Partial Clock Functions in ACL2

John Matthews and Daron Vroon[*]

Galois Connections, Inc.
Beaverton, OR 97005 USA
matthews@galois.com

and

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498 USA
vroon@cc.gatech.edu

November 8, 2004

Abstract

J Moore has discovered an elegant approach for verifying state invariants of imperative programs without having to write a verification condition generator (VCG) or clock function. Users need only make assertions about selected *cutpoint* instructions of a program, such as loop tests and subroutine entry and exit points. ACL2's rewriter is then used to automatically propagate these assertions through the intervening instructions.

We extend this methodology so that users can similarly prove termination properties of programs via induction over the sequence of cutpoint instructions the program executes. Just as with Moore's methodology, there is no need to specify a VCG or program-specific clock functions. These termination proofs can then be used to write efficient executable program simulators in ACL2 that don't require step-counters but are still guaranteed to terminate.

1 Introduction

In his paper *Inductive Assertions and Operational Semantics*, J Moore describes a simple and elegant method for proving partial correctness results for imperative programs [12]. What interests us most about his approach is the fact that it requires no clock function or verification condition generator. This frees the user from the headache of defining state invariants by providing assertions for every program point. Instead, the user defines assertions only on specified *cutpoints*, such as loop tests and the entry and exit points of programs. ACL2's rewriter is used to symbolically simulate the program in order to propogate these assertions through all the reachable program points. This

[*]Daron Vroon is currently affiliated with the Georgia Institute of Technology's College of Computing.

is done using tail recursive *partial functions*, which are admissable to ACL2 due to the work of Manolios and Moore who proved that every such partial function has an admissable total function as a witness [10].

In this paper, we present a variation on this method, built on the same concepts, which can easily be extended to reason about the termination of imperative programs. As with Moore's method, these termination proofs require no clock functions, VCGs, or assertions defined on program points other than select cutpoints. Thus, by our variation on Moore's approach to partial correctness, users can prove the partial correctness of their programs, and then extend this result to one of total correctness with minimal effort.

In addition, our approach to total correctness proofs can almost automatically generate efficient clockless simulators. In general, the operational semantics of an imperative programming language do not guarantee termination. Thus, in order to admit a function into ACL2 to simulate arbitrary programs written in these languages, users must include a *clock* parameter, which gives the maximum number of steps to run the program before halting. Such functions take the form

```
(defun run (k state)
   (if (zp k)
       state
     (run (1- k) (next state))))
```

However, given that we know that a program terminates, it would be nice to have a simulation function of the naive form form (i.e. without any counters):

```
(defun run (state)
   (if (halted state)
       state
     (run (next state))))
```

Given a program proven to terminate using our method, we have devised a way to automatically generate a function whose executable definition is of this form using ACL2's `mbt` macro for generating guarded executable counterparts. It is therefore an efficient clockless simulation function that runs a program to completion and is guaranteed to terminate.

We start with a detailed description of our total correctness method in Section 2. This is followed by a description in Section 3 of how to obtain efficient simulators in ACL2 by our approach. Then, in Section 4, we present a more substantial example that brings together the concepts of the previous two sections. Finally, we present related work and conclude in Sections 5 and 6.

2 The termination proof method

In this section we demonstrate the general termination proof method on the simplest state machine model we could think of that still has the potential for non-termination. However, we disable most of the model's definitions, so that the termination proof makes explicit exactly what machine model assumptions it relies on. This also paves the way for our future goal of encapsulating the proof and turning it into a generic ACL2 book.

2.1 A simple state machine model

Our example machine model, called `mstate-model`, consists of a `stobj` containing only an integer program counter field.

```
(defstobj mstate
  (progc :type integer :initially 0))
```

`mstate-model`'s next-state function simply decrements the program counter field of `mstate`.

```
(defund next (mstate)
  (declare (xargs :stobjs (mstate)))
  (update-progc (1- (progc mstate)) mstate))
```

The theory also requires us to give a predicate stating whether a machine state is a cutpoint. For `mstate-model` we arbitrarily choose our cutpoint states to be those with a non-negative program counter that is evenly divisible by 10.

```
(defund at-cutpoint (mstate)
  (declare (xargs :stobjs (mstate)))
  (and (mstatep mstate)
       (natp (progc mstate))
       (equal (mod (progc mstate) 10)
              0)))
```

We need to specify what it means for the machine to have "halted". We place quotes around the word "halted" because for most applications we don't actually expect the machine to halt once that state has been reached. Instead, we expect to be reasoning mostly about subroutines that will return to the caller and continue executing when it has finished. Therefore, we prefer to call such states *exitpoints*. The termination proof states that all cutpoint states eventually reach an exitpoint state. The resulting theorems do not state anything about what happens after an exitpoint is reached. In the `mstate` model, we specify an exitpoint to have been reached when the program counter is zero.

```
(defund at-exitpoint (mstate)
  (declare (xargs :stobjs (mstate)))
  (and (mstatep mstate)
       (equal (progc mstate)
              0)))
```

Our theory also makes a technical requirement that `nil` not be a cutpoint. This requirement should be easy to meet for non-trivial machine models.

```
(defthm nil-not-cutpoint
  (not (at-cutpoint nil)))
```

The `next-cutpoint` function, described in Section 2.2.3, will return the default value `nil` when no cutpoint is reachable.

