `(`$\exists$`y : (P x y))`. Further, `defun-sk` supports universal quantification `forall` by exploiting the duality between existential and universal quantification.

3 Proof Strategies

Operational semantics have been used in ACL2 (and other theorem provers) for modeling complex programs in practical systems. For example, formal models of programs in the Java™ Virtual Machine (JVM) have been formalized in ACL2 [3, 13]. Operational models accurately reflecting the details of practical computing systems are elaborate and complex; however such elaborations are not of our concern in this paper. For this presentation, we assume that a state transition function `step` of a single argument has been defined in the logic, possibly following the approach described in Section 1.1, which, given the "current state" of the underlying machine, returns the "next state". We also assume the existence of unary predicates `pre` and `post` specifying the preconditions and postconditions respectively, and the predicate `halting` below specifying termination.

```
(defun halting (s) (equal s (step s)))
```

We now formalize the *inductive invariant* and *clock function* proofs in this framework. The theorems we describe here are straightforward translations of our descriptions in Sections 1.2 and 1.3. In particular, an *inductive invariant* proof of partial correctness constitutes the following theorems for some function `inv`.

```
(defthm pre-implies-inv (implies (pre s) (inv s)))
(defthm inv-persists (implies (inv s) (inv (step s))))
(defthm inv-implies-post
  (implies (and (inv s) (halting s)) (post s)))
```

A total correctness proof also requires a "measure function" `m` and the following theorems:[1]

```
(defthm m-is-ordinal (e0-ordinalp (m s)))
(defthm m-decreases
  (implies (and (inv s) (not (halting s)))
           (e0-ord-< (m (step s)) (m s))))
```

Analogously, a *clock function* proof in the logic comprises a definition of the function `clock` and theorems that express an ACL2 formalization of our discussions in Section 1.3. A total correctness proof constitutes the following theorems:

```
(defthm clock-run-is-halting
  (implies (pre s) (halting (run s (clock s)))))
(defthm clock-run-is-post
  (implies (pre s) (post (run s (clock s)))))
```

where the function `run` is simply the iterated application of `step` as defined below:

```
(defun run (s n) (if (zp n) s (run (step s) (- n 1))))
```

Finally, a partial correctness theorem is the "weakening" of the above theorems, requiring them to hold only if there exists a `halting` state reachable from `s`.

```
(defthm clock-run-is-halting
  (implies (and (pre s) (halting (run s n)))
           (halting (run s (clock s)))))
(defthm clock-run-is-post
  (implies (and (pre s) (halting (run s n)))
           (post (run s (clock s)))))
```

To prove equivalence between the two proof styles we use the encapsulation principle; that is, we encapsulate function symbols `step`, `inv`, `m`, `clock`, constrained to satisfy the corresponding theorems, and show that the constraints associated with *inductive invariants* can be derived from the constraints associated with *clock functions* and vice-versa. In Section 5, we will use these generic proofs to implement tools to translate proofs in one style to the other.

[1] We have used the set of ordinals below ϵ_0 with the relation `e0-ord-<` instead of a generic well-founded structure $\langle W, \prec \rangle$, since this is the only well-founded set axiomatically defined in ACL2. However, the structure of the ordinals is of no consequence here, and our proofs can be translated in terms of any well-founded structures.

3.1 Equivalence Theorems

To obtain a *clock function* proof from *inductive invariants* we will define a `clock` that "counts" the number of `steps` until a `halting` state is reached. Recall from our discussions in Section 1.2 that if `inv` is an inductive invariant that holds for some state `s` then `inv` holds for all states reachable from `s`. In ACL2, such a statement is formalized by the theorem `inv-run` below and can be proved by induction on `n`.

```
(defthm inv-run (implies (inv s) (inv (run s n))))
```

Hence if `clock` can be defined to count the number of `steps` to `halting`, then the obligations for a *clock function* proof will follow from the properties of *inductive invariants*, in particular, `pre-implies-inv` and `inv-implies-post`. For total correctness, the following recursive definition provides such a count.

```
(defun clock (s)
  (if (or (not (inv s)) (halting s)) 0
    (+ 1 (clock (step s)))))
```

The crucial observation is that the function `clock` above is admissible to the ACL2 logic under the definitional principle. The recursion is justified by the theorems provided by the termination proofs in the *inductive invariants* approach, namely, that there exists a "measure", in this case `m`, that maps the arguments of `clock` to some well-founded set (ordinals) and decreases (according to `e0-ord-<`) in the recursive call.

