USING THE ACL2 THEOREM PROVER: A TUTORIAL INTRODUCTION AND CASE STUDIES

Chapter 1

HIGH-SPEED, ANALYZABLE SIMULATORS

David Greve, Matthew Wilding, and David Hardin*
Rockwell Collins Advanced Technology Center
Cedar Rapids, Iowa 52498
{dagreve, mmwildin}@collins.rockwell.com, david.hardin@ajile.com

Abstract High-speed simulation models are routinely developed during the design of complex hardware systems in order to predict performance, detect design flaws, and allow hardware/software co-design. Writing such an executable model in ACL2 brings the additional benefit of formal analysis; however, much care is required to construct an ACL2 model that is both fast and analyzable. In this chapter, we develop techniques for the construction of high-speed formally analyzable simulators in ACL2, and demonstrate their utility on a simple processor model.

Introduction

It is common practice when designing sophisticated hardware systems to construct simulators to predict performance and to flush out design flaws as early as possible. Hardware simulators also enable early evaluation of system behavior and allow hardware/software co-design. Such simulators are commonly implemented in C or C++ and provide either clock-cycle or behavioral-level accuracy. These simulators must execute quickly because they are often used interactively or are expected to execute regression tests composed of millions of vectors.

ACL2 is a particularly attractive system for developing hardware simulators. ACL2 not only allows for the construction of a formal representation of hardware designs, it also enables execution of that representation as a simulator. ACL2's logic is constructed on top of a real programming language, Common Lisp, and supports a wide variety of

*David Hardin's current affiliation: Ajile Systems, Inc., Oakdale, IA 52319

language features that add nothing to the logic but enable optimizations that are important for executing ACL2 code efficiently. Unification of the device models for formal analysis and simulation has the important additional property of validating the formal model through its execution and validating the simulator through formal verification. However, a good deal of care is required to construct a model that is both high-speed and analyzable.

This chapter explains how one can develop hardware device models that execute quickly and can be analyzed by the ACL2 theorem prover. We begin the chapter with a discussion on exactly how to write ACL2 models that execute quickly. We then present an example from [Wilding et al., 1998] of processor modeling in ACL2 and conclude by highlighting some techniques and ACL2 features that help in reasoning about models of this type.

1. HIGH-SPEED SIMULATORS

A simple approach for writing hardware simulators in software is to define a data structure containing a field corresponding to each register in the hardware design. The actual simulator is then nothing more than a program that reads the current state values from the data structure, computes the next state of the machine based on the current inputs, and updates the data structure fields appropriately. Such simulator models can be quite fast when state data structure access is efficient and the next state function is optimized.

This method for architecting simulators is similar to the interpreter-style of specification used in many previous formal verification projects [Bevier et al., 1989]. In the interpreter style specification, a "next state" function is defined that accepts as input a data structure representing the current state of the machine and returns an updated machine state. A recursive stepper function repeatedly applies the next state function to an initial state to compute the final state after some number of clock cycles.

The challenge of building a high speed simulator in ACL2 is to represent state in a way that allows efficient manipulation and to craft an efficient ACL2 implementation of the next state function. While the following sections detail what an ACL2 user can do to obtain maximal execution efficiency, the extent to which this efficiency is realized depends largely on the Common Lisp compiler used. The optimizations presented here have been tested and shown to work with GCL 2.2.2 under 32-bit Linux and Solaris operating systems.

1.1 REPRESENTING STATE

The first step in developing an efficient hardware simulator is to find an efficient implementation of the state data structure. An efficient implementation provides fast data structure access and updating and imposes negligible runtime overhead.

1.1.1 Lists. One possible option for state representation is lists. Lists are easy to use and are well-supported by the ACL2 theorem prover. Simple lists, however, do not typically provide efficient access and update operations. Association lists, while providing relatively efficient updating, also do not allow for efficient access. The manipulation of lists also imposes significant runtime overhead in the form of garbage collection [Wilding et al., 1998].

1.1.2 Arrays. A more ambitious implementation might wish to make use of Common Lisp arrays or structures to store the machine state. Such constructs are good for representing fixed-size data structures such as the state of a machine and provide extremely efficient access and update capability. The use of such imperative Common Lisp objects, however, is typically ruled out by the applicative nature of ACL2.

ACL2 does, however, support a clever implementation of applicative arrays that provides constant time access and destructive updates. Unfortunately, ACL2 maintains 2 versions of the array contents in order to satisfy the applicative requirements of ACL2. Besides the imperative array implementation, ACL2 uses an association list implementation that is used when a multi-threaded manipulation of the data structure is encountered. Experience shows that, while better than lists, ACL2 arrays still impose significant runtime overhead in the form of garbage collection [Wilding et al., 1998].

1.1.3 Single-Threaded Objects. The efficiency of ACL2's implementation of arrays is at the heart of the difference between imperative and applicative programming languages. Imperative languages have an implicit concept of state. It is possible to modify this implicit state without "returning" the new value of that state. This is commonly known as a side effect of executing an imperative program. Common Lisp, for example, provides constructs that allows destructive modification of global variables in an imperative way. ACL2, however, omits such constructs from its logic.

In contrast to imperative languages, functional languages do not maintain global state. In a purely functional language, there are no side effects, no destructive updates, and no global variables. Modified versions of any so-called state variables in a functional language must be returned to the calling context in order to be visible.