We use the following definition of `run` to simulate the machine model a given number of steps. The expression `(at-cutpoint (run n mstate))` tests whether stepping `mstate` n times results in a cutpoint state.

```
(defun run (n mstate)
  (declare (xargs :stobjs (mstate)
                  :guard (natp n)))
  (if (zp n)
      mstate
    (let ((mstate (next mstate)))
      (run (1- n) mstate))))
```

2.2 The termination theory

Given a machine model satisfying the requirements above, the next step is to define a theory that eases the process of proving that all cutpoint states eventually lead to an exitpoint state. The main trick we will employ is to partially define a generic clock function, which we call a *partial clock function*. It is defined using the ACL2 book `defpun` [10]. The partial clock function returns the minimum number of steps the machine must take until a cutpoint state is reached. If no cutpoint state is reachable then the function returns an arbitrary value. Although its definition mentions `next` and `at-cutpoint`, the body of the partial clock function is always defined in the same way, regardless of the machine model or program we are verifying. This means we can in principle create a macro to generate the clock function automatically.

Although a partially-defined clock function does not sound very useful, it turns out we can use `run` to logically test whether the function has returned the correct value. This test allows us to determine whether a cutpoint state is reachable, and also allows us to define a (non-executable) total clock function `steps-to-cutpoint` that from a starting state returns the number of steps until the machine can reach the next cutpoint, or else returns `(omega)` if no cutpoint state is reachable. We can then use properties of ordinal arithmetic to derive stronger rewrite rules for `steps-to-cutpoint` than we can for the original partial clock function in terms of which it is defined.

2.2.1 The partial clock function

In our theory the partial clock function is called `steps-to-cutpoint-tail`. It is defined with ACL2's `defpun` macro.

```
(defpun steps-to-cutpoint-tail (n mstate)
  (if (at-cutpoint mstate)
      n
    (steps-to-cutpoint-tail (1+ n) (next mstate))))
```

`steps-to-cutpoint-tail` is defined tail-recursively, and takes an initial step-counter parameter `n`. It returns the minimum number of steps to the next cutpoint minus `n`, when a cutpoint state is reachable. Otherwise the function is unspecified.

Using ACL2's `defchoose` construct, a specification of a tail-recursive function can always be completed to a non-executable total function definition. This is precisely what the `defpun` macro does, generating the theorem `steps-to- cutpoint-tail-def`, which states:

```
(equal (steps-to-cutpoint-tail n mstate)
       (if (at-cutpoint mstate)
           n
         (steps-to-cutpoint-tail (1+ n) (next mstate)))).
```

The steps-to-cutpoint-tail function satisfies several key invariant properties. They are proved simultaneously with the theorem

```
(defthmd steps-to-cutpoint-tail-inv
  (implies (and (at-cutpoint (run k mstate))
                (integerp steps))
           (let* ((result (steps-to-cutpoint-tail steps mstate))
                  (cutpoint-steps (- result steps)))
             (and (integerp result)
                  (natp cutpoint-steps)
                  (implies (natp k)
                           (<= cutpoint-steps k))
                  (at-cutpoint (run cutpoint-steps mstate))))))
```

Together these properties state that if a cutpoint state is reachable in a finite number of steps from mstate, then

- steps-to-cutpoint-tail returns an integer value.
- The value steps-to-cutpoint-tail returns is always greater than or equal to its initial step-counter parameter steps.
- Given any cutpoint state (not necessarily the first one) reachable in k steps, where $k \geq 0$, then the value returned by steps-to-cutpoint-tail minus steps is less than or equal to k.

  In other words, the function has found the number of steps needed to get to the next cutpoint state.
- The state is actually a cutpoint state.

An important corollary states that when a cutpoint state is reachable, then the initial step-counter parameter can be moved outside of the partial step function.

```
(defthm steps-to-cutpoint-tail-diff
  (implies (and (at-cutpoint (run k mstate))
                (syntaxp (not (equal n ''0)))
                (integerp n))
           (equal (steps-to-cutpoint-tail n mstate)
                  (+ n (steps-to-cutpoint-tail 0 mstate)))))
```

2.2.2 Total clock functions

We have extended the partial clock function steps-to-cutpoint-tail into a total clock function called steps-to-cutpoint. We do this by calling the partial clock function with an initial step-counter parameter of zero, and then testing whether the function was able to reach a cutpoint state. If so, then steps-to-cutpoint returns the number of steps to that cutpoint, otherwise it returns (omega), the first infinite ordinal, indicating that a cutpoint can not be reached in a finite number of steps.

```
(defun steps-to-cutpoint (mstate)
  (declare (xargs :non-executable t))
  (let ((steps (steps-to-cutpoint-tail 0 mstate)))
    (if (at-cutpoint (run steps mstate))
        steps
      (omega))))
```

For Turing-complete machine models this function is not computable, although it is still a well-defined total function in ACL2's logic. However, `steps-to-cutpoint` is still a useful function. Logically the partial clock function `steps-to-cutpoint-tail` always returns some value. If the value is a natural number, then `run` will step the machine state that number of times. Otherwise `run` will just return `mstate` itself. In either case we know that the state returned by `run` is reachable from `mstate`.

Furthermore, we know that if that state is a cutpoint state, then a cutpoint state is reachable from `mstate`. So from the theorem `steps-to-cutpoint-tail-inv` we get that `steps-to-cutpoint-tail` returns the correct value in this case.

On the other hand, if no cutpoint state is reachable then `run` will return a non-cutpoint state. Thus the formula `(at-cutpoint (run steps mstate)))` in the definition of `steps-to-cutpoint` faithfully tests whether a cutpoint state is reachable from the input state.