The situation is a bit more subtle for partial correctness, since there may be no such measure. In this case, therefore, we use the defchoose principle to define the appropriate `clock` as follows:

```
(defun-sk exists-pre-state (s)
  (exists (init i j)
          (and (pre init) (natp i) (natp j) (equal s (run init i))
               (halting (run init j)))))
(defun clock (s)
  (if (exists-pre-state s)
      (mv-let (init i j) (exists-pre-state-witness s)
              (nfix (- j i)))
    0))
```

The function `nfix` above is the identity function if its argument is a natural number; otherwise it returns `0`. The function `clock` can be interpreted as follows. If there is a state `init` satisfying `pre` and numbers `i` and `j` such that `s` is reachable from `init` in `i steps` and a `halting` state in `j steps`, then *clock* returns `(- j i)`; otherwise it returns `0`. But if `s` is indeed reachable from some state satisfying `pre`, and a `halting` state is reachable from `s`, then `(- j i)` represents the number of `steps` to reach a `halting` state from `s`. Thus in this case the proof obligations for *clock functions* follow from the *inductive invariants* constraints analogous to total correctness. The return value of `0` is arbitrary when no `halting` state is reachable from `s`, and can be replaced by any value.

To obtain an *inductive invariant* proof from *clock functions*, we define the predicate `inv` expressing the following property: A state `s` satisfies `inv` if and only if `s` is reachable from some state `init` satisfying `pre`. Notice that the obligations `pre-implies-inv` and `inv-persists` are trivial for such a predicate. Further, the *clock functions* proofs guarantee that a `halting` state reachable from some `pre` state must also satisfy `post`: Recall that the theorem `clock-run-is-post` guarantees that for every state `s` satisfying `pre`, the `halting` state `(run s (clock s))` reached (for partial correctness under the hypothesis that some `halting` state is reachable) from `s` must satisfy `post`. Since by definition of `halting`, `stepping` from a `halting` state does not change the state, it follows that *any* `halting` state reachable from `s` must satisfy `post`. We formalize this using defchoose principle as follows:

```
(defun-sk inv (s)
  (exists (init n)
          (and (pre init) (natp n) (equal s (run init n)))))
```

For total correctness, we define the measure `m` by determining the number of `steps` to reach the first `halting` state:

```
(defun m-aux (s i clk)
  (if (or (halting s) (>= i clk) (not (natp i)) (not (natp clk)))
      (nfix i)
    (m-aux (step s) (+ i 1) clk)))
(defun m (s) (m-aux s 0 (clock s)))
```

The function `m` returns a natural number (and hence an ordinal). Further, for any state `s` reachable from some `pre` state `init`, if `s` is not `halting`, then `(m (step s))` is exactly `1` less than `(m s)`, since `m` merely counts the number of `steps` to reach the first `halting` state. Hence `m` decreases along a `step`, justifying termination.

4 Verifying Program Components

We now show how to generalize our framework to allow different components of a program to be verified using different strategies. The thorny issue in verifying components of a program separately arises from the use of the predicate `halting` in the framework. Recall that the predicate `halting` specifies termination in a very strong sense, specifying that `(step s)` must be equal to `s`! However, when a program completes a specific procedure, it merely returns control to the calling procedure. Our verification framework is modified as follows in order to be meaningful for verification of program components.

1. In *clock functions*, for a state `s` poised to execute a program component of interest, `(clock s)` must precisely characterize the number of `steps` from `s` to the "corresponding" exit.
2. In *inductive invariants*, `inv` needs to "persist" only along the `steps` which execute the instructions in the component of interest.

To formalize this, we introduce a new predicate `external` to indicate the "exit" of the program control from the component of interest, and modify the proof obligations for each style. For technical reasons, we first restrict the predicate `pre` so that `pre` states also do not satisfy `external`:

```
(defthm pre-not-external (implies (pre s) (not (external s))))
```

The restriction, though introduced for technical reasons, is natural, as the subsequent discussions will show. We now "strengthen" the *clock functions* strategy, so that `clock` specifies the *minimum* number of `steps` to reach an `external` state. This is achieved by adding the following constraints to `clock`.

```
(defthm clock-is-natural (natp (clock s)))
(defthm clock-is-minimal
  (implies (and (pre s) (natp n) (external (run s n)))
           (<= (clock s) n)))
```

These constraints, along with those described in Section 3 modified to use `external` instead of `halting`, comprise the *clock function* proof of an individual component. Notice that `(clock s)` now characterizes the number of `steps` to reach the *first* `external` state from `s`. A casual reader might complain that this is not general enough to characterize proofs of complicated program components like recursive procedures. After all, if `external` specifies the return from a procedure, then for a state `s` poised to invoke a recursive procedure, the first `external` state reached from `s` does not represent the "corresponding" return! However, notice that `external` can be an *arbitrary* function of state; for example, a legitimate definition of `external` for a recursive procedure is that *pc* points to the instruction after the return and the stack of recursive calls is empty.