Consider the following ACL2 code fragment that represents the computation of the next state of a model whose state is stored in a data structure called HWSTATE that is manipulated by functions of the form update-N and access-N:

```
(let ((HWSTATE (update-1 HWSTATE)))
  (let ((var (access-1 (update-2 HWSTATE))))
    (let ((HWSTATE (update-3 HWSTATE)))
```

Which version of HWSTATE does update-3 operate on, the version produced by update-1 or the version produced by update-2? In applicative languages like the logic of ACL2, update-3 operates on the version of HWSTATE produced by update-1. The implication of this is that during the process of computing the value to be bound to var, there must be two copies of state: one produced by update-1 and used eventually by update-3 and one produced by update-2 and used immediately by access-1.

Maintaining multiple copies of state, however, leads to inefficient execution. The natural way to implement such programs in imperative languages is by using global state. This is faster since multiple copies are not maintained, but allowing for side effects makes reasoning about imperative programs difficult.

As of version 2.4, ACL2 provides a facility, called single-threaded objects, for defining data structures with applicative semantics but an imperative implementation.[Robert S. Boyer and J S. Moore, 1999] These objects provide constant-time access and destructive update operators without imposing any runtime overhead. ACL2 enforces syntactic restrictions on the use of single-threaded objects that disallow cases in which the applicative and imperative semantics differ, such as in the example given above. ACL2's single-threaded objects provide an efficient, imperative data structure implementation while still allowing program reasoning using applicative logic.

1.2 EFFICIENT NEXT-STATE COMPUTATION

Lisp has a dubious reputation when it comes to execution speed. However, one can write Lisp code that executes with nearly the speed of optimized C code when simulating hardware models.

1.2.1 Compilation and ACL2. The first step in obtaining efficient execution of Lisp code is to compile the code. While most Lisp systems support interpreted execution of Lisp code, compiled Lisp runs significantly faster. ACL2 takes pains to support a wide variety of language features that add no real value to the logic but enable optimization of the resulting compiled code.

The ACL2 feature that is perhaps the most useful in this respect is guards. One kind of ACL2 guard is a Common Lisp type declaration. Type declarations are used in Common Lisp to insert interpretive type checks and to improve the compiled execution efficiency. One important use of guards in ACL2 is conveying type information to the Common Lisp compiler to enable compile time optimizations.

Common Lisp systems typically treat type declarations as unchecked conversions at compile time. The ACL2 user, however, is required to prove that the guards of a function are completely satisfied before ACL2 will allow the execution of the resulting optimized Lisp code. In this way, ACL2 guarantees via proof that type declarations result in safe type conversions at compile time, thus avoiding potentially unspecified behavior. Guard proofs are important in obtaining optimal performance from functions defined in ACL2.

1.2.2 Arithmetic. For maximal efficiency integer values should be used to encode the current state of the machine. The use of integer values to represent state makes efficient arithmetic operations crucial. Common Lisp integers, however, are of arbitrary precision by default. This is in contrast to C, where the results of arithmetic operations are automatically cast to fixed-sized integers.

Although arbitrary precision is sometimes handy, it is undesirable for our purposes as it is generally quite slow. Bignum integers (as they are called in Common Lisp) are *boxed*, meaning that their values are represented by objects rather than as raw machine integers. Creating boxed integers requires memory allocation and destroying boxed integers involves garbage collection. Operations such as addition and multiplication cannot be performed directly on boxed objects using the native addition and multiplication instructions of the host machine because boxed objects are not represented in a format native to the host machine.

Another reason arbitrary precision integers are undesirable is that the arithmetic we want to use to describe hardware devices is modular arithmetic. This is simply because hardware performs operations on fixed-word-width values.

Common Lisp does not provide primitive support for modular arithmetic and Common Lisp arithmetic and logical functions consume and produce bignum values by default. Even so, it is possible to optimize integer operations in Common Lisp, as Guy Steele points out:

> In every Common Lisp implementation there is a range of integers that are represented more efficiently than others; each such integer is called a fixnum, and an integer that is not a fixnum is called a bignum. Common Lisp is designed to hide this distinction as much as possible; the distinction between fixnums and bignums is visible to the user in only a few places where the efficiency of representation is important. Exactly which integers are fixnums is implementation-dependent; typically they will be those integers in the range -2^n to 2^n-1, inclusive, for some n not less than 15. [Steele, 1990]

In the following sections we explore the boundaries of those "few" places where the efficiency is important and attempt to produce ACL2 code that performs integer operations efficiently.

1.2.3 Declarations. In Common Lisp, declarations are a mechanism that can be used for advising the compiler of potential optimizations based on the argument and return types of various functions.

> [...] declarations are of an advisory nature, and may be used by the Lisp system to aid the programmer by performing extra error checking or producing more efficient compiled code. Declarations are also a good way to add documentation to a program. [Steele, 1990]

By declaring that a `+` operation can be interpreted as operating on only fixnum values, a good Common Lisp compiler will replace the functionality normally used to compute a bignum `+` with a more efficient fixnum `+` operation. This can result in significant savings both in execution time and in operating overhead by avoiding garbage collection. For most built-in integer operations in Common Lisp there exists a more efficient version of the functionality that operates on fixnum values. The only requirement for using the fixnum functionality is that the compiler be informed, through declarations, that the operation will be used with values representable as fixnums.

There are two kinds of type declarations in Common Lisp. Binding type declarations are type declarations associated with variable bindings such as function arguments and variables bound in let expressions. The following example illustrates both argument and let binding declarations:

```
(defun foo (a b)
  (declare (type (signed-byte 32) a)
           (type (signed-byte 32) b))
  (let ((x (goo a b)))
```

```
    (declare (type (signed-byte 32) x))
    x))
```

A second form of declaration is the inline `the` type declaration that allows the user to declare intermediate computational results.