Although the function `steps-to-cutpoint` uncomputable in general, it can be evaluated on well-chosen concrete machine models. In fact, for many machine programs it can be automatically simplified by the following rewrite rules.

```
(defthm steps-to-cutpoint-zero
  (implies (at-cutpoint mstate)
           (equal (steps-to-cutpoint mstate) 0)))

(defthm steps-to-cutpoint-nonzero-intro
  (implies (not (at-cutpoint mstate))
           (equal (steps-to-cutpoint mstate)
                  (o+ 1 (steps-to-cutpoint (next mstate))))))
```

These rewrite rules can be used to turn ACL2's rewriter into a symbolic simulator. In any subgoal containing an expression of the form `(steps-to-cutpoint` *term*`)`, where *term* is a sub-expression representing a machine state, ACL2 will iteratively apply `steps-to-cutpoint-nonzero- intro` as long as it can discharge the hypothesis of the rule. There are three possible outcomes of this symbolic simulation process:

- There is some expanded expression `(at-cutpoint (next (`$\cdots$ `(next` *term*`)` $\cdots$`)))` containing zero or more occurrences of `next` that ACL2 can simplify to true. In this case the rule `steps-to-cutpoint-zero` will eventually fire. The end result is that ACL2 will deduce that the original expression `(steps-to-cutpoint` *term*`)` is equal to the expanded expression `(o+ 1 (`$\cdots$ `(o+ 1 0)` $\cdots$`))`, which will be simplified to a constant number.

- During the symbolic simulation process there is some expanded machine state *term′* such that ACL2 can't simplify `(at-cutpoint` *term′*`)` to either true or false. In this case the original `(steps-to-cutpoint` *term*`)` expression will end up being simplified to
`(o+` *k* `(steps-to-cutpoint` *term′*`))`, for some natural number *k*. It means that the symbolic

simulation process is not powerful enough for the subgoal this expression occurs in. The user needs to strengthen the rules associated with their machine model so that ACL2 can decide whether *term'* is a cutpoint state or not.

- ACL2 can simplify every sequence of terms `(at-cutpoint (next (··· (next term) ···)))` containing zero or more occurrences of `next` to false. This means that *term* can not reach a cutpoint state. However, ACL2 can not detect this and instead continues rewriting until it is interrupted or runs out of memory. In this case the user must add a new cutpoint state along the path of the symbolic simulation to break the cycle.

The main advantages of `steps-to-cutpoint`'s rewrite rules are that their proofs aren't specific to the underlying machine model, and that they are valid regardless of whether a cutpoint state is reachable or not. Thus they can be used to automatically calculate the number of steps needed until the next cutpoint state is reached (if there is such a state). This should lead to more automated safety and termination proofs about the machine program.

2.2.3 Computing reachable cutpoint states

We can now use the `run` and `steps-to-cutpoint` functions to define a function that steps the machine to the next cutpoint state, provided it exists.

The function `next-cutpoint` returns the next cutpoint state reachable from a given starting state, if there is one. However, if a cutpoint state is not reachable then `next-cutpoint` returns the default value `nil`, which we require to be a non-cutpoint state.

```
(defun next-cutpoint (mstate)
  (declare (xargs :non-executable t))
  (let ((steps (steps-to-cutpoint mstate)))
    (if (natp steps)
        (run steps mstate)
      nil)))

(defthm nil-not-cutpoint
  (not (at-cutpoint nil)))
```

This definition of `next-cutpoint` leads to the two simple symbolic simulation rules below. In particular, returning a default value when a cutpoint state is unreachable allows a simpler hypothesis for the second rewrite rule.

```
(defthm next-cutpoint-at-cutpoint
  (implies (at-cutpoint mstate)
           (equal (next-cutpoint mstate)
                  mstate)))

(defthmd next-cutpoint-intro-next
  (implies (not (at-cutpoint mstate))
           (equal (next-cutpoint mstate)
                  (next-cutpoint (next mstate)))))
```

Finally, because we forbid the default value nil from being a cutpoint state, we know that if the value returned by next-cutpoint is a cutpoint, then that state is reachable in a finite number of steps from the starting state:

```
(defthm next-cutpoint-reaches-cutpoint
  (iff (at-cutpoint (next-cutpoint mstate))
       (natp (steps-to-cutpoint mstate))))
```

2.3 Reasoning about cutpoints

Our main goal is to prove that an exitpoint state is eventually reached from any cutpoint state. We achieve this by providing a measure function cutpoint-measure on the cutpoint states, and then prove by ordinal induction on this measure that an exitpoint is eventually reached.

For mstate-model the cutpoint measure function just observes the value of the current program counter. We require that the measure function always returns a valid ACL2 ordinal.

```
(defun cutpoint-measure (mstate)
  (declare (xargs :stobjs (mstate)))
  (nfix (progc mstate)))

(defthm cutpoint-measure-is-ordinal
  (o-p (cutpoint-measure mstate)))
```

Next, we define a function cutpoint-to-cutpoint that atomically transitions from one cutpoint state to the next one, if it exists.

```
(defun cutpoint-to-cutpoint (mstate)
  (declare (xargs :non-executable t))
  (next-cutpoint (next mstate)))
```

By expanding the definition of cutpoint-to-cutpoint, ACL2 can prove by symbolic simulation that if a cutpoint state is not an exitpoint then another cutpoint state can be reached, and that the measure of that next cutpoint has decreased. The first of these three theorems below also demonstrates how partial correctness results can be proved by symbolic simulation with (extended) partial clock functions.

```
(defthm steps-to-next-cutpoint-natp
  (implies (and (at-cutpoint mstate)
                (not (at-exitpoint mstate)))
           (natp (steps-to-cutpoint (next mstate)))))

(defthm cutpoint-to-cutpoint-returns-cutpoint-state
  (implies (natp (steps-to-cutpoint (next mstate)))
           (at-cutpoint (cutpoint-to-cutpoint mstate))))

(defthm cutpoint-measure-decreases
  (implies (and (at-cutpoint mstate)
                (not (at-exitpoint mstate)))
           (o< (cutpoint-measure (cutpoint-to-cutpoint mstate))
               (cutpoint-measure mstate))))
```

A few basic lemmas about modular arithmetic are needed in `mstate-model` for the symbolic simulation to succeed, since `at-cutpoint` and `next` are defined in terms of the `mod` operator and subtraction, respectively.