To capture the notion of *first* `external` state in the *inductive invariants* framework, we first attach the constraint that `inv` does not hold for `external` states. The intuition, then, is that `inv` should hold for every state starting from a `pre` state "until" an `external` state is encountered. We can then decide if `(step s)` is the first `external` state, by checking if both `(inv s)` and `(external (step s))` hold. The proof obligations for *inductive invariants* are modified as follows:

```
(defthm inv-persists
  (implies (and (inv s) (not (external (step s)))) (inv (step s))))
(defthm inv-implies-post
  (implies (and (inv s) (external (step s))) (post (step s))))
(defthm m-decreases
 (implies (and (inv s) (not (external (step s))))
          (e0-ord-< (m (step s)) (m s))))
```

We have surveyed several proofs of system models, including JVM proofs in ACL2 [3]. For all non-trivial programs, the verification has been decomposed into "component proofs", and the proof of each component could always be described in terms of the frameworks above.

In this generalized framework again, one can derive the proof obligations in one approach from a proof in the other. Space does not permit a thorough discussion of the generalized equivalence proofs, but the informal intuition is the

same as in Section 3. The complete ACL2 script for the proof of this equivalence is available to the interested reader from the web-page of the first author.

An immediate nice consequence of the generalized equivalence results is the capability of mechanically composing proofs of different components of a program obtained using different styles into a proof of the complete program. Consider, for example, *sequential composition*, that is, a program composed of two sequential code blocks `A` and `B`. An important, though trivial, observation about *clock functions* is that when two blocks of code are sequentially executed, the `clock` for the composite program is given by summing the clocks for each component. Thus if `clock-A` and `clock-B` are used in deriving *clock function* proofs of `A` and `B` respectively, then the composite `clock` is as given below:

```
(defun clock (s) (+ (clock-A s) (clock-B (run s (clock-A s)))))
```

More involved and complicated compositions involving branches, loops, and recursion are possible and can be built out of sequential compositions.

5 Switching Proof Strategies

Since the equivalence theorems have been proved using the encapsulation principle, functional instantiation can be used to automatically transform proofs from one style to another. Assume that for a specific program modeled by the "step function" `c-step`, and functions `c-pre`, `c-external` and `c-post`, an *inductive invariant* proof has been constructed for total correctness by introducing some invariant predicate `c-inv` and measure `c-m`. The following directive then proves the equivalent of `clock-run-is-post` for the concrete system.

```
(defthm c-run-post
  (implies (c-pre s) (c-post (c-run s (c-clock s))))
  :hints ((''Goal'':use ((:functional-instance clock-is-post
                          (inv c-inv) (pre c-pre) ....)))))
```

Thus `c-run-post` is proved by instantiating the "abstract" functions in theorem `clock-run-is-post` with the "concrete" functions provided. Recall that to use functional instantiation, ACL2 must prove that the concrete functions, namely `c-pre`, `c-step`, etc., satisfy the constraints imposed by the abstract counterparts. But such constraints are exactly the proof obligations for *inductive invariants*, which have been already dispatched for the concrete functions.

We have developed two macros `inv-to-clock` and `clock-to-inv` to transform proofs from one strategy to another, along with basic tools for automatic composition of sequential blocks. While the actual implementation is more elaborate, the basic approach is to automatically generate "concrete" theorems like the one above and prove them by functionally instantiating the abstract proofs.

6 Conclusion

Operational semantics for modeling programs was proposed by McCarthy [1]. The *inductive invariants* framework is often regarded as the "classical approach"

in formal verification of programs. Numerous operational system models have since been mechanically verified using inductive invariants in theorem provers like ACL2 [9, 10], HOL [7], and PVS [8], and tools implemented to facilitate such proofs. On the other hand, in the presence of operational models, especially in Boyer-Moore theorem provers, *clock functions* have found greater success, at least for total correctness proofs. Similar proofs have been done using other theorem provers too, though less frequently [14].