> Frequently it is useful to declare that the value produced by the evaluation of some form will be of a particular type. Using declare one can declare the type of the value held by a bound variable, but there is no easy way to declare the type of the value of an unnamed form. For this purpose the the special form is defined; (the type form) means that the value of form is declared to be of type type. [Steele, 1990]

In the following example we use an inline declaration to assert that the result of calling `goo` is a `(signed-byte 32)`:

```
(defun foo (a b)
  (declare (type (signed-byte 32) a)
           (type (signed-byte 32) b))
  (let ((x (the (signed-byte 32) (goo a b))))
    (declare (type (signed-byte 32) x))
    x))
```

Note that the degree to which declarations improve the efficiency of any function is strongly dependent on the sophistication of the Lisp compiler being used. Some compilers are good at keeping track of variable types, inferring the types of bound variables or establishing function return types. For optimal compiler independence, one might attempt to declare every occurrence of every term and every bound variable. While such overkill should never decrease the compiled performance of the code, it may significantly decrease the interpreted execution speed and, further, over-employing declarations may lead to excessive or redundant guard proof obligations. A good rule of thumb is to declare all bound integer variables and all intermediate integer value results.

1.2.4 Functions. To preserve modularity, nontrivial simulators require the use of several levels of function definitions. One must expend effort in optimizing function calls in Lisp in much the same way one optimizes arithmetic operations.

Proclamations. Most modern computer systems provide efficient means for returning values from function calls. If properly informed, most Lisp systems can take advantage of this fact during the compilation process and optimize function calls so that they do not create garbage by pushing boxed values onto the Lisp value stack during the function call.

In Common Lisp, one informs the compiler of a function that can directly return fixnum values using the `proclaim` form:

```
(proclaim foo (fixnum fixnum) fixnum)
```

In ACL2 `proclaim` is not an accepted form. However, if the body of a function definition is enclosed in a `the` declaration form that indicates that the function is expected to return a fixnum value then ACL2 will automatically proclaim the function to the Common Lisp compiler. This process informs the Lisp compiler when a function can be optimized to use native calling procedures.[1]

Multi-value returns. While ACL2 provides an implementation of the Common Lisp multi-valued function return, it is often not possible to optimize such function calls to use native calling procedures. For this reason, the use of multi-value returns should be avoided when writing efficient models.

Although prohibiting multi-value returns may seem inconsequential, it can be significant when writing functional code for use in an interpreter. It is not uncommon to wish to update the state in a function and at the same time return a value, perhaps indicating some sort of status. Without multi-valued returns, however, this is difficult given the applicative nature of ACL2 and the fact that functions that update single threaded objects must return a reference to the updated object.

A determined programmer can, however, emulate a Lisp value stack through the addition of scratch variables to the single threaded object itself. For example, the following function returns two values to its calling context by storing them in the registers `SCRATCH1` and `SCRATCH2`. This is a serious programming hack, however, as it violates the concept of separation of data structure and implementation and should only be done *in extremis*.

```
(defstobj HWSTATE
  (DATA     :type (array integer (10)) :initially 1)
  (SCRATCH1 :type integer              :initially 0)
  (SCRATCH2 :type integer              :initially 0))
```

[1] As we will see in Section 1.2.4, however, ACL2 will not accept recursive function definitions whose outermost form is a `the` statement.

```
(defmacro 2V-RETURN (v1 v2)
  `(let ((HWSTATE (update-SCRATCH1 ,v1 HWSTATE)))
     (let ((HWSTATE (update-SCRATCH2 ,v2 HWSTATE)))
       HWSTATE)))

(defmacro 2V-BIND (vars fncall form)
  `(let ((HWSTATE ,fncall))
     (let ((,(first vars) (SCRATCH1 HWSTATE)))
       (let ((,(second vars) (SCRATCH2 HWSTATE)))
         ,form))))

(defun FWT (a b HWSTATE)
  (declare (xargs :stobjs (HWSTATE)))
  (2V-RETURN (+ a b) (- a b)))

(defun APPLY-FWT (HWSTATE)
  (declare (xargs :stobjs (HWSTATE)))
  (2V-BIND (x y) (FWT (DATAi 0 HWSTATE) (DATAi 1 HWSTATE) HWSTATE)
     (let ((HWSTATE (update-DATAi 0 x HWSTATE)))
       (let ((HWSTATE (update-DATAi 1 y HWSTATE)))
         HWSTATE))))
```

Recursive functions. It is often desirable to define functionality iteratively using recursion in ACL2. Recursive functions, however, present a couple of unique optimization challenges. The first is that ACL2 does not allow one to wrap the body of a recursive function with a `the` form and, thus, ACL2 is not inclined to proclaim the function to the compiler for function call optimization. As a result it is not possible to completely optimize recursive functions to use native calling procedures and recursive functions returning integers will likely be compiled to return bignum results.

The second hurdle becomes apparent when one attempts to submit a form such as:

```
(defun foo (x)
  (declare (type (signed-byte 32) x))
  (if (> x 0) (foo (Int32 (1- x))) 0))
```

This function is not admissible in the logic of ACL2. In order to guarantee the termination of this recursive function, one must establish that `x` is an integer. Otherwise the value returned by `(1- x)` is unspecified in Common Lisp and the function is not guaranteed to terminate. One might argue that because `x` has been declared to be of type `(signed-byte 32)` it must be an integer. This, however, is not the case because guards are not a part of the ACL2 logic. In other words, when performing the termination proof of this function, one is not allowed to assume that `x` is a `(signed-byte 32)`. This can lead to some redundancy between function definitions and type declarations:

```
(defun foo (x)
  (declare (type (signed-byte 32) x))
  (if (and (integerp x)
           (> x 0))
      (foo (Int32 (1- x))) 0))
```

While this definition is acceptable to ACL2, note that the implementation of `foo` is inefficient in that it has an unneeded call to `integerp` during each recursive iteration. This is particularly frustrating considering that we expect `x` always to satisfy this test.

