Proof Styles in Operational Semantics

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
moore@cs.utexas.edu

December 10, 2003

Abstract

We relate two well-studied methodologies, use of *inductive invariants* and *clock functions*, in deductive verification of operationally modeled sequential programs. We prove that the two methodologies are equivalent in the following sense: If the logic used is sufficiently expressive then 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 in this context. We also show that the mechanical transformation is compositional in that different parts of a program can be verified using different methodologies to achieve a complete proof of the entire program. Our proof of equivalence has been mechanically verified by the ACL2 theorem prover [17, 16] and we discuss automatic tools in the form of ACL2 macros to carry out the mechanical transformation between the two methodologies in the context of ACL2 theorem proving.

1 Background

This paper is concerned with relating strategies for deductive verification of sequential programs. We consider a generic verification framework for reasoning about sequential programs modeled operationally in some mathematical logic. This framework has been widely studied in the literature starting from the seminal work by McCarthy [26], and successfully used in reasoning about practical programs and computing systems. In this framework, correctness of a program is given by a statement of the following form: *If the program runs on some machine from a state satisfying some given precondition, then the state of the machine on termination of the program satisfies some desired postcondition.*

Deductive verification of sequential programs has traditionally used one of two reasoning strategies, which we call the *inductive invariant* approach, and the *clock functions* or *direct* approach respectively. The inductive invariant approach is more widely known, but is probably used less frequently in connection with a formally defined operational semantics. While both the strategies guarantee some form of the correctness statement above, no formal analysis has been performed to our knowledge on whether the theorems proved using one strategy are in any sense stronger than those proved using the other. However, it has been informally believed that the two strategies are fundamentally different and incompatible. In particular, we are not aware of any technique for verifying parts of a program by different strategies and obtaining a correctness theorem for the complete program.

This paper seeks to formally analyze and answer such questions on relations between the two strategies. In particular, we show that the informal beliefs are flawed in the following sense: In a sufficiently expressive logic, given a proof of a specific program property using one strategy, it is possible to mechanically transform the proof into a proof using the other strategy. Our proof has been mechanically checked by the ACL2 theorem proving system [17, 16], and our transformation tools have been implemented as ACL2 macros.

To provide the relevant background for our work, we first summarize the operational approach on modeling and reasoning about programs, and an overview of the two proof strategies. In particular, we discuss formalizations of the correctness statement of programs above, and show how each strategy guarantees a proof of correctness in the context of verification of a specific program. To facilitate understanding of the two strategies, we deliberately consider a very simplified view of operational models and correctness proofs. Then we discuss the contributions of this paper in greater detail.

1.1 Operational Program Models

Consider a program Π executing in a machine M. The program is typically a fixed sequence of instructions. The *operational semantics* approach to modeling the executions of Π has been succinctly described by McCarthy [26] as: "The meaning of a program is defined by its effect on the state vector".

To reason about the program Π operationally in a mathematical logic, therefore, we need some formalization of the "states" of the machine M. Loosely, the *states* of M is a tuple describing available computational resources, for example registers, memory, stacks, and so on. Typically every state s contains two special state components, namely the program counter $pc(s)$, and the current program $prog(s)$. For this discussion, we assume that for every state s, the component $prog(s)$ will always contain the program Π. The two components $pc(s)$ and $prog(s)$, determine the "next instruction"i to be executed by the machine, which is the instruction in Π at the position pointed by pc(s). The state s is is then said to be *poised* to invoke the instruction i.

Meaning is assigned to an instruction by defining, for every state s and every instruction i, the effect of executing i on s. This notion is often concisely expressed by defining a function $\mathit{effect}: S \times I \rightarrow S$, where S is the (possibly infinite) set of states of M, and I is the set of instructions. For example, 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. More complicated instructions might affect many parts of the state.

A state s of M will be referred to as the *halting state* if s is poised to execute an instruction i whose effect on s is a no-op, that is, $\mathit{effect}(s,i) = s$. 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 program counter is the `HALT` instruction.

To formally reason about such operationally modeled programs, it is convenient to define a "next state function" $step: S \rightarrow S$. 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 by $pc(s)$. Then $step(s)$ is defined to be $\mathit{effect}(s,i)$. Further, one defines the following iterated step function:

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

In order to formalize the correctness theorem we referred to at the beginning of this section, we would assume three predicates *pre*, *post* and *halting* on the set S. The predicates *pre* and *post* are the specified preconditions and postconditions respectively, and the predicate *halting* is `true` of a state s if and only if $step(s) = s$. To understand the pre and post conditions, consider a program Π that purportedly sorts a list of integers. The precondition then might specify that some machine variable contains a list l of integers, and the postcondition might specify that some (possibly the same) machine variable contains a list l' of integers that is an ordered permutation of l.

Two correctness notions are normally used for proofs of programs. The notions are often naturally described as *partial* and *total* correctness of the program under inspection.

- **Partial Correctness:** Partial correctness involves showing that if, starting from a state satisfies the precondition, the machine ever reaches a halting state, then the postcondition holds for such halting state. Nothing is claimed if the machine does not reach a halting state. Formally, the partial correctness theorem can be specified by 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 the precondition, eventually halts. Thus proving total correctness is tantamount to verifying partial correctness along with a termination proof. Formally, the termination proof can be specified by the following formula:

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

1.2 Inductive Invariants

Design of *inductive invariants* is one strategy for proving the correctness theorems above. Roughly, the idea is to define a predicate that (i) is implied by the precondition, (ii) persists along every *step* of M, and (iii) implies the postcondition in a halting state. More precisely, one defines a predicate *inv* on the set S of states of machine M with the following properties:

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

The three properties above guarantee the partial correctness theorem described in Section 1.1. To see this, note that the following lemma follows from property 2 by induction on n.

Lemma 1 *For every state that satisfies inv and for every natural number n, the state $\mathit{run}(s, n)$ also satisfies inv. In other words, $\forall s, n : \mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{run}(s, n))$.*

The proof of the partial correctness theorem now follows from property 1, lemma 1, and instantiation of property 3 with s by $\mathit{run}(s, n)$.

To obtain a total correctness theorem as described in Section 1.1 using the inductive invariant strategy, one normally applies an argument based on "well-foundedness". Roughly, a *well-founded structure* is a pair $\langle W, \prec \rangle$ where W is a set and $\prec$ is a binary relation on the elements of W, such that $\prec$ is an irreflexive partial order on the elements of W and there are no infinitely decreasing chains in W with respect to $\prec$. In order to derive total correctness, one establishes a mapping $m : S \rightarrow W$, where $\langle W, \prec \rangle$ is well-founded, and establishes the following property relating *inv* and m, in addition to the three properties above.

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

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 $\mathit{pre}(s)$ holds. By property 2, each state in the sequence $\langle s, \mathit{step}(s), \mathit{step}(\mathit{step}(s)) \ldots \rangle$ satisfies *inv*. Further, by property 4, the sequence $\langle m(s), m(\mathit{step}(s)), m(\mathit{step}(\mathit{step}(s))) \ldots \rangle$ 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.

1.3 Clock Functions

A direct approach to proving the total correctness theorem in Section 1.1 is the *clock function* strategy. Roughly, the idea is to define a function that maps every state s in M that satisfies the precondition, to a natural number, describing the number of *steps* required to reach a halting state from s. More precisely, one defines a function $\mathit{clock} : S \rightarrow \mathbb{N}$ with the following property:

- $\forall s : \mathit{pre}(s) \Rightarrow \mathit{halting}(\mathit{run}(s, \mathit{clock}(s))) \wedge \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$.

To see that the total correctness theorem follows from this property, note that the termination follows from the property of *clock*, since for every state s, such that $\mathit{pre}(s)$ holds, there exists at least one n, namely $\mathit{clock}(s)$ such that $\mathit{run}(s, n)$ is halting. To prove correctness, one needs the following two lemmas which are provable from the definitions of *run* and *halting* using induction on n.

Lemma 2 *Running from a halting state does not change the state. In other words,* $\forall s, n : \mathit{halting}(s) \Rightarrow \mathit{run}(s, n) = s$.

Lemma 3 *For every state* s*, running* m *times from* s *and then running* n *times from* $\mathrm{run}(s, m)$ *is the same as running* $(m+n)$ *times from* s*. In other words,* $\forall s, m, n : \mathit{run}(\mathit{run}(s, m), n) = \mathit{run}(s, m+n))$.

Correctness follows from these two lemmas and the property of *clock* as follows. Assume that $\mathit{pre}(s)$ and there exists a natural number n such that $\mathit{halting}(\mathit{run}(s, n))$. Without loss of generality, assume that $n \leq \mathit{clock}(s)$, since the other case is analogous. We need to show $\mathit{post}(\mathit{run}(s, n))$. If $n = \mathit{clock}(s)$, then the proof follows from the property of *clock*. Otherwise, by Lemma 3 we have:

$$\mathit{run}(s, \mathit{clock}(s)) = \mathit{run}(\mathit{run}(s, n), \mathit{clock}(s) - n) \tag{1}$$

Since $\mathit{run}(s, n)$ is halting, by Lemma 2, the right hand side of the equation reduces to $\mathit{run}(s, n)$. Now the result follows by the property of *clock*.

For specifying partial correctness using clock functions, one weakens the property of the clock function to imply *halting* and *post* only under the condition that a halting state is reachable from s. In other words, one defines the function $\mathit{clock} : S \rightarrow \mathbb{N}$ to have the following property instead of the property above.

- $\forall s : \mathit{pre}(s) \wedge (\exists n : \mathit{halting}(\mathit{run}(s, n))) \Rightarrow \mathit{halting}(\mathit{run}(s, \mathit{clock}(s))) \wedge \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$.

Using this weaker definition of *clock*, the partial correctness theorem as described in Section 1.1 now follows using exactly the correctness argument above.