The fact that the cutpoint measure decreases allows us to define a total function that from any cutpoint state returns the first reachable exitpoint state.

```
(defun next-exitpoint (mstate)
  (declare (xargs :non-executable t
                  :measure (cutpoint-measure mstate)))
  (cond ((not (at-cutpoint mstate)) mstate)
        ((at-exitpoint mstate) mstate)
        (t (next-exitpoint (cutpoint-to-cutpoint mstate)))))
```

We can finally prove that `next-exitpoint` behaves correctly, and that cutpoint states eventually lead to exitpoint states.

```
(defun steps-to-exitpoint (mstate)
  (declare (xargs :non-executable t
                  :measure (cutpoint-measure mstate)))
  (cond ((not (at-cutpoint mstate)) 0)
        ((at-exitpoint mstate) 0)
        (t (+ 1 (steps-to-cutpoint (next mstate))
              (steps-to-exitpoint (cutpoint-to-cutpoint mstate))))))

(defthmd next-exitpoint-correct
  (implies (at-cutpoint mstate)
           (equal (run (steps-to-exitpoint mstate) mstate)
                  (next-exitpoint mstate))))

(defthm at-cutpoint-implies-reaches-exitpoint
  (implies (at-cutpoint mstate)
           (at-exitpoint (next-exitpoint mstate))))
```

These theorems are proved by ordinal induction on `cutpoint-measure`, derived by ACL2 from the definitions of `steps-to-exitpoint` and `next-exitpoint`, respectively.

3 Efficient simulators and ACL2 limitations

We would like to take advantage of our termination proof method to build efficient terminating machine simulators that do not require step-counter parameters. As a first step, consider the following `stobj`-compliant version of `next-cutpoint` (where `dummy-mstate` creates some valid mstate that isn't a cutpoint):

```
(defun next-cutpoint-exec (mstate)
  (declare (xargs :stobjs (mstate)
                  :measure (steps-to-cutpoint mstate)
                  :guard (and (mstatep mstate)
                              (natp (steps-to-cutpoint mstate)))))
  (if (mbt (and (mstatep mstate)
                (natp (steps-to-cutpoint mstate))))
      (if (at-cutpoint mstate)
          mstate
        (let ((mstate (next mstate)))
          (next-cutpoint-exec mstate)))
    (dummy-mstate mstate))).
```

The `mbt` macro utilized here stands for "must be true" and is used to introduce a test that is not to be evaluated. Logically, the test is necessary to prove termination. However, in practice the guard check assures that the body of the `mbt` is true, so it doesn't need to be evaluated when executing the function. Thus, the executable version of `next-cutpoint-exec` is just the "then" branch of the outer `if` statement. The `dummy-mstate` function makes `mstate` into a valid `mstate` that is not an exitpoint. This is necessary since the rules of `stobj` use require that any function that alters the `stobj` returns it. Given the theory presented in Section 2, ACL2 is able to prove the following properties about the guard:

```
(defthm mstatep-next
  (implies (mstatep mstate)
           (mstatep (next mstate))))

(defthm natp-steps-to-cutpoint-next
  (implies (and (mstatep mstate)
                (not (at-cutpoint mstate))
                (natp (steps-to-cutpoint mstate)))
           (natp (steps-to-cutpoint (next mstate)))))
```

Together these properties imply that the guard conjectures for `next-cutpoint-exec` are satisfied. However, our guard is not executable, since it calls the non-executable function, `steps-to-cutpoint`. ACL2 version 2.8 requires that all guards of executable functions be executable, so that the guard can be checked when the user is invoking the function at the interactive prompt. This means that we can't verify the guards for `next-cutpoint-exec`. This is unfortunate because it prevents the function from being compiled in contexts where the guard is statically known to hold, such as in this efficient version of `cutpoint-to-cutpoint`:

```
(defun cutpoint-to-cutpoint-exec (mstate)
  (declare (xargs :stobjs (mstate)
                  :guard (and (at-cutpoint mstate)
                              (not (at-exitpoint mstate)))))
  (let ((mstate (next mstate)))
    (next-cutpoint-exec mstate)))
```

In this case the guard for `cutpoint-to-cutpoint-exec` is executable, and moreover it implies the guard for the call to `next-cutpoint-exec` holds by `natp-steps-to-cutpoint-next` above and `steps-to-next-cutpoint-natp` (Section 2.3). Thus we see that invoking the executable definition of `next-cutpoint-exec` in this context should not cause any logical inconsistency or non-termination problems.

3.1 The elegant solution: a modest proposal

ACL2 currently allows the definition of an executable function whose body contains a call to a non-executable function. The result of running such a function is that it runs normally until it reaches the non-executable function call, at which time it throws an error. If the execution never reaches this call, the function terminates normally.