One saving grace in this particular case is that `integerp` is a built-in Lisp function and many compilers are capable of detecting, from the type declarations, that `x` is an integer value and that any call to `(integerp x)` will return `T`, thus allowing the compiler to optimize away the function call. It may not be possible to optimize recursive functions for compiler implementations for which this is not the case.

1.2.5 Macros. ACL2 support of Common Lisp Macros is another example of something that adds little to the ACL2 logic but has a significant impact on the usability of the ACL2 system on real problems. Lisp macros can make writing efficient, readable code much simpler. Macros can also be used to improve execution efficiency and readability in several ways.

Inlining. Macros provide ACL2 users with compiler-independent constant and function inlining. Although ACL2 provides a `defconst` event that allows the definition of a symbolic name for a constant value, it does not necessarily result in efficient executable code. Macros are always efficient because they are evaluated prior to compilation, thus allowing inlining of constant values.

```
(defmacro MAX_INT<32> ()  2147483647)
```

```
(defmacro MIN_INT<32> () -2147483648)
```

Function inlining can be implemented using macros in a similar way. Following is an example of a macro implementation of a 32-bit version of the Common Lisp increment function.

```
(defmacro 1+<32> (x)
  `(if (= (MAX_INT<32>) ,x) (MIN_INT<32>)
     (Int32 (+ ,x 1))))
```

One drawback of using macros to replace functions in this way is that one cannot control the expansion of macro expressions in the theorem prover. In this case, for example, one cannot disable the definition of `1+<32>` and then reason about it directly.

Syntactic Sugar. Macros can be used to simplify the syntax of otherwise cumbersome Lisp expressions. The `the` declarations are an especially good candidate for this type of macro.

```
(defmacro Int32 (x) `(the (signed-byte 32) ,x))
```

Macros can also be used to wrap functions with type declarations, thus encouraging compiler optimizations and simplifying the syntax of such declarations. For example, one might define:

```
(defmacro logand<32> (a b)
  `(Int32 (logand (Int32 ,a) (Int32 ,b))))
```

allowing us to write:

```
(logand<32> x y)
```

instead of:

```
(the (signed-byte 32)
     (logand (the (signed-byte 32) x)
             (the (signed-byte 32) y)))
```

Readers. Macros can be used to define Lisp readers to change the meanings of predefined Lisp function symbols. The following simplistic reader macro converts the standard Lisp + symbol into the +<32> (fixnum addition) symbol. (Note that this macro doesn't work over many Common Lisp special forms.)

```
(defun map-fn (fn form)
  (if (consp form)
      `((,fn ,(car form)) ,@(map-fn fn (cdr form)))
    nil))

(defmacro reader (form)
  (case (car form)
      ((+) `(+<32> (reader ,(second form))
                   (reader ,(third form))))
      (t   `(,(first form) (map-fn 'form (cdr form))))))
```

This allows us to write functions succinctly.

```
(defun foo (x) (reader (+ (+ x 2) 3)))
```

While this type of reader macro doesn't directly improve efficiency, it can make writing and reading optimized Lisp code much easier. By re-mapping Lisp operations to their fixnum equivalents, the clutter introduced by type declarations can be largely overcome.

Exercise 1.1 *What is the macroexpansion of* `foo`*'s body?*

Exercise 1.2 *Extend the reader macro presented above to handle the special Lisp forms* `case` *and* `cond`*. (Remember,* `case` *key forms are not evaluated!) Expand its capability to handle some of the other basic integer operations such as* `logand`*,* `logior`*, etc. Try to automatically insert type declarations into simple* `let` *bindings, assuming that each* `let` *binds a fixnum value.*

`progn` **Emulation.** The syntactic restrictions imposed by single threaded objects make writing and reading code that manipulates these data structures difficult. For functions involving deeply nested updates of single threaded objects, it is convenient to define a `progn`-style macro that automatically binds a sequence of state updates.

```
(defmacro ST-PROGN (stobj &rest rst)
```

```
  (cond ((endp rst) stobj)
        ((endp (cdr rst)) (car rst))
        (t `(let ((,stobj ,(car rst)))
             (ST-PROGN ,stobj ,@(cdr rst))))))
```

This allows us to write a sequence of HWSTATE updates as follows:

```
(ST-PROGN HWSTATE
          (update-1 HWSTATE)
          (update-2 HWSTATE)
          (update-3 HWSTATE))
```

Exercise 1.3 *Write a* `progn`*-style macro that binds single threaded object updates and allows for the introduction of* `let` *bindings of intermediate variables. For example :*

```
(ST-PROGN HWSTATE
   (update-1 HWSTATE)
   (let ((value1 (+ 4 5))))
   (update-2 value1 HWSTATE))
```

should expand to:

```
(let ((HWSTATE (update-1 HWSTATE)))
  (let ((value1 (+ 4 5)))
    (let ((HWSTATE (update-2 value1 HWSTATE)))
      HWSTATE)))
```

1.3 32-BIT MODULAR ADDITION

In this section we bring together many of the concepts presented so far concerning efficient execution of ACL2 programs. We illustrate these techniques by writing a function that accepts two 32-bit integer arguments and returns their 32-bit modular sum. Assuming that fixnum values are at least 32-bit integers, this function uses only fixnum operations.