1.4 Contributions of this Paper

It should be clear from the discussions above that 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 either 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 particular, in the *clock functions* approach, the function $\mathit{clock}(s)$ gives for every state s satisfying the precondition, the exact number of *steps* required to reach a halting state from s. In fact, one normally defines the function *clock* such that $\mathit{clock}(s)$ 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![1] 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.[2]

Modern sequential programs are normally verified mechanically, that is, by the use of a trusted computer program or *theorem prover* responsible for checking the correctness theorems and assisting in the proof process. In this context, both *inductive invariants* and *clock functions* have independent but orthogonal advantages. For example, none of the obligations in the *inductive invariant* strategy involves reasoning about more than one *step* of the system model. Hence once a suitable predicate *inv* is obtained, the proof obligations can often be dispatched without resorting to inductive proofs. In particular, if the number of states of M is finite and tractable, it is even possible to verify the obligations automatically by exhaustive case analysis. The proof obligations for the *clock function* strategy, on the other hand, normally require induction on the length of the execution to be dispatched. The issue with *inductive invariant* proofs, however, arises in

[1] One might argue that in the proofs we just described, *clock* was not required to be a characterization of time complexity but only an upper bound. While that is the case in the simplified view of the situation, we will see in Section 4, that for composing proofs the clock functions need to be modified and be precisely the time required for the program to reach from a state satisfying the precondition to the first `return` of the program.

[2] It might also appear that the *inductive invariant* strategy is stronger than *clock functions* in the following sense: For proving property 2 for an inductive invariant, one needs to show that for every state s which satisfies *inv*, the state $\mathit{step}(s)$ also satisfies *inv*. This seems to indicate that the predicate *inv* needs to explicitly characterize every state s reachable from a state p that satisfies the precondition. However, recent work by Moore [30] shows that this is not the case. In particular, it is possible to define the predicate *inv* only by specifying assertions on certain "cutpoints", when a partial correctness proof is desired.

the context of coming up with the appropriate predicate *inv* which satisfies all the obligations. For example, the obligation 2 for *inv* requires that for every state s that satisfies *inv*, the state *step*(s) must satisfy *inv* as well. On the other hand, a user familiar with how the different branches of a program operate can often come up with a clock function with relative ease. Further, as verifications with clock functions reveal, for example in [28], the definition of the function *clock* often provides a hint on the inductive approach to be taken in verifying a correctness theorem involving clocks.

We note that use of both strategies has been popular in mechanical verification of sequential programs. In fact, almost all reported mechanical proofs of operationally modeled practical sequential programs that we are aware of, has principally followed one of the two strategies.[3] The *inductive invariant* strategy has often been considered the "classical approach" in proving program correctness, especially in the context of partial correctness.

The *clock function* strategy has been principally used in theorem-proving community in the Boyer-Moore style using the NQTHM theorem prover [4] and its industrial-strength successor ACL2 [17, 16]. While reasoning using *inductive invariants* is possible, and indeed, has been successfully accomplished in these systems especially for partial correctness proofs, the *clock function* technique "plays" to the strength of the theorem prover in using induction when the total correctness is desired.

In spite of extensive use of the two techniques in program verification, especially using mechanical theorem provers, we are not aware of any formal analysis on the relation between the two proof techniques. However, informally, several researchers, including the authors, have been concerned about the requirement of reasoning about time complexity of a program in the *clock function* approach when "merely" a correctness theorem is desired. In fact, this issue has often been a serious grievance encountered by the authors in describing ACL2 proofs of programs to the mechanical verification community.

In this paper, we therefore, attempt to settle the question on the relation between the two proof techniques. We discuss the following two questions:

1. Are the two strategies equivalent?
2. If they are equivalent, then is it possible to verify different parts of a program using different strategies to obtain a complete proof of the entire program?

Our answer to both the questions is positive. In particular, we show that the two proof techniques are equivalent in the following sense: If the logic used is sufficiently expressive, namely, if quantified first order statements are definable in the logic, then it is possible to mechanically transform a proof in one strategy to a proof in the other, in a way that allows composition of proofs. Note that even if the answer to question 1 is positive it does not directly imply a positive answer to question 2. In particular, when only a part of a program is verified, for example one procedure, then the simplified picture of proof we showed above breaks down, at least for *clock function* arguments. In that argument, we used the definition of *halting* to justify that the *clock function* strategy indeed guarantees correctness. However, if only a single procedure of a program is considered, then execution of such a procedure will normally not bring the program into a halting state but return control to the caller. We discuss a variant of the framework in Section 4, which lets us deal with composition of strategies.

Our proofs have been mechanically checked by the ACL2 theorem prover. While ACL2, and indeed, any theorem prover, is unnecessary for our equivalence proof, we decided to mechanically verify our proofs for two reasons. First, ACL2 is a practical logic with theorem proving support in which program models such as we described can be and have been successfully verified. Hence the use of the ACL2 logic for carrying out this verification clearly demonstrates that the expressibility required of the modeling logic to state and prove the equivalence is achievable by practical logics used in program verification. In fact, in terms of expressibility, the logic of ACL2 is limited compared to other practical logics supported by theorem provers like HOL [14] and PVS [31]. Hence, our proof demonstrates that the equivalence is provable in these stronger logics,

[3] We say "almost" since some reported work uses model checking techniques [8, 9], which are fundamentally different. However, such techniques are more commonly used for verification of reactive or non-terminating system models, especially when the models are of finite states. Such techniques, therefore, are more appropriate for verifying hardware models rather than sequential programs.

and correspondingly, mechanical translation tools can be designed in these theorem provers to decompose a program verification into proofs of component parts that can use different strategies to obtain a complete proof of the composite whole. Secondly, as we mentioned, ACL2 is a theorem prover which has routinely and successfully used both strategies for verification of sequential programs. Our equivalence proofs have enabled us to design simple ACL2 macros that allow automatic translation of proofs in different strategies. We believe that such automatic translation will assist in simplifying ACL2 proofs of large programs in future, by allowing the user to choose between different strategies for verifying components of a program without any requirement to think whether a single strategy will be useful for the entire system.

In the sequel, we provide details of our work. In Section 2, we briefly review rudiments of the ACL2 logic and discuss the ACL2 formalization of *inductive invariant* and *clock functions* for sequential program models. Our overview is not comprehensive, and our objective is merely to provide a sufficient logical framework to discuss our theorems. In Section 3, we provide a simplified description of the equivalence proof in the ACL2 logic. In Section 4, we elaborate on our framework to allow composition of proof strategies. In Section 5, we describe the two macros for translation between proof strategies. Finally, in Section 6, we summarize our work and provide some concluding remarks. We note that this paper assumes no previous acquaintance with the ACL2 logic or the theorem prover. The concepts we require are described in passing. However, previous exposure to proofs about sequential programs in ACL2 is useful to appreciate our specific choices of modeling and reasoning framework. We also point out that the ACL2 theorems we describe in this paper are available as `books` for the current version of the theorem prover and are available upon request from the first author.

2 Program Verification in ACL2

In this section, we outline the ACL2 approach to model and reason about sequential programs. The modeling approach we describe here essentially provides a formalization of the operational models we discussed in Section 1.1 in the ACL2 logic. To make the formalization precise, we sketch the basic features of the ACL2 logic used in our models and proofs in Section 2.1. We then discuss the operational models for sequential programs in the logic of ACL2, and provide a simple example of how the two strategies are used in verifying such models. Our discussions in this section are not comprehensive, either in the description of the logic of the theorem prover, or in the presentation of operational models and reasoning strategies. The purpose of the section is merely to sketch the essential features of the logic required in our proofs and provide a very rough overview of these features on actual system examples. We refer the reader interested in learning the theorem prover and investigating practical examples of ACL2 verification to the extensive online documentation of the URL: `http://www.cs.utexas.edu/users/moore/acl2/` and to numerous books and papers written about it. Many such books and papers are referenced in the URL above, and we will also cite some of them in this section for more elaboration of the concepts described here. Further, we emphasize that this section does not discuss any novel idea of the paper but simply serves as the summary of already existing work in ACL2. Readers familiar with the ACL2 logic and system models in ACL2 might skip this section without loss of continuity.

2.1 The ACL2 Logic

ACL2 is essentially a first-order logic of recursive functions. The language supported by the theorem prover is the applicative subset of Common Lisp [12]. Therefore, instead of writing $f(a)$ as the application of function f on argument a, one would write `(f a)` in the ACL2 logic. 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. For example, the term above can be thought of as specifying 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."

From a logical perspective, ACL2 provides an axiomatization of all primitive functions of Common Lisp supported by the system. For example, Lisp provides functions `car`, `cdr` and `cons` to implement lists. To reason about lists, ACL2 provides the semantics to such functions by specifying axioms like the following for `cons`, `car`, and `cdr`, among others. A comprehensive description of the axiomatically defined Common Lisp primitives appears in [18].

Axiom 1:
`(equal (car (cons x y)) x)`
Axiom 2:
`(equal (cdr (cons x y)) y)`

The axioms above specify that the function `car`, applied to the `cons` of two arguments, returns the first argument of `cons`, and correspondingly, `cdr` returns the second argument. The axioms can thus be thought of as providing formal semantics to the primitive Common Lisp functions. On the other hand, the semantics provided by ACL2 differs from the Common Lisp semantics in an important way. Many Common Lisp functions are *partial*; values are specified for the functions only when the functions are applied to inputs from a specific intended domain. For example, the value returned by `car` is specified by Common Lisp only in the case that the argument of `car` is a non-empty list (recognized by the predicate `consp`) or `nil`. In particular, since the number `3` is not a `consp`, the Common Lisp Standard does not specify what value `(car 3)` must return. Implementations of Common Lisp are free to return any value, and in fact, are not required to return the same value every time the function `car` is called with argument `3`. However, all functions in ACL2 are total, and can, in fact, be called on any argument. To achieve consistency in the logic, ACL2 therefore provides additional axioms that specify a unique value for every call of each Common Lisp function. This is achieved by specifying for every function a "natural" default value to return when the function is not called on an argument in its intended domain. For example, the following "completion axiom" for `car` specifies that the function returns `nil` if it is called on an argument which is not a `consp`. ACL2 provides a logical "story" of its relation in Common Lisp by the notion of "guards". For a thorough discussion on the relation between Common Lisp and ACL2, see the topic `guard` in the online documentation.

Axiom 3:
`(implies (not (consp x)) (equal (car x) nil))`

Theorems can be proved for axiomatically defined functions in the ACL2 system. (In addition, as we will see shortly, the user can define his/her own functions in ACL2 via the extension principles and prove theorems about them.) Theorems are proved by the `defthm` command. For example, the command:

```
(defthm sort-is-permutation
  (perm (sort x) x))
```

asks the theorem prover to prove that for every `x`, the output of the function `sort` applied to `x` produces a "permutation" of `x`,[4] and store this theorem in its internal database by the name `sort-is-permutation`.