We propose that ACL2 take a similar policy with regards to function guards. As we pointed out above, a function's guard is not evaluated in the case where the function is called from another function whose guards have been verified. In this case, the guard is proven to hold when the function is called, and therefore known to hold statically. In the case where ACL2 attempts to evaluate a non-executable guard, an error can be thrown. We feel that this policy would be more consistant with the already existing policy of allowing non-executable function calls within executable functions.

A more aggressive approach would be to call the simplifier on non-executable guards at the prompt, and if they simplify to true then to invoke the function's executable counterpart. Even more daring would be to try to simplify non-executable guards during subgoal proofs (???). However, care must be taken in this case not to rely on subgoal assumptions, since these assumptions may not hold in the ACL2 runtime environment.

3.2 A workaround

Our proposed change to ACL2 would allow us to define our efficient clockless simulator without any additional effort. However, we have devised a way to work around the limitation in ACL2's guard policy. It involves using ACL2's more lenient policy of allowing executable functions to contain calls to non-executable functions in order to define an executable version of `steps-to-cutpoint`. The main difficulty here involves two of ACL2's necessarily strict rules for using `stobj`s. The first says that a `stobj`-compliant function cannot pass a `stobj` to a non-`stobj`-compliant function. Thus, we cannot pass `mstate` to `steps-to-cutpoint-tail`. The second is that any `stobj`-compliant function that alters a `stobj` must return that `stobj`. Our `steps-to-cutpoint` function calls `run`, which alters the `mstate`, but we want to return the number of steps to the next cutpoint, not the `mstate`.

In order to get around these problems, we created a way to copy data from a `stobj` to a normal object with the same logical structure as the `stobj`, and vice versa. The result is a command we call `defstobj+`. This command has the same general form as a `defstobj` command. However, in addition to creating a `stobj` with all the normal functionality, it provides functions for copying to and from the `stobj` as well as proofs that these functions are logically identity functions. For our `mstate` example, we alter the definition of `mstate` to use `defstobj+` instead of `defstobj`:

```
(defstobj+ mstate
  (progc :type integer :initially 0))
```

In addition to the normal functionality, this command provides the following functions:

```
(defun logical-mstatep (x)
  (declare (xargs :guard t))
  (and (true-listp x)
       (equal (len x) 1)
       (progcp (nth *progc* x))))
```

```
(defun copy-to-mstate (copy mstate)
  (declare (xargs :stobjs (mstate)
                  :guard (logical-mstatep copy)))
  (let* ((mstate (update-progc (nth *progc* copy)
                               mstate)))
        mstate))

(defun copy-from-mstate (mstate)
                    (declare (xargs :stobjs (mstate)))
                    (list (progc mstate)))
```

as well as the following theorems:

```
(defthm logical-mstatep-mstatep
  (equal (logical-mstatep x) (mstatep x)))

(defthm copy-to-mstate-noop
  (implies (and (mstatep x) (mstatep y))
           (equal (copy-to-mstate x y) x)))

(defthm copy-from-mstate-noop
  (implies (mstatep mstate)
           (equal (copy-from-mstate mstate)
                  mstate)))
```

These definitions and theorems are `stobj`-specific, and work for any `stobj` structure (even in the presence of array fields). In addition, the `defstobj+` book, in which the command is defined, contains a command called `with-copy-of-stobj`. This macro has the same general form as the `with-local-stobj`. It creates a local `stobj` that is a copy of the global one and performs all the actions specified within the body on that local copy, and not the global one.

With these two features combined, we can create an executable version of our `steps-to-cutpoint` function:

```
(defun steps-to-cutpoint-exec (mstate)
  (declare (xargs :stobjs (mstate)))
  (let ((steps (steps-to-cutpoint-tail 0 (copy-from-mstate mstate))))
    (if (and (natp steps)        ;the number of steps is a natural number.
             (with-copy-of-stobj ;running a copy of mstate forward steps steps
              mstate             ;gives us a cutpoint.
              (mv-let (result mstate)
                      (let ((mstate (run steps mstate)))
                        (mv (at-cutpoint mstate) mstate))
                      result)))
        steps
      (omega))))
```

By calling `steps-to-cutpoint-tail` on a non-`stobj` copy of `mstate`, we no longer pass our `stobj` to a non-`stobj`-compliant function. To bypass the problem caused by altering mstate without returning it, we use the `with-copy-of-stobj` macro. We run a duplicate of mstate forward to be sure the cutpoint is actually reachable. The parameter mstate is untouched through this whole process. Therefore, we can legally return the value of `steps` or `(omega)` without returning the mstate. So now we have an executable version of our `steps-to-cutpoint` function, as we can prove in ACL2:

```
(defthm steps-to-cutpoint-exec-steps-to-cutpoint
 (implies (mstatep mstate)
          (equal (steps-to-cutpoint-exec mstate)
                 (steps-to-cutpoint mstate))))
```

Using `steps-to-cutpoint-exec`, we can verify the guards of both `next-cutpoint-exec` (once we replace `steps-to-cutpoint` with `steps-to-cutpoint-exec`) and `cutpoint-to-cutpoint-exec`. Finally, we can use these functions to create our clockless simulator:

```
(defun fast-cutpoint-to-cutpoint (mstate)
  (declare (xargs :stobjs (mstate)
                  :measure (cutpoint-measure mstate)
                  :guard (at-cutpoint mstate)))
  (if (mbt (at-cutpoint mstate))
      (if (at-exitpoint mstate)
          mstate
        (let ((mstate (cutpoint-to-cutpoint-exec mstate)))
          (fast-cutpoint-to-cutpoint mstate)))
    (dummy-mstate mstate)))
```

4 Putting it all together: Fibbonacci sequence on the TINY Machine

We have presented our method for proving termination using clockless simulators, as well as a method for extending that termination proof to create efficient clockless simulators. In this section we provide a more realistic example to demonstrate how it all fits together.