We start by defining a type macro to relieve us of typing `the` forms:

```
(defmacro Int32 (x)
```

```
  `(the (signed-byte 32) ,x))
```

Our first attempt at a 32-bit modular addition function is:

```
(defun plus<32> (a b)
  (declare
   (type (signed-byte 32) a)
   (type (signed-byte 32) b)
   (xargs :verify-guards nil))
  (Int32 (+ a b)))
```

This function usually works in raw Common Lisp, assuming one is executing on a 32-bit machine. However, when we submit the event `(verify-guards plus<32>)` to ACL2, we find that the guards for the function are unsatisfiable. The reason this function usually performs as expected in raw Common Lisp is that Common Lisp accepts the type declarations at face value and optimizes the + function accordingly. The reason the guard proof fails in ACL2, however, is that it is not necessarily the case that when you add two 32-bit integers that you will obtain a result representable in 32 bits.

One possible solution to this problem is to constrain `a` and `b` in such a way as to guarantee that the result is representable in 32 bits. We can do this using the `(xargs :guard)` argument to write a more expressive guard statement such as the following:

```
(defmacro MIN_INT<32> () -2147483648)

(defmacro MAX_INT<32> ()  2147483647)

(defun plus<32> (a b)
  (declare (type (signed-byte 32) a)
           (type (signed-byte 32) b)
           (xargs :guard (and (<= (MAX_INT<32>) (+ a b))
                              (>= (MIN_INT<32>) (+ a b)))))
  (Int32 (+ a b)))
```

The guards for this function are now satisfiable and this function does in fact compile into an efficient implementation. Unfortunately, there are a couple of problems with this particular implementation. The first is that any other function that calls this function must be able to demonstrate not only that the two arguments passed to `plus<32>` are representable as 32-bit integers but that the sum of those two arguments is representable in 32-bits as well. While this may be possible in some special cases, it will not be true in general. The second problem with this function is that it does not satisfy our original requirements because it does not actually perform modular integer addition. In modular integer addition there are cases when both operands are positive and the result is negative. Such overflow cases, however, are specifically excluded by the guards of `plus<32>`.

With these observations in mind, we now present a function that satisfies our original requirements:

```
(defmacro +<32> (x y) `(Int32 (+ ,x ,y)))

(defun plus<32> (a b)
  (declare (type (signed-byte 32) a)
           (type (signed-byte 32) b))
  (Int32 (if (< a 0)
             (if (>= b 0) (+<32> a b)
               (let ((psum (+<32> b
                                  (+<32> 1
                                    (+<32> a (MAX_INT<32>))))))
                 (declare (type (signed-byte 32) psum))
                 (if (< psum 0)
                     (+<32> (+<32> (MAX_INT<32>) psum) 1)
                   (+<32> psum (MIN_INT<32>)))))
           (if (< b 0) (+<32> a b)
             (let ((psum (+<32> (+<32> a (MIN_INT<32>)) b)))
               (declare (type (signed-byte 32) psum))
               (if (>= psum 0)
                   (+<32> (MIN_INT<32>) psum)
                 (+<32> (+<32> psum (MAX_INT<32>)) 1)))))))
```

Given the relative complexity of the above function, it is interesting to note that it is exactly the expressive power of Common Lisp that makes it so difficult to implement such a seemingly simple procedure. One false move during any intermediate step of the algorithm will result in the creation of a bignum integer object or guards conjectures that are unprovable. While it is somewhat frustrating that one cannot directly access the low level 32-bit modular addition functions that often exist natively on the machine, it is nonetheless comforting to know that ACL2 will not allow us to create models whose execution is inconsistent with their formal description.

Exercise 1.4 *Prove that, given two 32-bit integers arguments,* `plus<32>` *implements 32-bit modular integer addition. Hint: You may want to use* `logext` *and* `logext-+-logext` *from the IHS books provided with the ACL2 distribution to specify and prove this behavior.*

Exercise 1.5 *Implement a function that accepts two 32-bit integer values and returns their 32-bit integer bitwise logical and. Was this function simpler or more complex than* `plus<32>`*? Were the guards simpler or more complex to prove?*

2. TINY : A HIGH-SPEED SIMULATOR EXAMPLE

In the events accompanying this book there is an illustration of the use of ACL2 to develop a high speed model of a simple processor called TINY. The TINY machine is a simple, stack-based machine whose state is composed of a program counter, a memory of integers, a data stack top-of-stack pointer, and a call stack top-of-stack pointer.

The TINY model is represented in the ACL2 logic as an interpreter that operates over a single threaded object. This single threaded object contains fields corresponding to each of the TINY state elements and all of the entries in this single threaded object are represented using fixnum integer values. The following event defines the single threaded object used to represent the TINY state:

```
(defstobj st
 (progc :type (unsigned-byte 10)                 :initially 0)
 (mem   :type (array (signed-byte 32) (1024)) :initially 0)
 (dtos  :type (unsigned-byte 10)                 :initially 0)
 (ctos  :type (unsigned-byte 10)                 :initially 0))
```

A series of ACL2 functions and macros designed to operate efficiently over this state are introduced and ultimately composed to define the function `next`, the TINY next-state function. This function interprets the TINY instructions with respect to their effect on the TINY state. The TINY instruction set is described in Figure 1.1. The ACL2 term `(tiny s n)` evaluates to a TINY state that reflects the effect of executing `n` instructions on an initial TINY state `s`.

Figure 1.2 presents a TINY program that calculates the integer remainder of two positive values in memory using repeated subtraction. Since we have expressed the TINY model in the ACL2 logic, we can use the ACL2 theorem prover to prove interesting properties about this code. Rather than reason about the assembly code, however, we reason about

Opcode	Numonic	Description
0	**pop a**	Pop a data stack value into location **a**
1	**pushs a**	Push the value at location **a** onto the data stack
2	**pushsi i**	Push **i** onto the data stack
3	**add**	Pop two data stack values and push their 32-bit sum
4	**jump a**	Set the program counter (PC) to **a**
5	**jumpz a**	Set the PC to **a** if popped data stack value is 0
6	**call a**	Push the next address on the call stack and set the PC to **a**
7	**ret**	Set the PC to the popped call stack value
8	**sub**	Pop two values and push the maximum of 0 and their 32-bit difference
9	**dup**	Duplicate the top of data stack

Figure 1.1 TINY Instructions

the binary code that is actually loaded onto the machine. Techniques for doing such proofs are presented in the following chapter.