The inference rules for the logic basically constitute propositional calculus with equality and instantiation, along with well-founded induction up to ϵ_0. Theorems proved in the logic are stored in a database and can be used to prove subsequent theorems. The informed user can therefore "guide" the theorem prover to prove complicated theorems by a careful and judicious choice of lemmas. A detailed description of the inference engine of ACL2 and the numerous heuristics used by the theorem prover to apply a previously proved lemma in guiding the search for a proof of a current theorem is beyond the scope of this paper and the interested reader is referred to the online documentation of the system for fuller discussions.

In addition to axiomatically specifying Common Lisp primitives, ACL2 provides three *extension principles* that allow the user to extend the logic by introducing new function symbols and adding axioms about such

[4]Note that before attempting to prove this theorem, one must first define the functions `sort` and `perm` via the extension principles below, since those functions are not axiomatically specified Common Lisp primitives.

function symbols in a way consistent with the logic.[5] The extension principles constitute (i) the *definitional principle* for adding new function definitions, (ii) the *encapsulation principle* to introduce a constrained function definition, and (iii) the *defchoose principle* to introduce Skolem functions. Since we will make extensive use of these principles, particularly `defchoose`, in our equivalence arguments, we provide a sketch of these principles here. A detailed description of these principles along with arguments for their soundness appears in [19].

Definitional Principle

The definitional principle allows the user to define new function definitions in the logic. For example, the following function `fact` is a definition of factorial in ACL2.

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

The "body" of the `defun` form must be an ACL2 term consisting of known function symbols (that is, function symbols previously introduced using one of the three extension principles or axiomatically defined for Common Lisp) and the new symbol being introduced, applied to the specified number of arguments. The variables used in the term must be a subset of the variables in the "argument list" of the `defun` form.

The effect of such a definition is the extension of the logic by the addition of a *definitional axiom.*

Definitional Axiom:

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

(Remark: In our discussions we sometimes write `t1 = t2` for two terms `t1` and `t2`. This notation is simply an abbreviation for `(equal t1 t2)` and is used merely for a clearer exposition of the concepts to the reader familiar with more traditional infix notations.)

To ensure the consistency of the extended logic, the definitional principle in ACL2 requires a proof that the recursion terminates [5]. 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 for some binary relation $\prec$ on W. The proof obligation, then, is to show that on every recursive call, this measure "decreases" according to relation $\prec$.

To add the definitional axiom for `fact` above, ACL2 therefore is required to find a measure `m` where for every n `(m n)` is an element of some set W, such that $\langle W, \prec \rangle$ is a well-founded, and `(m (- n 1))` $\prec$ `(m n)` when `(zp n)` is false. For proving well-foundedness, ACL2 uses a specific well-founded structure called $\langle$*Ord*, `e0-ord-<`$\rangle$, where *Ord* is the set of ordinals below ϵ_0 [2, 24], and `e0-ord-<` is the extension of the ordinary $<$ relation to the ordinals. ACL2 provides a special predicate `e0-ordinalp` to recognize whether its argument is a member of *Ord*. The proof of termination therefore involves defining the function `m` and proving the following two theorems.

```
(defthm m-is-ordinal (e0-ordinalp (m n)))
(defthm m-less (implies (not (zp n)) (e0-ord-< (m (- n 1)) (m n))))
```

Such functions exist, and in fact, the following function `nfix` satisfies the two theorems above, since, in particular, it returns a natural number, and hence an ordinal.

```
(defun nfix (n) (if (and (integerp n) (<= 0 n)) n 0))
```

We briefly remark here on arguments regarding well-foundedness in ACL2. In the ACL2 logic, the only well-founded structure axiomatically defined is the set of ordinals with relation `e0-ord-<`, and the only arguments known to show that a structure $\langle W, \prec \rangle$ to be well-founded by showing an embedding of W in `Ord` and justifying that $\prec$ "reduces" to `e0-ord-<` in this embedding. We believe ACL2 is not expressive enough for specification of generic well-founded structures other than via embeddings into ordinals. This explains why we characterize the termination proof in *inductive invariants* approach in Section 2.3 using the ordinals

[5]The user can add an arbitrary ACL2 formula as an axiom in the logic using the `defaxiom` command. But the practice of adding arbitrary `defaxiom` commands is discouraged because of the possibility of unsoundness. ACL2 gives no guarantee on the validity of theorems once a user adds a `defaxiom` event in ACL2.

rather than by a generic well-founded relation as we discussed in Section 1.2. However, the special structure of the ordinals is immaterial to our proofs and our theorems will work unchanged in a more expressive logic which allows generic well-founded structures.

Encapsulation Principle

The *encapsulation principle* allows the extension of the logic of ACL2 with the introduction of undefined functions constrained to satisfy certain specified properties. For example, suppose we want to define a function `foo` of a single argument, and the only constraint that we want `(foo n)` to satisfy is that it returns a natural number. Notice that the constraint does not uniquely specify a function. The following command extends the logic with such a function `foo` which is only known to return a natural number.

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

The effect of the form above is to extend the logic with a function symbol `foo` known only to satisfy the proposed constraints. The axiom added to the logic is the following:

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

In order to ensure that the extension is consistent, ACL2 needs to be convinced that some (total) function exists that can satisfy the proposed constraints. This is achieved by exhibiting such a function to the theorem prover, in this case simply the (constant) function that always returns `1`

Since for a constrained function f the only axioms known about f are the constraints, it stands to reason that any theorem proved about f is also valid for a function f' that also satisfies the constraints. To make this precise, 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 called *functional instantiation* [3] says that from any theorem θ one can derive the theorem $\hat{\theta}$ provided one can prove $\hat{\phi}$ as a theorem. That is, one may infer a theorem about f' from an analogous theorem about f provided f' satisfies the constraint ϕ. In the example, notice that the function `nfix` we described above satisfies the the constraint for `foo`. Hence, if `(bar (foo n))` is a formula that can be proved for some function `bar`, then one will be able to use functional instantiation to prove `(bar (nfix n))`.

Encapsulation and functional instantiation provide a higher order aspect in the ACL2 logic, which is essentially first order. One typically produces a "generic theory" based on encapsulated functions which can then be "instantiated" by specific functions satisfying the constraints. In Section 5 we will see an application of this approach in our translation tool for proof strategies. More detailed discussions on the use of functional instantiations can be found in [25].

Defchoose Principle

The *defchoose principle* allows the user to introduce "Skolem functions" into the ACL2 logic. 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 merely claims 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`.

The use of `defchoose` allows quantified functions to be introduced in the logic of ACL2. This comes somewhat as a surprise even to some ACL2 users, although the topic has been documented in the online documentation of the system, and discussed in [19]. The reason for the surprise is that the syntax of ACL2 is quantifier-free! As we have discussed earlier, variables are assumed to be implicitly universally quantified. What does a quantifier mean in such a logic then?

The answer is that although every term in ACL2 is implicitly universally quantified, one might often require quantifiers *inside* the body of a term to express certain properties. To understand this, assume that we have defined the function `(P x y)` above, and consider defining a predicate `Exists-y` with the following informal first-order property: `(Exists-y x)` returns `true` if and only if there is a `y` such that `(P x y)` is `true`. (For example, if `(P x y)` is defined to be `nil` then no such `y` exists.) Notice that since the universe of ACL2 objects is infinite, indeed uncountable, it is not possible to define the function by recursively checking for every object `y` in the ACL2 universe whether `(P x y)` is true, until one either finds an object `y` such that `(P x y)` does not hold or the universe is exhausted. (Remark: Defining a recursive function is possible and in fact, often achieved if `P` has the property that a finite number of `y`s is sufficient to deduce the answer one way or the other. But it is not possible for an arbitrary predicate `P`.)

However, the use of `defchoose` allows a simple definition of `Exists-y` as follows:

```
(defun Exists-y (x) (P x (exists-y-witness x)))
```

Here `exists-y-witness` is the function symbol introduced by the `defchoose` form above. Using the Defchoose axiom for `exists-y-witness` and the Definitional axiom for `choose-y`, it is easy to prove the following theorem in ACL2:

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

In other words, if one can find a `y` such that `(P x y)` is `true` then one can immediately conclude `(exists-y x)`. Further, from the definitional axiom of `exists-y`, one can conclude that if `(exists-y x)` is `true`, then there is at least one `y`, namely `(exists-y-witness x)`, such that `(P x y)` is true. The function `exists-y-witness` can therefore be regarded as a "Skolem witness" for the property `P` if such a witness exists. We point out that 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:[6]

```
(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)))
```

Hence, by the discussion above, the predicate `(exists-y x)` can be thought of as describing the first-order statement: $(\exists$`y` : `(P x y)`). Henceforth we will treat the `defun-sk` form above as specifying this first-order statement. In addition, to existential quantification, the `defun-sk` form also supports universal quantification `forall` by making the following observation in first order logic about any binary predicate f:

$$(\forall x : f(x, y)) = \neg(\exists y : (\neg f(x, y)))$$

2.2 Computing Systems in ACL2

We now briefly describe operational models of sequential programs in the ACL2 logic. This approach has been successfully used in the ACL2 theorem prover for modeling and reasoning about sequential programs of practical complexity. For example, Liu and Moore [21] discusses how to use this approach to obtain a fairly

[6]In Section 5, we will briefly discuss macros in the ACL2 system that enable the user to specify such abbreviations. Macros will be used in our translation tool to convert one proof strategy to another. However, since macros are merely "syntactic sugar" and not part of the logic of the theorem prover, we refrain from discussing them in this section.

accurate model of the Java^{TM} Virtual Machine (JVM) in ACL2 and reason about JVM programs. Similar approaches have been reported for modeling and verifying other system models, for example microprocessors [7, 22, 27, 33], compilers and program translators [27, 13], and concurrent programs [32]. This section simply provides a nugget for the idea in order to concretize our informal description in Section 1.1 and set the background for our result in the context of ACL2 verification of system models. The interested reader will find more details of this approach in these reported work.