The semantics of this example are provided by the TINY model, a small, stack-based machine first presented in [6] as a high-speed simulator example using `stobjs`. The Fibonacci sequence is the sequence whose first two elements are 1, and every subsequent element of which is the sum of the previous two elements: $(1, 1, 2, 3, 5, 8, 13, \ldots)$. Our `fib` function takes a positive integer, `n`, and returns the `nth` value in the Fibonacci sequence. The specification for this function, written in ACL2, is the following:

```
(defun fib-spec (n)
  (cond ((not (integerp n)) 0)
        ((< n 0) 1)
        ((equal n 0) 1)
        ((equal n 1) 1)
        (t (logext *word-size* (+ (fib-spec (- n 1)) (fib-spec (- n 2)))))))
```

Figure 1 TINY assembly code for fib program

```
(pushsi 1    ;100 start-prog-address
 dup         ;102
 dup         ;103
 pop 20      ;104                    fib0 = 1;
 pop 21      ;106                    fib1 = 1;
 sub         ;108
 dup         ;109 loop-label
 jumpz 127   ;110                    if n = 0, goto done-label;
 pushs 20    ;112
 dup         ;114
 pushs 21    ;115
 add         ;117
 pop 20      ;118                    fib0 = fib1;
 pop 21      ;120                    fib1 = fib0 (old value) + fib1;
 pushsi 1    ;122
 sub         ;124                    n = n-1;
 jump 109    ;125                    goto loop-label;
 pushs 20    ;127 done-label
 add         ;129                    return fib0 + n;
 halt)       ;130 halt-prog-address
```

where `(logext n x)` returns the integer corresponding to the low `n` bits of `x` interpreted as a signed integer, and `*word-size*` is the number of bits in a word in TINY, which is 32. The `fib` program written in the TINY assembly language is

The program addresses are given immediately to the right of the instructions. Note that arguments take up 1 address space each, so that every address in the program is not necessarily an instruction. To the right of the addresses are the cutpoint labels. Finally, right of those are some comments to help clarify the code. Basically, the two most recently computed values of the Fibonacci sequence are stored in addresses 20 and 21. Each iteration of the loop puts the sum of the values in these addresses in 21, and moves the old value of 21 to 20. The counter (`n`) is maintained on the stack. It is assumed that this counter is on the top of the stack at the beginning of the program. Note that at each cutpoint, the counter is the only thing on the stack.

The basic functions for reasoning about arbitrary cutpoints in the TINY model are in Figure 2. The `at-cutpoint` function captures several important invariants of our program. First, it checks if our program counter (`progc`) is one of the cutpoint addresses. Second, it verifies that the fib program is loaded into memory at the appropriate location. Third, it makes sure that `tiny-state` is indeed a `tiny-state` `stobj`. Next, it checks that there is only one item on the stack (which is our loop counter). Finally, it verifies that the loop counter has the right value (`dtos-val` gets the value off the top of the stack).

Our `dummy-state` function puts all the default field values into `tiny-state`. It does this by creating a fresh local `stobj` with `with-local-stobj`, which it copies into the global `stobj` using the `copy-to-tiny-state` and `copy-from-tiny-state` functions created by the `defstobj+` construct. This is equivalent to setting `tiny-state` equal to `(create-tiny-state)`.

Figure 2 Basic cutpoint functions for fib on TINY

```
(defconst *fib-cutpoints*
          (list *prog-start-address* *loop-label* *done-label* *prog-halt-address*))

(defun at-cutpoint (tiny-state)
  (declare (xargs :stobjs (tiny-state)))
  (and  (member (progc tiny-state) *fib-cutpoints*)
        (program-loaded tiny-state *fib-prog* *prog-start-address*)
        (tiny-statep tiny-state)
        (equal (dtos tiny-state) *init-dtos*)
        (cond ((equal (progc tiny-state) *prog-start-address*)
               (< 0 (dtos-val tiny-state 0)))
              ((equal (progc tiny-state) *loop-label*)
               (<= 0 (dtos-val tiny-state 0)))
              ((equal (progc tiny-state) *done-label*)
               (= 0 (dtos-val tiny-state 0)))
              (t t))))

(defun dummy-state (tiny-state)
  (declare (xargs :stobjs (tiny-state)))
  (let ((ts (with-local-stobj
             tiny-state
             (mv-let (result tiny-state)
                     (mv (copy-from-tiny-state tiny-state) tiny-state)
                     result))))
    (copy-to-tiny-state ts tiny-state)))

(defun at-exitpoint (tiny-state)
  (declare (xargs :stobjs (tiny-state)))
  (and (equal (progc tiny-state) *prog-halt-address*)
       (program-loaded tiny-state *fib-prog* *prog-start-address*)
       (tiny-statep tiny-state)
       (equal (dtos tiny-state) *init-dtos*)))

(defconst *max-prog-address* (1- (+ *prog-start-address*
                                    (len *fib-prog*))))

(defun cutpoint-measure (tiny-state)
  (declare (xargs :non-executable t))
  (if (at-exitpoint tiny-state)
      0
    (o+ (o* (omega) (nfix (dtos-val tiny-state 0)))
        (nfix (- *max-prog-address* (progc tiny-state))))))
```

The `at-exitpoint` function is our predicate for recognizing exit states in our program. For the fib example, this function verifies that the `tiny-state` is at the halt address, that the fib program is loaded in the proper location in mamory, that `tiny-state` is in fact a `tiny-state`, and that there is only one value on the stack.