We do not advocate one proof style over another. Our goal is to allow the possibility of going "back and forth" between the two styles; thus a program component can be verified using the strategy that is natural to the component, independent of other components. This is particularly important in theorem proving, where a user needs to guide the theorem prover in the proof search. Avoiding the necessity to adhere to a monolithic strategy solely for composition makes the user-dependent aspect of theorem proving simpler and more palatable. Note however, that the definition of `clock` becomes complicated when proofs of a number of components are composed. This complication is of no consequence for correctness; however, to reason about efficiency it is imperative to obtain a "simpler" `clock`. Our work does not address that issue. For reference, Golden [private communication] uses the simplification engine of ACL2 to produce simpler `clocks`, which can provide effective solutions to such concerns.

Our techniques are applicable to operational models alone. Another approach called *denotational semantics* [15–17] models programs in terms of transformation of predicates rather than states as we described. Our framework cannot be directly applied to that approach. Indeed, the notion of *invariants* we use is tied to an operational view, and cannot be formally reconciled with the denotational approach without an extra-logical *verification condition generator*. However, [18] gives a way of proving partial correctness using *inductive invariants* incurring exactly the proof obligations for a denotational approach. Consequently, this work shows that *clock functions* can be derived using the same proof obligations as well. But operational models are requisites for both results.

Our work also emphasizes the power of quantification in ACL2. The expressiveness of quantification has gone largely unnoticed in ACL2, the focus being on "constructive" definitions using recursive equations. The chief reasons for this focus are executability, and amenability for induction. However, in practical verification, it is useful to be able to reason about a generic model of which different concrete systems are "merely" instantiations. Quantifiers are useful for reasoning about generic models. For example, for some state `s`, assume we want to consider the property of "some state `p` from which `s` is reachable". It is then convenient to posit that "some such `p` exists", and use the `witness` as the specific `p` to reason about. We and others have found this convenient in diverse contexts, in formalizing *weakest precondition*, and reasoning about pipelined machines [19].

Acknowledgements The authors benefitted from discussions with the ACL2 group at UT Austin. We particularly thank Jeff Golden, Matt Kaufmann, Robert

Krug, Erik Reeber, Rob Sumners, and Vinod Vishwanath for several comments and suggestions.

References

1. McCarthy, J.: Towards a Mathematical Science of Computation. In: Proceedings of the Information Processing Congress. Volume 62., North-Holland (1962) 21–28
2. Boyer, R.S., Moore, J.S.: Mechanized Formal Reasoning about Programs and Computing Machines. In Veroff, R., ed.: Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, MIT Press (1996) 141–176
3. Moore, J.S.: Proving Theorems about Java and the JVM with ACL2. In Broy, M., Pizka, M., eds.: Models, Algebras, and Logic of Engineering Software, Amsterdam, IOS Press (2003) 227–290
4. Bevier, W.R.: A Verified Operating System Kernel. PhD thesis, Department of Computer Sciences, The University of Texas at Austin (1987)
5. Young, W.D.: A Verified Code Generator for a Subset of Gypsy. Technical Report 33, Computational Logic Inc. (1988)
6. Flatau, A.D.: A Verified Implementation of an Applicative Language with Dynamic Storage Allocation. PhD thesis (1992)
7. Gordon, M.J.C., Melham, T.F., eds.: Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press (1993)
8. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In Kapoor, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of LNAI., Springer-Verlag (1992) 748–752
9. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
10. Kaufmann, M., Manolios, P., Moore, J.S., eds.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
11. Kaufmann, M., Moore, J.S.: Structured Theory Development for a Mechanized Logic. Journal of Automated Reasoning **26** (2001) 161–203
12. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1975)
13. Liu, H., Moore, J.S.: Executable JVM Model for Analytical Reasoning: A Study. In: ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines, and Emulators, San Diego, CA (2003)
14. Wilding, M.: Robust Computer System Proofs in PVS. In Holloway, C.M., Hayhurst, K.J., eds.: Fourth NASA Langley Formal Methods Workshop. Number 3356 in NASA Conference Publication (1997)
15. Floyd, R.: Assigning Meanings to Programs. In: Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs. Volume XIX., Providence, Rhode Island, American Mathematical Society (1967) 19–32
16. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12** (1969) 576–583
17. Dijkstra, E.W.: Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. Language Hierarchies and Interfaces (1975) 111–124
18. Moore, J.S.: Inductive Assertions and Operational Semantics. In Geist, D., ed.: 12th International Conference on Correct Hardware Design and Verification Methods (CHARME). Volume 2860 of LNCS., Springer-Verlag (2003) 289–303
19. Ray, S., W. A. Hunt, Jr: Deductive Verification of Pipelined Machines Using First-Order Quantification. In Alur, R., Peled, D.A., eds.: Computer-Aided Verification (CAV). Volume 3114 of LNCS., Boston, MA, Springer-Verlag (2004) 31–43