Recall from our discussions in Section 1.1 that a formal operational semantics consists of some formal representation of the states of the underlying machine executing the program in the logic, along with a formal description of the state transition function `step`. States are often modeled in ACL2 using constant length lists, although it is possible (depending on the objective) to use more efficient structures like single threaded objects [6] or "records" [20] provided with ACL2. In the list representation, the elements at the different positions of a list represent the values of the different components, like pc, program, stack, registers, and so on. Each of these components, in turn, can possibly be built out of other components represented using different data structures in ACL2. For example, the program is typically modeled as a constant list of instructions, and the pc as a natural number. An instruction is often modeled as a list whose first component is the opcode and the rest is the list of arguments used by the instruction. For example, a simplified JVM model in ACL2 [29] models the "goto" instruction as a list of two components, `(GOTO k)`, where `GOTO` is the opcode and `(pc + k)` is the "target" of the goto statement.

To formally model the effects of the different instructions on the states of the machine, one typically defines functions that take a machine state and an instruction and returns the "updated state". For example, consider the `GOTO` instruction above. The effect of executing the instruction might be to change the value stored in the pc to the value specified by the target of the instruction. Assuming that the pc is the first (0-th) component of a state in its list representation, the following ACL2 code models the effect of executing the instruction for an arbitrary state `s`.[7]

```
(defun execute-GOTO (s inst)
  (let* ((pc-component 0)
         (pc (nth pc-component s))
         (target (+ pc (second inst)))
         (new-pc target)
         (new-state (update-nth pc-component target s)))
   new-state))
```

More complicated instructions might involve updates of more components of the state. Elaborate examples of functions modeling a large subset of JVM can be found in [28, 29, 21].

The state transition function `step` can then be defined by choosing the next instruction pointed to by the pc, and updating the state by calling the appropriate "update function". The typical state transition function is shown as follows:

```
(defun execute-inst (s inst)
  (let ((opcode (first inst)))
    (case opcode
      (LOAD (execute-LOAD s inst))
      (GOTO (execute-GOTO s inst))
      .....
      (t s))))
```

[7]The functions described in [29] are slightly more complicated than the simplified presentations provided in this paper. The main reason is that the machine modeled there is multithreaded; as a result, the functions `execute-opcode` and `step` are not functions of a single argument, that is, the current state, but also have another argument, namely the thread `th` that is taking the `step`. The technique described in this paper do not pertain to multithreaded models, and indeed, we are not sure of a "reasonable" approach using *clock functions* for verifying concurrent programs. However, for sequential programs, even in multithreaded machine models, the second argument is of no significance. One can introduce a predicate `mono-threadedp` [30] so that the same thread is always picked. Our techniques can be applied in such cases by appropriately modifying the clock to return a `mono-threaded` sequence of threads instead of a natural number.

```
(defun step (s)
  (let* ((pc-component 0)
         (prog-component 1)
         (pc (nth pc-component s))
         (program (nth prog-component s))
         (inst (nth pc program)))
     (execute-inst s inst)))
```

The crux of program verification in ACL2 is to derive properties of the state transition function `step` defined above. We first note that the iterated state transition function `run` and the predicate `halting` can be defined naturally in ACL2 as follows:

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

Based on the above terminology, we now discuss a very basic program to illustrate the two proof strategies in verification of sequential programs in ACL2.

2.3 Verifying Sequential Programs in ACL2

Consider a simple machine language program that simply adds `1` to the the value of a variable `x` and stores the result in a variable `y`. In C-like syntax, the operation of this program can be coded by the single statement: `y=x+1;`. The following program, denoted by `*incr-program*`, is a formalization for the simplified JVM model [28] in ACL2. Note that the choice of this machine model and program is arbitrary and only serves to provide an example of how ACL2 is used for reasoning about models using the different approaches. We omit the several ACL2 definitions defining the semantics of every instruction in the program. We provide comments on the side of every instruction describing its action.

```
(defconst *incr-program*
  '((ICONST_1)   ;; Push the constant 1 on the stack
    (ILOAD_1)    ;; Push the content of location 1 (x) on stack
    (IADD)       ;; Pop the top two objects, add them and push the result.
    (ISTORE_2))) ;; Store the top object to location 2 (y)
```

We want to prove that if the machine starts from a state s in which `*incr-program*` is present in the `program` component of s and the value stored in location `1` in s is a natural number, then on termination of the program, the value at location `2` is equal to `1` plus the value at location `1`. Further, we want to show total correctness, that is, the system eventually reaches a halting state. The statement of correctness is specified in the precondition and postcondition defined below. Assume that for a state `s`, `(pc s)` and `(program s)` returns the contents of the program counter and program, and `(location i s)` returns the contents of location i.

```
(defun pre (s)
  (and (equal (program s) *incr-program*)
       (equal (pc s) 0))
       (natp (location 1 s)))
(defun post (s)
   (equal (location 2 s)
          (+ (location 1 s) 1)))
```

We will look at how the two strategies help us prove the correctness, and discuss the basic differences between them.

Inductive Invariant Proof

Recall from Section 1.2 that an inductive invariant proof of a total correctness theorem involves defining functions `inv` and `m`. The ACL2 properties we discuss are natural formalizations of the proof obligations in ACL2 and should be self-explanatory. Note that as we discussed in Section 2.1, we use the special set `Ord` of ordinals with the relation `e0-ord-<` instead of a general well-founded structure.

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

The following two functions `inv` and `m` satisfy the theorems.

```
(defun inv (s)
  (let ((pc (pc s))
        (x (location 1 s))
        (y (location 2 s)))
     (case pc
       (0 (natp x))
       (1 (natp x))
       (2 (and (natp x) (equal (top (stack s)) x)))
       (3 (equal (top (stack s)) (+ 1 x)))
       (t (equal y (top (stack s)))))))
(defun m (s) (nfix (- 4 (pc s))))
```

We note that the proof is trivial once the appropriate functions `inv` and `m` are defined. However, even for this simple program, the definition of `inv` is slightly convoluted. The reason is that one must specify some assertion for *every* value of the program counter. This is required since the proof theorem `inv-persists` involves showing that for every state for which `inv` holds, the next state must satisfy `inv`. Moore [30] shows an alternative way of specifying invariants which does not involve assertion at every program counter value, but at selected "cutpoints". The approach however, still requires one to justify that the assertion at one cutpoint is strong enough to prove the assertion at the "next cutpoint". While the result in [30] significantly reduces the search for invariants, specifying and proving invariants is still the most significant component of program verification using *inductive invariant* proofs.

Clock Function Proof

To prove the program correct using *clock functions*, recall that we need to define the function `clock` so that the following theorem can be proved.

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

For this trivial program, one can define the function `clock` based on the following observations. The only way a state satisfies `pre` is if the program counter is `0` and it is executing `*incr-program*`. However, since the program is a straight-line code with no branches or loops, all the instructions in the program will be exhausted in time equal to the length of the program which is `4`. Thus the program will terminate after `4` steps, starting from a `pre` state. Hence, the function `clock` can be defined as:

```
(defun clock (s) 4)
```

The two theorems can simply be proved by simplification of the term `(run s 4)` under the assumption `(pre` s). In this case, and indeed, for any program fragment that does not involve loops, the theorem can be proved without using induction on the length of the execution. However, for programs containing loops, the clock function strategy needs to use induction on the number of iterations in the loop.

To illustrate this point, consider a simple loop program that tests some condition (c s) to enter a loop, and halts when the condition is false. Assume also the loop consists of a straight line code of α instructions, and the pre and post conditions are both defined by the same predicate (loop-predicate s). In other words, we want to show that if the program starts from a state s such that (loop-predicate s) holds then on termination (loop-predicate s) still holds.

```
(defthm loop-correct
  (implies (loop-predicate s)
           (loop-predicate (run s (loop-clock s)))))
```

To define the clock for such a theorem, recall that the clock simply represents the number of steps to reach the halting state. Since every iteration of the loop contains α instructions, the clock we will want to define is as follows:

```
(defun loop-clock (s)
  (if (not (c s)) 0
      (+ α (loop-clock (run s α)))))
```

Notice that *if* we can define this function, then the typical approach will be to prove by induction on the number of iterations in the loop that loop-predicate holds every time the program counter reaches the entry to the loop [28].[8] In other words, one would typically prove loop-correct using induction and establishing the base and induction cases as follows:

Base Case:

```
(implies (not (c s))
         (implies (loop-predicate s)
                  (loop-predicate (run s (clock s)))))
```

Induction Case:

```
(implies (and (c s)
              (implies (loop-predicate (run s α))
                       (loop-predicate
                        (run (run s α)
                             (clock (run s α))))))
         (implies (loop-predicate s)
                  (loop-predicate (run s (clock s)))))
```

The soundness of this induction in fact, guaranteed by the definition of loop-clock. In particular, since loop-clock is recursive, there must be a measure m such that (m s) is an ordinal, and under the condition (c s), the term (m (run s α)) is smaller (according to e0-ord-<) than (m s). The existence of m justifies that the induction hypothesis in the Induction Case above is indeed a "smaller" instance of the theorem. The rest, therefore, follows from the Principle of Induction.

The discussion above should make it clear, that while proofs about programs using *clock functions* are sometimes inductive, the induction is usually based on the same principles as the recursive function to specify the clock. Hence, the proof critically depends on the definition of the function. Notice that the function loop-clock we discussed can be admitted to ACL2 logic only after we can justify, according to the Definitional Principle, that the recursive calls eventually terminate. However, the recursive calls will terminate if and only if the program itself terminates! In fact, at least for this one-loop program, the function loop-clock is a precise description of the time complexity of the loop.

We do not know which approach is better or more natural in the context of proofs of sequential programs in ACL2 and indeed, in any logic, using the operational approach. The invariant proof involves describing

[8] If loop-predicate does not have that property, then it can usually be strengthened into a predicate inv-predicate such that inv-predicate implies loop-predicate and inv-predicate indeed holds every time the program reaches the entry of the loop.

strong predicates that will enable one to verify the alleged invariant to be indeed an inductive invariant. The clock function approach involves finding a function `clock` that specifies the time complexity of the program in a very strong way. In specific contexts one approach might be better or easier than the other. However, we will claim in the next section that finding a proof in one approach immediately implies in the logic a proof in the other approach. In particular, for total correctness, for an arbitrary function `step`, if the theorems listed under *Inductive Invariant Proof* can be proved using some functions `inv` and `m`, then it is possible to define a function `clock` so that the theorems listed under *Clock Function Proof* can be proved for the same function `step`, and vice versa. Analogous results are shown for partial correctness. Logically, therefore, one can use either approach in deriving correctness results in a specific context, and use the transformation to get a proof in the other strategy.