3. SIMULATOR EXECUTION ANALYSIS

An ACL2 proof about the remainder program presented in Section 2. is provided in the events that accompany this book. This section highlights several aspects of this proof that are generally useful for this kind of analysis, and points out particularly some advanced features of ACL2's simplifier that are useful. Most or all of these techniques have been introduced or presented in various publications, most comprehensively in [Boyer and Moore, 1996].

Assembly			Comment
.ORG	18		Origin Program at address 18
VAL	0		value
MODU	0		modulus (assumed positive)
REM	pop	MODU	
	pop	VAL	
LOOP	pushs	VAL	repeated subtraction loop
	pushs	MODU	
	sub		
	dup		
	jumpz	DOSUB	end loop when VAL <= MODU
	pop	VAL	
	jump	LOOP	
DOSUB	pushs	MODU	
	pushs	VAL	
	sub		
	jumpz	EXIT1	
	jump	EXIT2	
EXIT1	pushsi	0	VAL=MODU: set VAL to 0
	pop	VAL	
EXIT2	pop	MODU	RETURN VAL on stack
	pushs	VAL	
	ret		

Figure 1.2 A TINY Program for Calculating Remainder

3.1 ALGORITHM PROOFS AND SYMBOLIC SIMULATION

The remainder program from Section 2. calculates the remainder of two positive integers by repeated subtraction. We can specify its behavior using the ACL2 (and Common Lisp) function `mod`.[2]

```
(defun good-initial-remainder-state (st)
  (declare (xargs :stobjs (st)))
   (and
    (stp st)
    (< (dtos st) 1000) (>= (dtos st) 100)
    (<= (ctos st) 1020) (> (ctos st) 1010)
    (equal (progc st) 24)
    (program-loaded st *mod-prog* 20)
    (< 0 (memi 18 st))
    (< 0 (memi 19 st))))

(defthm mod-correct
  (let ((s2 (tiny st (remclock st))))
  (implies
    (good-initial-remainder-state st)
    (equal (memi (1+ (dtos s2)) s2)
           (mod (memi 18 st) (memi 19 st))))))
```

The function `good-initial-remainder-state` identifies TINY states that have data stack and call stack register values within certain bounds, a program counter that points to the beginning of the remainder program, a memory that contains the remainder program, and positive values in two specific locations. The remainder program we refer to here is the binary version of the remainder program listed in Figure 1.2. A useful property of the remainder program is described by `mod-correct`: the program deposits the expected value at the top of the data stack.

It is always important when proving theorems in ACL2 to partition the proof effort as much as possible. An excellent way to partition a proof about simulator execution such as `mod-correct` is to divide the work into proving two sublemmas:

[2]We omit `:hints` and `:rule-classes` arguments of `defthm`s in this chapter when they are irrelevant to the discussion. The unexpurgated events can be found on the CD that accompanies this book.

- a lemma that describes the effect the simulator's execution has on the machine state, and
- a lemma that shows that the implemented algorithm works.

We describe both these kinds of proofs in the following sections.

3.2 IMPLEMENTATION PROOFS

3.2.1 Symbolic Simulation. To prove a lemma describing the simulator's effect on the machine state, we must specify the behavior of the program. We introduce a function that calculates the result of the execution of the simulator in exactly the same way as the program. That is, we implement the algorithm being reasoned about in the ACL2 logic directly, not using the interpreter, and use the same types of conditions and operations as are used in the implementation.

For example, the result of the remainder program is calculated by `remainder-prog-result`.

```
(defun remainder-prog-result (n p)
  (if (or (zp p) (zp n) (< n p))
      (nfix n)
    (remainder-prog-result (- n p) p)))
```

Note that `remainder-prog-result` calculates remainder just as the program does, and performs the same operations each iteration of the loop. Using this function as a specification of the program execution simplifies the proof.

We wish to show that the symbolic expression that represents the result of running the remainder program is in part described by the function `remainder-prog-result`. In order to use the ACL2 theorem prover to generate the symbolic result we introduce ACL2 rules that direct the simplifier to execute the interpreter symbolically. For straightline code where the number of instructions is constant this is straightforward. The following rule causes the ACL2 simplifier to execute the TINY interpreter symbolically. Note the hypothesis involving `syntaxp` requires that this rule is applied only to terms `(tiny st n)` where `n` is a constant.

```
(defthm tiny-straightline
  (implies
   (syntaxp (quotep n))
   (equal (tiny st n)
          (if (zp n) st (tiny (next st) (1- n))))))
```

```
(UPDATE-NTH
        0 24
        (UPDATE-NTH
             1
             (UPDATE-NTH 18
                         (+ (NTH 18 (NTH 1 ST))
                            (- (NTH 19 (NTH 1 ST))))
                         (UPDATE-NTH (NTH 2 ST)
                                     (+ (NTH 18 (NTH 1 ST))
                                        (- (NTH 19 (NTH 1 ST))))
                                     (UPDATE-NTH (+ -1 (NTH 2 ST))
                                                 (+ (NTH 18 (NTH 1 ST))
                                                    (- (NTH 19 (NTH 1 ST))))
                                                 (NTH 1 ST))))
             ST))
```

Figure 1.3 Symbolic expression resulting from one iteration of **remainder**'s loop

Using `tiny-straightline` we can prove, for example, what happens when the loop in the **remainder** program is executed once. There are seven instructions in the body of the loop, and `tiny-straightline` together with other rules rewrites the term `(tiny st 7)` to the symbolic result of Figure 1.3, given the assumption that `st` represents a good TINY state just about to iterate on the loop.