The `cutpoint-measure` function give us the measure function that will allow us to prove termination. Since there is one loop with a natural number counter that decreases until it reaches 0, our measure is ω multiplied by the counter (which is the value at the top of the stack at each cutpoint) added to the value of the program counter.

These are the only program-specific functions necessary for our method. Beyond the rewrite rules necessary for reasoning about the TINY machine, the rest is virtually identical to the mstate example. The one difference is that we don't need to bother with the non-executable versions of `steps-to-cutpoint` and `next-cutpoint` (see our supporting material for details). We do this because ACL2 can just as easily reason about the executable versions of these functions, since our `defstobj+` construct proves that the `copy-to-tiny-state` and `copy-from-tiny-state` functions are logically just identity functions. Thus for the relatively small effort required to use `defstobj+` and to write out the executable version of `steps-to-cutpoint`, we get an efficient, clockless, executable function for running our fib program.

5 Related work

There are strong parallels between the method we have presented here to prove termination and J Moore's work combining inductive assertions with operational semantics in order to prove the partial correctness of imperative programs [12]. Like our method, Moore's work limits reasoning to cutpoints, as users need only specify assertions for the cutpoints of the program. A partial function that steps the machine to the next cutpoint (if such a cutpoint exists) is then used to push these assertions through the program points between the cutpoints. The partial function invariant from Moore's work applied to our TINY example would take the form

```
(defpun invariant (tiny-state)
   (if (at-cutpoint tiny-state cutpoints)
       (assert state)
     (invariant (next tiny-state))))
```

and the invariant correctness theorem would take the form

```
(implies (invariant tiny-state)
         (invariant (next tiny-state)))
```

This is where our work differs from Moore's. Instead of defining the invariant as a partial function, we define the partial clock function. This serves two purposes. First, it allows us to more easily extend the method for proving partial correctness to apply to termination proofs. The partial clock function gives us the measure necessary to run from cutpoint to cutpoint. The second purpose of the partial clock function is that it more thoroughly pushes reasoning about the program up to the cutpoint level rather than the single step level. For example, rather than defining assertions on cutpoints and extending them to invariants on all states, we simply state the invariant as an invariant over the cutpoints and use symbolic simulation to prove correctness at the cutpoint level by a theorem of the form

```
(implies (invariant tiny-state)
         (invariant (cutpoint-to-cutpoint tiny-state)))
```

This helps us verify the guards of the simulation function which runs from cutpoint to cutpoint rather than state to state. However, the result is still equivalent to that of Moore's work. That is, we still get an invariant that implies partial correctness for the program.

Both Moore's work and ours are dependent on the existence of partial functions in ACL2, added by Manolios and Moore [10].

Our work is also closely related to Ray and Moore's work on the formal correspondence between the inductive invariants method and the clock functions method of proving partial and total correctness results for state machines [14].

Specifically, Ray and Moore show that given valid theorems of total (respectively, partial) correctness in terms of either method, then the required definitions and theorems to prove total (partial) correctness using the other method can be generated automatically. Thus both methods have the same logical strength.

Ray and Moore go on to generalize both methods so that they can be used compositionally. This allows, for example, total (partial) correctness results about individually verified software subroutines to be combined into a total (partial) correctness result for a client program that calls the subroutines. Formal correspondences are also proved between the generalized methods.

In Ray and Moore's approach, different versions of the clock function are defined, depending on whether partial correctness or total correctness is being proved. For partial correctness, their clock function is defined in terms of Skolem functions, using the `defun-sk` macro. In our own work, clock functions are uniformly defined as tail-recursive partial functions with the `defpun` package. However, this alone is not sufficient to distinguish their clock function definitions from ours, since the core theorems produced by both `defpun` and `defun-sk` rely on the same underlying `defchoose` facility of ACL2.

However, one clear difference with our form of clock function is that it has been explicitly designed to satisfy ordinal arithmetic properties that are not conditioned on whether a cutpoint or exitpoint state is reachable. This allows the same clock definition to be used for proving both partial and total correctness properties. We believe, but haven't proved, that Ray and Moore's correspondence proofs could be adapted to use our form of clock function.

A second difference is that defining the clock function as a tail-recursive partial function gives us a way to evaluate the clock function on concrete machine states in ACL2, and to symbolically simulate them on symbolic state expressions. Since the clock function is partial it may not terminate, but if it does terminate then it returns the correct value. In contrast, there is no method for evaluating or symbolically simulating functions defined using `defchoose` or `defun-sk`, in general. The ability to symbolically simulate clock functions up to the next cutpoint or exitpoint significantly increases proof automation, and paves the way for building efficient clockless simulators.

Termination in ACL2 was a topic in last year's ACL2 workshop with Manolios and Vroon's paper implementing a new ordinal notation and ordinal arithmetic library which are now the foundation of termination reasoning in ACL2 [11]

Most theorem provers for higher order logics provide some level of support for admitting well-founded (i.e. terminating) function definitions. Classical higher order logic is strong enough for these functions to be admitted definitionally in terms of a higher-order `defchoose`-like function called the Hilbert choice operator. Slind has developed a theory and portable library of theorem proving tactics that helps automate these proofs. Given a set of pattern-matching recursion equations over an inductive datatypes and a well-founded relation, the library attempts to prove that all recursive calls in the pattern matching equations are applied to smaller values according to the well-founded relation. If succcessful, the library generates the pattern matching equations as theorems, as well as a function-specific induction scheme [15, 17].