3 Equivalence of Proof Strategies

In this section, we discuss our equivalence proofs. To make our proofs generic, we will merely assume that the system model has been provided by specifying some state transition function `step`; the actual implementation of `step` might involve some of the considerations we described in Section 2.2, but the implementation plays no part in the equivalence proofs.

Our approach to model the generic *inductive invariant* and *clock function* proofs is to use the Encapsulation Principle in ACL2 to constrain functions `inv`, `clock`, and (in case of total correctness) `m`, with appropriate constraints. For example, the constraints for an inductive invariant proof of total correctness are exactly the properties described in the theorems in Section 2.3 for a generic function `step` instead of the special function described there. In Section 5 we will describe macros that allow us to instantiate the generic proofs using functional instantiation for specific system models.

3.1 Equivalence for Total Correctness

Assume that we have a unary function `step` modeling the next state function of some computing system, and that we are provided unary predicates `pre` and `post`. Recall from our discussions in Section 2.3, that an *inductive invariant* proof of total correctness involves defining functions `inv` and `m` and proving the following theorems:

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

On the other hand, the *clock function* proof involves the definition of the function `clock` and dispatching the following proof obligations:

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

To show their equivalence, we will prove that the theorems in one strategy can be derived from the theorems in the other. In other words, from definition of `inv` and `m` known to satisfy the theorems under the *inductive invariant* strategy, one can define `clock` which satisfies the theorems under *clock function* strategy, and vice-versa.

Inductive Invariants To Clock Functions

Assume that for some functions `step`, `pre`, `post`, `inv`, and `m`, the theorems `pre-implies-inv`, `inv-persists`, `inv-implies-post`, `measure-is-ordinal` and `m-decreases` have been proved. We now define the function `clock` as follows:

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

Roughly, the function `clock` simply `steps` the machine until it encounters a state that is either `halting`, or does not satisfy `inv`. To admit this recursive function according to the Definitional Principle, ACL2 needs to prove that some measure function `(measure s)` exists, which maps the states of the system to the ordinals, and which decreases in every recursive call. In other words,

```
(e0-ordinalp (measure s))
(implies (and (inv s) (not (halting s)))
         (e0-ord-< (measure (step s))
                   (measure s)))
```

But at least one such measure already exists, namely `(m s)`, by the theorems `measure-is-ordinal` and `m-decreases`! Hence the function `clock` above is admissible to the logic.

Once this function is defined, the theorems `clock-run-is-halting` and `clock-run-is-good` follow as simple consequence. The crucial observation is the theorem `inv-run`, which is essentially a restatement of Lemma 1 in the ACL2 logic.

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

The theorem `clock-run-is-halting` now follows by the definition of `clock`. In particular, the term `(run s (clock s))` must be either `halting`, or not satisfy `inv`. However, if `s` satisfies `inv` then by theorem `inv-run`, `(run s (clock s))` must in particular satisfy `inv` and thus must also be `halting`. The theorem `clock-run-is-good` now follows by using `inv-implies-good`, and instantiating `s` in the theorem by `(run s (clock s))`.

An ACL2 user will note that the proof is extremely trivial. Indeed, the crucial aspect of proofs in *clock function* strategy is in the definition of `clock` in a way that it is admissible to the ACL2 logic. Our chief contribution is the observation that `clock` can be admitted by using the same termination argument as used in an *inductive invariant* proof to guarantee eventual termination of the program.

Clock Functions to Inductive Invariants

To obtain an *inductive invariant* proof from a *clock function* proof, we will make use of Defchoose Principle in ACL2. Assume that the functions `step`, `pre`, `post`, and `clock` are defined and the theorems `clock-run-is-halting` and `clock-run-is-good` have been verified. We now define the functions `inv` and `m` as follows:

```
(defun-sk inv (s) (exists (init n) (and (pre init) (natp n) (equal s (run init n)))))
(defun m-aux (s i clk)
  (if (or (halting s) (>= i clk)
          (not (natp i))
          (not (natp clk)))
      (nfix i)
    (m (step s) (+ i 1) clk)))
(defun m (s) (m-aux s 0 (clock s)))
```

The function `(inv s)` posits that there exists state `init` and a natural number `n`, such that `init` satisfies the precondition `pre`, and `s` can be reached by `stepping` the machine `n` steps. On the other hand, the function `m` simply runs the machine until the first `halting` state is reached, and counts the number of `steps`.

The proof of `pre-implies-inv` and `inv-persists` follows from the definition of `inv`. Observe that for every state `s` that satisfies `(pre s)`, there exists at least one pair $\langle$`init`,`n`$\rangle$, namely $\langle$`s`$, 0\rangle$, such that running from `init` for `n` steps returns `s`. Moreover, if `(inv s)` holds, then assume $\langle$`init`,`n`$\rangle$ form the witness for `s`. Then `(inv (step s))` holds and is witnessed by $\langle$`init`,`n+1`$\rangle$.

The proof of `inv-implies-good` and `m-decreases` requires the property of `halting` that was specified in Lemma 2. The following is the restatement of the lemma in ACL2, and can be easily proved by induction on `n`.

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

Using this theorem, one can now easily show that for a halting state `s` reachable from a state `init` satisfying `pre`, `s` must be equal to `(run s (clock s))`.

```
(defthm halting-is-clock
  (implies (and (pre init) (halting (run init n)))
           (equal (run init (clock init)) (run init n))))
```

The proof of this theorem is easy by considering the three cases, namely (i) `n` $<$ `(clock s)`, (ii) `n` $=$ `(clock s)`, and (iii) `n` $>$ `(clock s)`, and the theorem `halting-run`. The proof of `inv-implies-good` now follows from `halting-clock` and `clock-run-is-good`. Further, the theorem `m-decreases` follows by simply observing that for any `init` satisfying `pre`, if `(run init n)` is not halting and `n` is a natural number, then `n` must be less than `(clock s)`.

```
(defthm not-halting-less-than-clock
  (implies (and (natp n) (pre init) (not (halting (run init n))))
           (< n (clock s))))
```

Note that this proof made extensive use of quantification in ACL2. Quantification is a feature of ACL2 whose power is largely neglected, even by serious ACL2 users. We found this feature invaluable in this and other work, when the object is to build generic theories for system models in ACL2. In particular, we do not believe that it is possible to verify the equivalence theorems we discussed in a logic which does not allow quantification. Another example of successful use of quantification using the Defchoose Principle is shown in [23] by Manolios and Moore. They discuss an approach to introduce partial functions in ACL2 satisfying certain classes of recursive equations. These and other independent work by us and others indicate that quantification can play to the strength of the theorem prover for proving non-trivial theorems about complicated system models.

3.2 Equivalence for Partial Correctness

To consider partial correctness, we need to weaken the proof obligations in both strategies. In particular, for *inductive invariants*, we will assume a proof that consists only of the definition of the predicate `inv` and the theorems `pre-implies-inv`, `inv-persists`, and `inv-implies-good` above. For *clock functions*, we weaken the theorems `clock-run-is-halting` and `clock-run-is-good` by adding hypothesis that posits that halting states are reachable from states satisfying the precondition. The modified *clock function* theorems are as follows:

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

We now show how we can derive the equivalence results for partial correctness.

Inductive Invariants to Clock Functions

To prove *clock function* theorems from *inductive invariants*, we define the function `clock` as follows. Note that all the functions we discuss in this section make heavy use of quantification and the Defchoose Principle.

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

Roughly, the function can be interpreted as follows. If there exists a state `init` and natural numbers `i` and `j` such that `(run init i)` is `s`, and `(run init j)` is halting, then the function returns the difference between `j` and `i`. If no such `init`, `i`, and `j` exist then it returns `0`.

Informally, one can think of the `clock` for every state `s` that will eventually reach a terminating state to measure the number of `steps` left to reach the terminating state. In other words, assume that from some state `init`, the state `s` is reachable in `i` steps, and some terminating state is reachable in `j` steps, for $\mathtt{j} \geq \mathtt{i}$. Then, the number of steps required to reach some terminating state from `s` must be `(- j i)`, which is defined to be `(clock s)`. Note that the value returned from `clock` if no terminating state is reachable from `s` is not critical, and the choice of `0` is arbitrary.

Now consider the question of proving the theorem `clock-run-is-halting` for partial correctness as specified above. The theorem roughly posits that *if* there is a number `n` that takes some state `s` satisfying `pre` to a halting state, then `(clock s)` takes `s` to a halting state too. To prove this statement, notice that `(exists-pre-state s)` is definitely `true` for such `s`, since there exist witnesses for $\langle \mathtt{init}, \mathtt{i}, \mathtt{j} \rangle$, namely $\langle \mathtt{s}, 0, \mathtt{n} \rangle$ witnessing the predicate. In other words, we can prove the following theorem in ACL2:

```
(defthm pre-has-pre-state (implies (pre s) (exists-pre-state s)))
```

Notice however, that the witness for `exists-pre-state` has to provide *some* state `init` from which `s` is reachable, not necessarily the *minimal* such state. However, we can still reason as follows. Since for `s`, `(exists-pre-state s)` holds, let $\langle \mathtt{init}, \mathtt{i}, \mathtt{j} \rangle$ be the corresponding witnesses. Then `(clock s) = (nfix (- j i))` by defintion of `clock`. To relate `i` and `j` with `s`, we prove the following theorem `run-compose` that is a fundamental property of `run` and a restatement of Lemma 3 in the ACL2 logic.

```
(defthm run-compose (equal (run p (+ m n)) (run (run p m) n)))
```

The theorem states that "running" from state `p` for $(\mathtt{m}+\mathtt{n})$ `steps` is the same as first running `m steps`, and then running `n steps` from the state so reached. The theorem can be easily proved by induction on `n`. Now consider the three different cases: (i) $\mathtt{i} < \mathtt{j}$, (ii) $\mathtt{i} = \mathtt{j}$, and (iii) $\mathtt{i} > \mathtt{j}$, for the witnesses `i` and `j` of `exists-pre-state`. For the last two cases, `s` must be `halting` by the property `halting-run` that we discussed in the previous section, and hence the proof of `clock-run-is-halting` is trivial. For the first case, however, note that `(run init j)` is equal to `(run s (- j i))` by the theorem `run-compose` and elementary arithmetic. But `(- j i)` is `(clock s)` in this case, and `(run init j)` is `halting` by definition of `exists-pre-state`, proving the theorem.