While `tiny-straightline` works well for constant number of instructions, it will not work on more complex code, such as code containing loops. In order to analyze more complex code we first introduce a *clock* function that calculates how many instructions are executed.[3] To automate the reasoning process, the clock function has the structure of the executing code. Straightline code is represented by a constant, alternation is represented using an `if`, and loops are represented using recursive functions. The ACL2 simplifier will expand the interpreter for straightline code and case-split for alternation. Loops will be dealt with by setting up inductive arguments that mimic the recursion of the clock function.

We define the function `c+` as `+` on natural numbers and then disable its definition. We use `c+` rather than `+` for fear that other rules about a commonly-used function such as `+` will interfere with these proofs. The following rule simplifies expressions that we will encounter that

[3]This technique has been used many times previously. Bevier was perhaps the first to do a proof of this type for his KIT operating system kernel. Explicitly using a clock function to control the prover was introduced in [Boyer and Yu, 1996], and these same ideas work using the PVS theorem prover [Wilding, 1997]. The idea is documented fully in [Boyer and Moore, 1996].

contain `c+` in their clock expressions by decomposing their execution into constituent parts.

```
(defthm tiny-c+
  (equal (tiny s (c+ x y))
         (tiny (tiny s x) y)))
```

The function `modloop-clock` presented below calculates the number of instructions required to execute the loop of our example program:

```
(defun modloop-clock-helper (x y)
  (if (or (not (integerp x)) (not (integerp y))
          (>= 0 y) (<= x y))
      (if (equal x y) 14 13)
    (c+ 7 (modloop-clock-helper (- x y) y))))

(defun modloop-clock (st)
  (declare (xargs :stobjs (st)))
  (let ((x (memi 18 st)) (y (memi 19 st)))
    (modloop-clock-helper x y)))
```

These rules are sufficient to derive symbolic expressions that describe the results. However, as can be seen in Figure 1.3, the expressions that result are very complex, even for a toy model like TINY and a simple program. Note that since an instruction typically updates a few fields with values calculated from the previous state, the size of the term grows exponentially in the number of instructions. The example symbolic expression in Figure 1.3 is complex, but not that complex! In the next section we introduce some ACL2 rules that simplify terms of this kind.

Exercise 1.6 *Calculate a symbolic expression that expresses the result of executing the TINY interpreter on the first two instructions of the remainder program. Simplify that expression using the fact that an update of an element of the state does not effect the other elements.*

3.2.2 Simplifying terms with access/update functions. In the previous section we described how to arrange for ACL2 to generate symbolic expressions that describe the effect of running the simulator. The resulting terms contain access and update functions that are used to define the interpreter. In order to automate proofs involving such expressions we must simplify and normalize them. One of the first challenges an ACL2 user confronts when facing such a proof is deciding which functions to disable.

Should we disable and reason about functions such as `progc` and `update-memi`, or enable them and reason about the functions of which they are composed, namely `nth` and `update-nth`? We choose to enable these functions so that we can reason about `nth` and `update-nth`. In this way we can prove a handful of rules about these two functions. If we left the access and update functions enabled we would need to add a rule for each accessor/updater pair. A model with 100 fields, for example, would require 10,000 rules just to reason about accessing an updated state!

One potential downside to the decision to expand the access and update functions during proofs is that rules we prove about the elements of the state are dependent upon their location in the state description. In the example we might want to write a theorem that simplifies the `progc` of the state and so place `(progc s)` on the left-hand side of a rewrite rule. Unfortunately the term will already have been rewritten to `(nth 0 s)` since `progc` is enabled. We might naively use `(nth 0 s)` on the left-hand side of our rule, but this is not good practice: if we change the location of `progc` in our representation of state our proofs will no longer work. We follow the convention of defining the location of the elements in the state symbolically using a macro and prove rules involving that element using that symbolic name. So, we define

```
(defmacro progcloc () 0)
```

and use `(nth (progcloc) s)` and `(update-nth (progcloc) v s)` in theorems about it. Note how this approach takes advantage of the fact that ACL2 rewrite rules have their macros expanded when they are introduced, before they are applied as rewrite rules.

The symbolic result of executing code on the simulator will be a term consisting of a nest of `update-nth` functions that reflect changes to the state of the machine. Of course, we want to simplify these expressions as much as possible to further our analysis. One such rule is

```
(defthm nth-update-nth2
   (equal
    (nth n1 (update-nth n2 v l))
    (if (equal (nfix n1) (nfix n2)) v (nth n1 l)))
```

which simplifies the access of updated state. We'd prefer not to introduce casesplits, so we introduce versions of `nth-update-nth2` that backchain on whether the the indices of `nth` and `update-nth` match.[4] Note that we list the backchaining rules *after* the casesplitting rule so that they are tried first as ACL2 rewrite rules.

```
(defthm nth-update-nth-1
  (implies
    (not (equal (nfix n1) (nfix n2)))
   (equal (nth n1 (update-nth n2 v l)) (nth n1 l))))

(defthm nth-update-nth-2
  (implies
    (equal (nfix n1) (nfix n2))
   (equal (nth n1 (update-nth n2 v l)) v)))
```

Another useful rule is

```
(defthm update-nth-update-nth-same
  (equal (update-nth i v1 (update-nth i v2 l))
         (update-nth i v1 l)))
```