These techniques can be used to model imperative programs in the same way that ACL2 does, as state-passing functions. However, many imperative algorithms call themselves recursively multiple times in succcession. A simple example is a function that destructively zeroes out every leaf node of a binary tree. In this case the returned state value of the first recursive call is used as the state parameter to the second recursive call. These *nested recursive* function definitions require more powerful termination proof techniques [16]. Krstic̀ and Matthews explore using *inductive invariants* to tackle these proofs in the context of verifying imperative Binary Decision Diagram algorithms [8, 9].

Researchers have studied for decades appropriate ways to structure partial- and total-correctness proofs for higher-level imperative programming languages. A recent text by de Roever et al [5] describes some of these techniques. It also introduces a general framework based on inductive assertions that can be directly adapted to cutpoint-based reasoning.

There have been several promising methods for automatically proving termination of imperative programming languages. For example, Podelski and Rybalchenko have given a complete method for proving termination for non-nested loops with linear ranking functions [13]. Dams, Gerth, and Grumberg have given a heuristic for automatically generating ranking functions [4]. Finally, Colón and Sipma have developed two algorithms for proving termination. One synthesizes linear ranking functions, but is limited to programs of few variables. The other is more heuristic in nature, but tends to converge faster to the invariants which it can discover [2, 1]. However, none of these develop general methods for reasoning about termination. They instead focus on decidable subsets of the termination problem by using decision procedures to develop linear ranking functions.

6 Conclusions

We have presented a variation of Moore's method for proving the partial correctness of programs using partial functions and symbolic simulation that provides an easy technique for verifying the total correctness of imperative programs. We presented a way to use this result to define efficient terminating program simulators in a perfect world, and described those features needed in ACL2 to make this a practical reality.

We intend to apply these techniques in a certifying compiler we are building at Galois Connnections, Inc. for the Cryptol™ domain-specific executable specification language [3]. Cryptol allows encryption algorithms to be specified declaratively and at a higher level of abstraction than can be done in conventional imperative programming languages, while still allowing efficient code to be generated. Our certifying Cryptol compiler will target the instruction set of the AAMP7 secure microprocessor, being developed at Rockwell Collins. In addition to object code, the compiler will emit a *correspondence proof* ACL2 script that automatically verifies that the generated code faithfully implements the original program's Cryptol semantics.

Acknowledgements

We would like to thank the ACL2 experts at Rockwell Collins, including Dave Greve, Dave Hardin, and Matt Wilding, for stimulating discussions and useful ACL2 advice, especially on the issues that arise when using ACL2 to reason about real-world machine models. We would like to thank Matt

Kaufmann and Bill Young for their ACL2 expertise as well. We also appreciate the advice and help Matt Kaufmann, Pete Manolios, and J Moore gave us in using the `defpun` book. Jeff Lewis and Mark Shields have been of great help as a sounding board for applying our ideas to the certifying compiler.

References

[1] M. A. Colón and H. B. Sipma. Synthesis of linear ranking functions. In *TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81, 2001.

[2] M. A. Colón and H. B. Sipma. Practical methods for proving program termination. In *International Conference on Computer Aided Verification, CAV'02*, volume 2404 of *LNCS*, pages 442–454, 2002.

[3] Information on Cryptol can be found at `http://www.cryptol.net`.

[4] D. Dams, R. Gerth, and O. Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advanced Verification*, July 2000. See URL `http://www.cs.utah.edu/wave/`.

[5] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, Nov. 2001.

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

[7] M. Kaufmann and J. S. Moore, editors. *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003. See URL `http://www.cs.utexas.edu/users/moore/-acl2/workshop-2003/`.

[8] S. Krstiċ and J. Matthews. Verifying BDD algorithms through monadic interpretation. 2294:182–195, 2002.

[9] S. Krstiċ and J. Matthews. Inductive invariants for nested recursion. In *16th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 2758 of *LNCS*, pages 253–259. Springer-Verlag, 2003.

[10] P. Manolios and J. S. Moore. Partial functions in ACL2. Technical report, Computer Sciences, University of Texas at Austin, 2001. See URL `http://www.cs.utexas.edu/users/moore/publications/defpun/%-index.html`.

[11] P. Manolios and D. Vroon. Ordinal arithmetic in ACL2. In Kaufmann and Moore [7]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.

[12] J. S. Moore. Inductive assertions and operational semantics. In D. Geist, editor, *The 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods – CHARME 2003*, volume 2860 of *LNCS*. Springer-Verlag, 2003.

[13] A. Podelske and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, Fifth International Workshop, VMCAI 2004*, volume 2937 of *LNCS*, pages 239–251, 2004.

[14] S. Ray and J. S. Moore. Proof Styles in Operational Semantics. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-aided Design (FMCAD 2004)*, number 3312 in LNCS, pages 67–81. Springer-Verlag, Nov. 2004.

[15] K. Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, Institut für Informatik, Technische Universität München, 1999.

[16] K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs'00*, number 1869 in Lecture Notes in Computer Science, pages 498–518, Portland, Oregon, USA, August 2000. Springer-Verlag.

[17] K. Slind. Wellfounded schematic definitions. In D. McAllester, editor, *Proceedings of the Seventeenth International Conference on Automated Deduction CADE-17*, volume 1831, pages 45–63, Pittsburgh, Pennsylvania, June 2000. Springer-Verlag.