Clock Functions to Inductive Invariants

To obtain the *inductive invariant* proof from *clock functions* for partial correctness, we will define `inv` as follows:

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

Notice that the definition of `inv` is the same as the one we defined in Section 3.1 in the context of total correctness. The proof of the *inductive invariant* theorems, namely `pre-implies-inv`, `inv-persists`, and `inv-implies-good` follow the same argument as for total correctness. In particular, notice that the proofs of the first two theorems never required any properties from the *clock function* proof for the predicate `inv` we defined. For proof of `inv-implies-good`, notice that if `(halting s)` is `true`, then for the witness state `init` there exists at least one `i`, namely the witness for `n`, such that `(run init i)` is halting. Hence using theorem `clock-run-is-halting` for partial correctness, we can conclude that `(run init (clock init))` satisfies `halting`. Further, by considering the three cases, namely (i) `n` < `(clock s)`, (ii) `n` = `(clock s)`, and (iii) `n` > `(clock s)`, and the theorem `halting-run`, we can conclude as in the corresponding total correctness proof in Section 3.1, that `s = (run init (clock s))`. The proof of `inv-implies-good` now follows from the theorem `clock-run-is-good`.

4 Proof Composition

From the discussions in the previous section, it should be clear that *clock functions* and *inductive invariants* are equivalent in that from correctness theorems in one strategy, one can obtain correctness theorems in the other. It makes sense, therefore, to ask whether parts of a program can be verified using different strategy and composed to provide a correctness result for the composite whole. For example, consider a program Π with two procedures `A` and `B`. The program first "calls" procedure `A`, and then "calls" `B`. Is it now possible to verify individual procedures `A` and `B`, possibly using different strategies to get a complete proof for Π?

The thorny issue in the verification of individual procedures separately comes from the use of the predicate `halting` in our framework. Recall that the predicate `halting` says that the program terminates in a very strong sense: It posits that `(step s)` must be equal to `s`! However, when a program completes a specific procedure, it does not halt, but simply returns control to the calling procedure. Hence our verification framework as is will not be useful for verifying individual procedures of a program. Further, our proofs of equivalence in Section 3 made extensive use of the definition of `halting`. Hence we need to adjust our framework and definitions a little in order to make them work for individual procedures and compose such proofs.

In order to "fix" this problem, we first consider replacing the predicate `halting` in the two approaches by a predicate `external`. Informally, the predicate `external` indicates the return of control to the calling procedure. A little reflection will reveal, however, that replacing the predicate `halting` by an arbitrary predicate `external` in the proof obligations for the two strategies makes them very different. To understand this, consider the case for total correctness. Assume that we have a *clock function* proof of a procedure, proving the analog of `clock-run-is-halting` and `clock-run-is-good`, replacing `(halting s)` by `(external s)`. Assume also that nothing else is known about the predicate `external`. Can we define `inv` and `m` based on this knowledge, which satisfy the analogs of *inductive invariant* theorems replacing `halting` by `external`?

To understand the difficulty in proving the *inductive invariant* theorems, observe that the *clock function* theorems only talk about *an* `external` state and not the *first* `external` state. In particular, consider the program Π again, and assume that it calls procedure `A`, then `B`, and then `A` again. Consider a precondition that says that the program is poised to invoke `A`, and the predicate `external` claims that the program returns from `A`. The analog of `clock-run-is-halting` using `external` states is a statement of the following form:

```
(implies (pre s)
         (external (run s (clock s))))
```

This condition, however, can be achieved if `(clock s)` for a state `s` poised to invoke `A` for the first time, returns the number of `steps` to return from `A` for the first time, or the second, that is, the return following the invocation of `A`, followed by `B` and then followed by `A`! In fact, by simply looking at the statement of this theorem, it is impossible to conclude whether the `clock` returns the number of states to complete the procesure `s` is poised to call, or some other subsequent invocation of `A`.

On the other hand, the *inductive invariants* provide a different but related complication. Notice that the theorem `inv-persists` is at the heart of the *inductive invariant* proof. However, this theorem is too strong! It states that one needs to define `inv` so that it holds for the next state if it holds at the current state, no matter what the next state is. In particular, consider a state `s` in which the program has already returned from procedure `A`. The theorem `inv-persists` needs to guarantee that even for such a state `s`, the next state produced by `stepping` the machine must satisfy the invariant. Hence, this theorem does not provide any opportunity for separating the invariant proof into separate proofs for a number of components. The invariant defined must take into account every possible state of the system, including those that execute procedure `A` and those that execute `B`.

To rectify this situation, we next consider a slight elaboration of the framework. We will generalize our framework and proof obligations for the different strategies so that we can keep track of the "first return" from the different procedures. We describe the generalized framework and proof obligations in Section 4.1. In Section 4.2 we show that the two proof strategies are equivalent in the generalized framework as well. In Section 4.3 we use this equivalence in composing proofs of different program components.

4.1 Generalized Framework

A slight reflection of the problems we discussed in composition proofs should reveal that the problems are draconian. In particular, for *clock function* proofs of a particular procedure, we do not want to define the function `clock` so that it returns, for a state `s` poised to invoke procedure `A`, a state in which some *subsequent* calls to `A` are returned. For clock functions to be meaningful in the context of verifying a single procedure, they need to describe precisely the first return from that procedure. In our discussions, note that the predicate `external` is assumed to be some predicate that recognizes the state at which such returns are made from a component of the program that we want to verify separately. To ensure that the clock function specifies the number of `step`s to the corrsponding return, we add the following constraints `clock-is-natural` and `clock-is-minimal` for a *clock function* proof of total correctness.

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

In addition, a *clock function* proof is given by the theorems `clock-is-external` and `clock-is-post` below for total correctness. These theorems are simply restatements of the theorems `clock-run-is-halting` and `clock-run-is-good` we discussed in Section 3, in terms of the predicate `external`.

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

The corresponding theorems for partial correctness will simply add the hypothesis that some `external` state is reachable from `s`. The restatements are shown below.

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

In addition, we will assume that no `pre` state is an `external` state.

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

The theorem above is necessary for technical reasons, but is a natural restriction, as the subsequent discussions on interpretation of these theorems will show. For a specific operational model `step` and predicates `pre`, `post` and `external`, we interpret the theorems above as follows. Consider a state `s` in the system that satisfies `pre`. The predicate `(pre s)` will posit, for example, that `s` is poised to invoke some particular segment of the code, for example, some procedure `A`, and that some "initial property" holds for `s`. The predicate `(external s)` typically describes that at state `s` the system has finished executing some block of code. In most cases, it will simply be a condition on the pc, for example, describing that the `pc` points to some specific statement of the program which indicates that the control has returned from executing some procedure. The predicate `(post s)` will describe the desired postcondition when the program reaches the state prescribed by `external` on starting from a state satisfying `pre`. For example, consider a system which has a procedure that sorts natural numbers. The predicate `pre` then might indicate that the state is poised to invoke the sorting routine and some list in the `heap` component of the system contains a list of natural numbers; the predicate `external` might that indicate that the pc points to the block of code immediately following the return from the sorting procedure, and the predicate `post` that the list in the `heap` is an ordered permutation of the original list.

In this context, what do the above theorems tell us? The theorems say that starting from a state `s` satisfying the precondition, which, in particular, also means that `s` is poised to invoke some specific section of the code, the system requires exactly `(clock s)` number of `step`s to exit from the block of code, that is, reach the state `f` satisfying `external`, and the state `f` so reached also satisfies the postcondition `post`. For partial correctness, this is guaranteed only if there exists some such state `f`.

One might legitimately ask whether the framework of using `external` and specifying program correctness based on the *first* state satisfying `external` is general enough. For example, can this framework handle recursive and iterative programs? Note that for example, for recursive programs, a procedure `A` might call `A` itself, several times. The procedures `return` in order opposite to the calls, that is, the last call of `A` will return first. Hence if the predicate `external` describes that the `pc` points to a return of `A`, and `pre` says that that the state is poised to invoke `A`, then presumably we are "matching" the wrong calls!

This objection is indeed legitimate if the predicate `external` could only be a function of the pc. However, in our framework, the predicate `external` can be any arbitrary predicate of the entire state. We do not impose any restriction on the form `external` can take, as long as it is a predicate expressible in ACL2. As a result, our framework can handle recursion and iteration with considerable ease. For recursive procedures, for example, notice that the return addresses for different "recursive calls" of the same procedure are stored and maintained in a stack. The predicate `(external s)` in this case can specify that the pc points to the next statement after the call of a procedure and the "stack of recursive calls" is empty.

We have surveyed several proofs of system models, including ACL2 models of sequential programs in the JVM models described in [28, 29]. In all sequential program models we studied, the verification has been split up into several components proving individual procedures, and the theorems about individual procedures could always be described in terms of `pre`, `post`, and `external`, as described above. This framework provides a powerful and natural generalization of the program verification framework for operational semantics that we described in Sections 1.1 and 2.3.

As the discussions above will indicate, we have restated the correctness theorem to roughly specify the following: *If starting from a state satisfying* `pre` *the system reaches a state satisfying* `external` *then the first such state must also satisfy* `post`. In addition, for total correctness, one proves that starting from a state satisfying `pre` one eventually reaches an `external` state. This statement can be seen easily from the *clock function* theorems above, since `(clock s)`, by the above theorems, is specifically the number of `steps` required to reach an `external` state from a `pre` state. How do we generalize the *inductive invariant* approach to imply the above statement?

The crucial observation for this generalization is that when verifying a specific component of a program, one does not want to define `inv` that persists over every `step`, but only till the first return from the block of code being verified is encountered. Recall from our rpevious discussions in this section that the requirement for `inv` to hold on *every* next state had caused the inductive invariant theorems to be unduly strong for verification of a single component of a program by itself. This observation has been implicitly used by Moore [30] in specifying his predicate `inv` for recursive and iterative programs. The crucial question, then, is how to recast the *inductive invariants* theorems so that `inv` is required to hold from a state satisfying `pre` until the first time a state satisfying `external` is encountered.