This rule deletes a list update from a term if the updated element is subsequently replaced. Although this seems a helpful simplification, it will not work on terms that have intervening updates of other list elements. For example, the term

```
(update-nth 2 v1 (update-nth 1 v2 (update-nth 2 v3 l)))
```

is not simplified by `update-nth-update-nth-same` because no subterm matches its left-hand side. What we need is another rule that rearranges the updates of a term so that the updates of a particular element appear consecutively. The following rule does just that:

```
(defthm update-nth-update-nth-diff
  (implies
```

[4] Avoiding casesplits is particularly important when reasoning about lambda expressions, such as are encountered using ACL2 2.4's implementation of `stobj`. If-expressions in the arguments of a lambda expression interfere with beta-reduction of lambda expressions because of ACL2's heuristics for nonrecursive function expansion. This problem is not apparent in the small example of this chapter, but it is a serious issue when reasoning about large `stobj`-based models where if-expressions are unavoidable. Either this problem will be overcome in ACL2 or users will have to use a homegrown version of `stobj` that doesn't have this problem such as is documented in [Wilding et al., 1998]. In either case, the discussion in this chapter applies.

```
  (not (equal (nfix i1) (nfix i2)))
  (equal (update-nth i1 v1 (update-nth i2 v2 l))
         (update-nth i2 v2 (update-nth i1 v1 l))))
 :rule-classes ((:rewrite :loop-stopper ((i1 i2)))))
```

The lemma `update-nth-update-nth-diff` can be used to rearrange the arguments of any term containing nested `update-nth` functions involving distinct fields. We instruct the simplifier using the ACL2 `:loop-stopper` parameter to apply this rule only when it brings us closer to achieving our goal of moving updates of the same field closer to each other. The loop-stopper `(i1 i2)` means that we insist that the simplifier apply `update-nth-update-nth-diff` only when the term that matches `i2` (on the left-hand-side of the rewrite rule) is "less than" the term that matches `i1`. This overrides ACL2's default loop-stopper heuristic which is to apply the rule only when the resulting term is "less than" the original according to ACL2's term ordering heuristic. ACL2 term order for positive integers is identical to standard `<`, so when rewriting the term

```
(update-nth 2 v1 (update-nth 1 v2 (update-nth 2 v3 l)))
```

the subterm

```
(update-nth 1 v2 (update-nth 2 v3 l))
```

is left untouched by `update-nth-update-nth-diff` since $2 \not< 1$. The entire term, however, is simplified since $1 < 2$, leading to

```
(update-nth 1 v2 (update-nth 2 v1 (update-nth 2 v3 l)))
```

The rule `update-nth-update-nth-same` is then applied, simplifying the term to

```
(update-nth 1 v2 (update-nth 2 v1 l))
```

Exercise 1.7 *Read the ACL2 documentation of `:loop-stopper`. Give an example term that `update-nth-update-nth-diff` simplifies but which would not simplified had `update-nth-update-nth-diff` not overridden the default heuristics with an explicit `:loop-stopper` argument.*

The handful of rules described in this section do a good job simplifying terms involving `nth` and `update-nth`.

3.3 ALGORITHM PROOF

The approach to implementation proof that we presented is only part of the story. We must also prove that the algorithm that is implemented does the "right" thing. In the case of our simple example ACL2 proves that the algorithm works without the need for any sublemmas, using the `quotient-remainder` book of the IHS library available in the standard ACL2 distribution.

```
(defthm remainder-is-mod
  (implies
   (and
    (integerp x)
    (<= 0 x)
    (integerp y)
    (<= 0 y))
   (equal (remainder-prog-result x y) (mod x y))))
```

Although for our simple example the algorithm proof is straightforward, this part of a proof is typically very challenging and requires a great deal of domain-specific proof.

3.4 REMAINDER PROOF

The correctness of our remainder example follows directly from the implementation and algorithm proofs of the previous sections:

```
(defthm mod-correct
  (let ((s2 (tiny st (remclock st))))
  (implies
    (good-initial-remainder-state st)
    (equal (memi (1+ (dtos s2)) s2)
           (mod (memi 18 st) (memi 19 st))))))
```

4. CONCLUSION

The unification of simulators for device development and interpreters for formal analysis has many advantages. We have demonstrated in this chapter how one can build a high-speed simulator that can be analyzed using ACL2. These techniques, although shown in the context of a "toy" processor model, have been shown to scale to real processor designs [Hardin et al., 1998, Wilding et al., 1998].

References

[Bevier et al., 1989] Bevier, W. R., Jr., W. A. H., Moore, J. S., and Young, W. D. (1989). An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428.

[Boyer and Moore, 1996] Boyer, R. S. and Moore, J. S. (1996). Mechanized formal reasoning about programs and computing machines. In Veroff, R., editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press.

[Boyer and Yu, 1996] Boyer, R. S. and Yu, Y. (1996). Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192.

[Hardin et al., 1998] Hardin, D., Wilding, M., and Greve, D. (1998). Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Hu, A. J. and Vardi, M. Y., editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Robert S. Boyer and J S. Moore, 1999] Robert S. Boyer and J S. Moore (1999). Single-threaded objects in acl2. `http://-www.cs.utexas.edu/users/moore/publications/index.html`.

[Steele, 1990] Steele, Jr, G. L. (1990). *Common Lisp The Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, second edition.

[Wilding, 1997] Wilding, M. M. (1997). Robust computer system proofs in PVS. In Holloway, C. M. and Hayhurst, K. J., editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356. (http://atb-www.larc.nasa.gov/Lfm97/).

[Wilding et al., 1998] Wilding, M. M., Greve, D. A., and Hardin, D. S. (1998). Efficient simulation of formal processor models. Technical report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498. `http://pobox.com/users/hokie/docs/efm.ps`.

Index