To capture the concept of *first* external state in an *inductive invariant* framework, we reason as follows. We will specify `inv` so that it does not hold for `external` states. This is simple by adding `(not (external` s)) as a conjunct in the definition of (inv s). The informal understanding is that inv is required to hold on every state starting from a pre state, until an external state is encountered. Since inv does not hold on external states, we can immediately decide if (step s) is the first external state, by checking if both (inv s) and (external (step s)) hold. This informal understanding is captured in the following theorems which define the proof obligations for the *inductive invariant* strategy for partial correctness.

```
(defthm pre-implies-inv (implies (pre s) (inv s)))
(defthm inv-persists-for-internal
  (implies (and (inv s) (not (external (step s))))
           (inv (step s))))
(defthm inv-and-external-implies-post
  (implies (and (inv s) (external (step s)))
           (post (step s))))
(defthm inv-is-not-external (implies (inv s) (not (external s))))
```

In addition, for total correctness, one requires the measure m to guarantee that an external state is eventually reached. The corresponding termination theorems are as follows.

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

The reader acquainted with model checking might immediately recognize some similarity between the constraints we imposed and the "until" operator U typically used in specifying certain classes of temporal properties in model checking [8, 9]. In model checking, for formulas P and Q, and given a sequence of states, the formula (P U Q) is said to hold for the sequence if and only if Q holds for some state s in the sequence, and P holds for every state before s. While the analogy between the until operator and our predicate inv is obvious — inv holds for every state until external holds, making (inv U external) true of every sequence of states starting from a pre state, at least for total correctness — the analogy is almost an afterthought and ends there. Our predicate is a stronger form, since it requires inv *not* to hold in the state satisfying external, a restriction not normally made in model checking work using "until". Indeed, while the definition of inv has been influenced by the model checking literature using "until" to specify the first time some predicate holds in a sequence, the definition of inv here is simply a by-product of our endeavor to capture the notion of "first external state" rather than any detailed analysis of the model checking approaches on specification of temporal properties.

4.2 Generalized Equivalence Theorems

Even in the generalized framework we discussed, the two approaches are equivalent. Given our discussions in Section 4.1 and proofs in Section 3 this should probably be anticipated. However, these generalized equivalence theorems will finally let us compose proofs of parts of programs into a complete proof of the entire system. In this section, we discuss proofs of equivalence in the generalized framework. Since most of our proofs will closely follow the proofs we described in Section 3 we merely provide the relevant definitions here, and omit the details of the ACL2 proofs.

Consider total correctness first. To get a *clock function* proof from an *inductive invariant* proof, we will define clock as follows:

```
(defun clock-aux (s)
  (cond ((not (inv s)) 0)
        ((external (step s)) 0)
        (t (+ 1 (clock-aux (step s))))))
(defun clock (s) (+ 1 (clock-aux s)))
```

Informally, the function clock-aux, given a state s satisfying inv counts the maximum number of steps from `s` that can be executed without encountering an `external` state. The following theorems formalize this intuition.

```
(defthm inv-run-1 (implies (inv s) (inv (run s (clock-aux s)))))
(defthm inv-run-2 (implies (inv s) (external (step (run s (clock-aux s))))))
```

The *clock function* theorems now follow simply by showing that `(clock s)` takes any state `s` satisfying `inv` to the first state that satyisfies `external`.

To get an *inductive invariant* proof from *clock functions* we will define `inv` and `m` as follows:

```
(defun-sk inv (s)
    (exists (init i)
            (and (pre init) (natp i)
                 (< i (clock init))
                 (equal s (run init i)))))
(defun m (s)
  (mv-let (init i)
          (inv-witness s)
          (nfix (- (clock init) i))))
```

The predicate `(inv s)` posits that there exists a state `init` which satisfies `pre` and from which `s` is reachable without encountering an `external` state on the way. The function `m` counts the number of `step`s still remaining before such an `external` state is encountered. Recall that in total correctness proofs, we know that starting from a `pre` state , we will eventually encounter an `external` state after `stepping` for `(clock init)` times. Hence, the difference between `(clock init)` and `i`, gives the number of `step`s remaining when `s` is encountered, before an `external` state is reached.

The arguments for partial correctness are similar. To obtain a *clock function* proof from *inductive invariants* we define `clock` as follows:

```
(defun-sk for-all-inv (s i) (forall j (implies (<= j i) (inv (run s j)))))
(defun-sk exists-run-to-external (s)
  (exists i (and (natp i)
                 (for-all-inv s i)
                 (inv (run s i))
                 (external (step (run s i))))))
(defun clock (s)
    (if (exists-run-to-external s)
        (1+ (exists-run-to-external-witness s))
      0))
```

While the definition might appear convoluted on first sight, it really is a formal expression of the following statement in the ACL2 logic: *For a state* `s`*, the function* `(clock s)` *returns the number of* `step`*s required to encounter the first* `external` *state if such an* `external` *state exists; otherwise it returns* `0`*.* Recall that this function is similar to the corresponding `clock` we described in Section 3.2, recast in terms of the "first external state". To obtain the *inductive invariants* from *clock functions* in partial correctness, we define `inv` as follows:

```
(defun-sk no-external-run (s) (forall i (not (external (run s i)))))
(defun inv (s)
  (if (no-external-run s)
      T
    (and (exists-pre-state s)
         (not (external s)))))
```

The only minor "twist" in this definition is the situation when no `external` state is reachable from `s`. This is recognized by the predicate `no-external-run`. Recall from our discussions of Defchoose Principle in Section 2.1 that there are no constraints on the return value of `exists-pre-state-witness` in this case.

Hence to define `inv` so that `inv-persists-for-internal` is a theorem, we "force" `inv` to return `T` in this case.

The proofs of equivalence of the two strategies using these definitions have been successfully checked by the ACL2 theorem prover. We should note that the actual ACL2 proofs of the theorems are not trivial; in fact, considerable experience with the theorem prover is required to lead the system to these proofs. The ACL2 scripts for these proofs are available as `books` for the current version of the theorem prover, and the interested reader can obtain them by request from the first author, to get an understanding of what is involved in the proof process. However, as our discussions should make clear, the definitions of the different functions capture the basic intuitions showing that the two proof strategies are indeed equivalent, and the difficulties of the proofs are "merely" technicalities.

4.3 Composing Component Proofs

We now discuss how our framework can handle composition of proofs of individual components into a proof of the complete program. For simplicity, we only discuss *sequential composition* here, that is, we consider a program that first "calls" some specific block `A` of code, and when it exits from that block, it calls another component `B`. More complicated compositions, including branches, loops, and recursion, can be built up from sequential compositions, and hence, our framework can handle such elaborate compositions as well. Further, we only focus on *clock function* proofs in this section. As we will see, compositions are more natural for such proofs than *inductive invariants*. Using our results in Section 4.2 however, any *inductive invariant* proof of a component can be translated into a *clock function* proof, and our composition approach can then applied to the "translated proof".

Let us denote a *clock function* proof using functions `pre`, `external`, `post`, and `clock` by the tuple ⟨`pre, external, post, clock`⟩. Assume that we have two proofs ⟨`pre-A, external-A, post-A, clock-A`⟩, and ⟨`pre-B, external-B, post-B, clock-B`⟩, for two program components `A` and `B` respectively. Informally, the proofs indicate that if a state is poised to execute the block of code in one component, then after the number of `steps` specified by `clock`, it must reach a state `external` for the component, and the post condition `post` for the component must hold. To compose two such proofs, we need another condition that says that the two blocks are sequentially composed.

```
(defthm sequence (implies (and (external-A s) (post-A s)) (pre-B s)))
```

The condition `sequence` merely says that once we reach the `external` for one component, we will be poised to execute the next component in the next step. This formalizes the notion of sequential composition in this framework. In this composition, we will define the `clock` for the composition of `A` and `B` as:

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

The function `clock` simply counts the number of `steps` to reach the `external` of block `B` starting from the "beginning" of block `A`. The composition proof ⟨`pre-A, external-B, post-B, clock`⟩ now depends upon the composition of `runs` which has been established by the theorem `run-compose` in Section 3.

```
(defthm run-compose (equal (run p (+ m n)) (run (run p m) n)))
```

We note that if both the component proofs are for total correctness, the composite proof is a total correctness theorem for the composite block, otherwise it is a partial correctness theorem. This exactly matches with the intuition that a program eventually terminates if and only if every sequential block in it eventually terminates.

We also note that as compositions are performed over a large number of blocks, the clocks justifying the composition eventually get complicated. By the same token, the inductive invariants for the composite blocks are also complicated. However, experience with sequential program verification suggests that the actual structure of either the clocks or the invariants are normally of no consequence for correctness results; what matters is that functions `clock` and `inv` *can* be defined which satisfies the constraints required. There is one serious exception to that, namely if one is actually interested in proving the time complexity of the program, once a total correctness result has been achieved. Recall from Section 1.4 that the `clock` provides a precise measure of the complexity of the program in a total correctness theorem. If complexity, in addition to correctness is a concern, then clock needs to have a simplified structure that is amenable to easier reasoning in ACL2. For example, given the clock we produce for a large sequence of composite blocks, one needs another function simple-clock that is logically equivalent to clock but is more simplified in structure in order to facilitate reasoning, particularly inductve proofs of efficiency. Independent research is being performed with ACL2 to achieve simplified "clock expressions". In particular, Golden [private communication] has applied concepts from term rewriting [1] to simplify clocks for simple blocks of code. Such techniques might aid in reasoning about complexity of programs. However, for our work, the focus has been on correctness rather than complexity. In this context, the elaborate structure of clock that we produce is of no concern; clocks provide witnesses justifying that the program eventually terminates, or, in case of an individual component, eventually reaches an external state.

5 Translation between Proof Strategies

We now describe two simple macros clock-to-inv and inv-to-clock that we have implemented in the ACL2 system to translate proofs done in one strategy to the proofs in the other strategy. Macros provide ways of defining useful abbreviations in ACL2. Details of the macro facility in ACL2 are presented in the online documentation under topic defmacro and in Chapter 5 of [17]. We omit descriptions of this feature here, but just give an example. In the earlier sections, we used natp with the implicit understanding that a function natp is defined. While indeed, we used such a function in our work, we could alternatively have defined it as a macro below:

```
(defmacro natp (n)
  `(and (integerp ,n)
        (<= 0 ,n)))
```

The effect of this macro is to replace any call of natp by the body. For example (natp x) will expand to (and (integerp x) (<= 0 x)). The concept is somewhat like "inlining" used in compilers. One defines abbreviations for commonly used terms in a particular application; calls to the macro result in replacing the call by the term produced!

Macros are specially suitable for implementing theorem proving tools in ACL2. For example, Manolios and Moore [23] discuss a macro defpun to introduce partial functions in ACL2. Almost any serious researcher using ACL2 to build generic tools uses macros to abbreviate certain programming and theorem proving styles.

To understand our macro, assume that for specific "concrete" functions c-step, c-pre, c-external and c-inv, c-post, (and possibly c-m if the proof is for total correctness), the *inductive invariant* theorems have been dispatched by the theorem prover. To obtain a *clock function* proof, one simply calls our macro inv-to-clock. The macro takes 6 arguments, namely a special argument mode which takes the value :total or :partial, the names of the 6 functions above. It also requires certain keyword arguments to specify the names of the function clock and the names of auxilliary functions to use in the definition of clock. If the mode is :total, then it also requires an additional argument :measure. The macro creates a function named c-clock and generates its body using the generic functions we described in Section 4.2. For example, a specific call to the macro (clock-to-inv :total c-pre c-step c-external c-post :clock c-clock :measure c-m :run c-run :aux (c-for-all-inv c-exists-run-to-external)) expands to the following set of definitions.

```
(defun-sk c-for-all-inv (s i) (forall j (implies (<= j i) (c-inv (c-run s j)))))
(defun-sk c-exists-run-to-external (s)
  (exists i (and (c-natp i)
                 (c-for-all-inv s i)
                 (c-inv (run s i))
                 (c-external (c-step (c-run s i))))))
(defun clock (s)
    (if (c-exists-run-to-external s)
        (1+ (c-exists-run-to-external-witness s))
```

```
      0))
```

The system then attempts to prove the corresponding "clock theorems" automatically. This is done by *functional instantiation* as we described in Section 2.1. In more concrete terms, the following is one of the `defthm` event is produced by the macro.

```
(defthm c-run-is-external
  (implies (c-pre s) (c-external (c-run s (c-external s))))
  :hints ((''Goal''
            :use ((:functional-instance clock-is-external
                    (inv (lambda (s) (c-inv s)))
                    (pre (lambda (s) (c-pre s)))
                    (external (lambda (s) (c-external s)))
                    (post (lambda (s) (c-post s)))
                    (c-run (lambda (s n) (run s n)))
                    (m (mabda (s) (c-m s))))))))
```

Roughly, this `defthm` event asks ACL2 to prove the theorem `c-run-is-external`, by instantiating the theorem `clock-is-external` that we discussed in Section 4.1, instantiating the "abstract" functions in that theorem by the "concrete" functions provided. Recall that in order to successfully prove this theorem by functional instantiation, ACL2 needs to prove that the concrete functions, namely `c-pre`, `c-step`, and so on, satisfy the constraints imposed by the abstract counterparts. But the constraints imposed by the abstract counterparts are exactly the proof obligations for *inductive invariants* that have been already dispatched for the concrete functions! Hence ACL2 can automatically prove theorems for the concrete functions by functional instantiation. The macro `inv-to-clock` is similar and expands into *inductive invariant* theorems from *clock function* proofs. We also provide basic macros for composing proofs.

Our actual macros are slightly more complicated than is apparent from the description above. For example, they allow the user to provide `hints` or guidances to the theorem prover in the `defthm` events generated, in case the theorems cannot be dispatched automatically. However, the basic structure of the macros is as described.

6 Summary and Conclusion

We have discussed a formal framework for verification of sequential programs in ACL2, and shown that two widely used strategies, namely *inductive invariants* and *clock functions* are essentially equivalent in sequential program verification. In particular, from proofs using one strategy one can get a proof in the other strategy. The result holds for both partial and total correctness. We also showed how to use the two proof strategies to verify different components of a single program and obtain a correctness result for the composite whole.

As we discussed in Section 1, the two proof strategies have been in vogue in the literature and used widely for verification of operationally modeled sequential programs. However, the strategies have been considered fundamentally different; in fact, the wide use of *clock functions* in precisely characterizing the number of `steps` to terminate has often been criticized. Our results establish that in defining clocks, no more or less work is done than what a proponent of *inductive invariants* will do for verifying total correctness, and the theorems proved are logically equivalent. Furthermore, no termination argument is required for defining clocks for partial correctness theorems.

We do not advocate one proof strategy over another. In specific contexts, one strategy might seem more natural and the proof obligations easier to dispatch. However, using our theorems and translation tools one can then convert such a proof into a proof in the other strategy. Hence proofs of an individual component of a program can be done using the strategy most natural for the component alone, without concern for verification of other components.

Our techniques are applicable to operational models alone. There is another widely used program verification technique called *inductive assertions* attributed principally to Floyd [11], Hoare [15], and Dijkstra [10].

The style of specification of sequential program semantics in this approach is fundamentally different: Programs are modeled not in terms of transformation of states, but rather as transformations of predicates. Our techniques or framework cannot be directly used for reasoning about programs in this style of specification. However, Moore [30] shows that at least in the context of partial correctness, one can model programs operationally using our framework and verify correctness using *inductive invariants*, but incurring exactly the proof obligations in the *inductive assertions* method. Thus our results, in addition to this, establish that we can derive *clock function* proofs of partial correctness by incurring exactly the obligations for *inductive assertions*. However, for our method, and indeed, also the method described in [30] to work, programs need to be modeled operationally as described in our framework.

Our work also emphasizes on the power of quantification in the logic. We do not believe the equivalence theorems hold for a logic that is not expressive enough to specify first order quantified expressions. However, practical logics used for building general purpose theorem provers are all sufficiently expressive for the theorems to hold. The expressiveness of quantification in ACL2 has gone largely unnoticed. In fact, while ACL2 does provide quantification using the Defchoose Principle, there is little automatic support for quantification in the logic. Among ACL2 users, the focus has always been to define "constructive" functions and predicates. The chief reasons provided for this focus are executability, and amenability for induction. The ACL2 logic is executable; functions defined in the logic can be efficiently evaluated on concrete values. Further, a strength of ACL2 as a theorem prover is in its capability to apply well-founded inductive arguments in reasoning about recursive functions. While both these arguments are justified, experience with proofs about large systems has shown that it is often useful to abstract the details of the particular system, and reason about an abstract but generic system model of which the different concrete systems are "merely" instantiations or elaborations. Most often, quantifiers are useful for reasoning about such generic models. For example, consider a system model in some state `s`. Assume we want to discuss the property of "some state `p` from which `s` is reachable". To talk about such a state `p` it is convenient to posit, using quantification, that "some such `p` exists", and pick the `witness` for such a property as the specific `p` to reason about. We and others have found this approach convenient in diverse contexts, for example in formalization of the notion of *weakest precondition* [10] inside the operational models, and in reasoning about complex pipelined machines.

Acknowledgments

The authors greatly benefited from discussions with the ACL2 group at UT Austin. The first author specifically thanks Jeff Golden, Matt Kaufmann, Hanbing Liu, and Rob Sumners, for numerous comments, suggestions, and insights.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] A. Beckmann, S. R. Buss, and C. Pollett. Ordinal Notations and Well-orderings in Bounded Arithmetic. *Annals of Pure and Applied Logic*, pages 197–203, 2003.

[3] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

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

[5] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1975.

[6] R. S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthy and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 9–27. Springer-Verlag, 2002.

[7] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FM-CAD)*, pages 275–293. Springer-Verlag, 1996.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 8(2):244–263, April 1986.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model-Checking*. The MIT Press, Cambridge, MA, January 2000.

[10] E. W. Dijkstra. Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. *Language Hierarchies and Interfaces*, pages 111–124, 1975.

[11] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[12] G. L. Steele (Jr). *Common Lisp the Language: Second Edition*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[13] W. Goerigk. Compiler Verification Revisited. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–212. Kluwer Academic Publishers, June 2000.

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

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

[16] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[17] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[18] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. 1997.

[19] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[20] M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In *Proceedings of Third International Workshop on the ACL2 Theorem Prover and Its Applications*, pages 141–150, Grenoble, France, April 2002.

[21] H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines, and Emulators*, San Diego, CA, June 2003.

[22] P. Manolios. Correctness of Pipelined Machines. In W. A. Hunt (Jr.) and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[23] P. Manolios and J S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, To Appear.

[24] P. Manolios and D. Vroon. Algorithms for Ordinal Arithmetic. In *Nineteenth International Conference on Automated Deduction (CADE)*, LNCS, pages 243–257. Springer-Verlag, July 2003.

[25] F. J. Martin-Mateos, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz-Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. In *Proceedings of Third International Workshop on the ACL2 Theorem Prover and Its Applications*, pages 188–201, Grenoble, France, 2002.

[26] J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, volume 62, pages 21–28, Munich, West Germany, August 1962. North-Holland.

[27] J S. Moore. *Piton: A Mechanically Verified Assembly Language*. Automated reasoning Series, Kluwer Academic Publishers, 1996.

[28] J S. Moore. Proving Theorems about Java-like Byte Code. In E. R. Olderog and B. Stefen, editors, *Correct System Design — Recent Insights and Advances*, volume 1710 of *LNCS*, pages 139–162, 1999.

[29] J S. Moore. Proving Theorems About Java and the JVM with ACL2. Marktoberdorf Summer School — Lecture Notes, 2002.

[30] J S. Moore. Inductive Assertions and Operational Semantics. In D. Geist, editor, *Proceedings of 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS. Springer-Verlag, October 2003.

[31] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapoor, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[32] R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In *Second International Workshop on ACL2 Theorem Prover and Its Applications*, Austin, TX, October 2000.

[33] W. A. Hunt (Jr). *FM8501: A Verified Microprocessor*. Springer-Verlag LNAI 795, 1994.