A Mechanically Verified Application for a Mechanically Verified Environment

Matthew Wilding

Computational Logic Inc., 1717 West Sixth Street Suite 290, Austin Texas, USA
and The University of Texas at Austin

Abstract. We have developed a verified application proved to be both effective and efficient. The application generates moves in the puzzle-game Nim and is coded in Piton, a language with a formal semantics and a compiler verified to preserve its semantics on the underlying machine. The Piton compiler is targeted to the FM9001, a recently fabricated verified microprocessor. The Nim program correctness proof makes use of the language semantics that the compiler is proved to implement. Like the Piton compiler proof and FM9001 design proof, the Nim correctness proof is generated using Nqthm, a proof system sometimes known as the Boyer-Moore theorem prover.

1 Introduction

Computer application programs and the systems software and hardware that support them can be very complex. The adoption of traditional software engineering practices such as rigorous testing helps computer programmers write programs with fewer errors, but correctness can not be guaranteed using them. One approach to developing dependable software is to formalize a problem specification in a logic and prove that a particular program written in some computer language meets the requirements of the problem specification. This requires a formal semantics of the language used in the program. Ideally, the language semantics will come with a verified compiler targeted to hardware whose design has been proved to work to specification. Extra reliability can be attained by generating the proofs with a trustworthy automatic theorem prover.

We discuss Nqthm and the proofs of the Piton compiler and the FM9001 design in Section 2. In Section 3 we develop a specification for a program that plays Nim, a centuries-old mathematical game. Section 4 describes an algorithm and Piton implementation of this program. The development of an Nqthm-generated proof that the program described in Section 4 meets the requirements outlined in Section 3 is discussed in Section 5.

[12] describes this project in greater detail and includes the input that causes Nqthm to generate the correctness proof.

2 Background

Nqthm is the name of both a logic and an associated theorem proving system that are documented in [4]. A large number of mathematical theorems from many disparate domains have been proved using Nqthm. The Nqthm logic is a quantifier-free, first-order logic resembling Pure Lisp. The user inputs definitions and conjectures to the Nqthm theorem prover which, when successful, outputs proofs. Proved conjectures are applied in later proofs. In a shallow sense the theorem prover is fully automatic since once a conjecture is input to the theorem prover the proof is uninfluenced by the user. Usually, however, important theorems require the proof and subsequent use of many subsidiary lemmas, and so a carefully designed sequence of conjectures is typically needed to lead Nqthm to a non-trivial proof.

The most important property of Nqthm-generated proofs is soundness. We have confidence that a conjecture for which Nqthm generates a proof is a mathematical truth. Since Nqthm runs on unverified hardware and is not proved correct in a formal way, however, it is possible that a bug in the implementation of Nqthm or its execution environment has caused us to conclude that a false conjecture is a theorem. [4] explores this issue more fully. Although we feel obliged to point out this potential weakness in our work, our experience is that Nqthm-generated proofs are extremely dependable.

One interesting domain to which Nqthm has been applied is computer systems verification. Proofs about computer systems are often mind-numbingly complex but not particularly deep, which is to say perfectly suited to automatic generation. The requirement imposed by Nqthm to get every detail of a proof exactly correct is very important in computer systems verification as even the most trivial-seeming mistake can lead to catastrophe. [8] describes the implementation of a compiler for the language Piton and the associated mechanically-produced correctness theorem. A formal semantics for Piton, a formal description of the FM9001 microprocessor, and the Piton compiler are introduced as Nqthm functions. The compiler correctness theorem relates the data values expected after running a Piton program using Piton's formal semantics to the data values computed by the FM9001 running a compiled Piton program.

The semantics of Piton is described using an interpreter function. A Piton state consists of elements that comprise a programmer's model of how Piton executes: a list of Piton programs to run, a user-addressable stack, a current instruction pointer, a subroutine calling stack, a data area, the word size, stack size limits, and a program status flag. The interpreter function **P** takes as arguments a Piton state and the number of Piton instructions to run, and returns the Piton state resulting from the computation. The semantics of FM9001 is also described using an interpreter function. An FM9001 state consists of elements that comprise a programmer's model for the FM9001 design: values for the register file, the condition flags, and the memory. The interpreter function **FM9001-DESIGN** takes as arguments an FM9001 state and the number of FM9001 instructions to run and returns the FM9001 state resulting from the computation.

The Piton correctness theorem is suggested by Figure 1. **LOAD** is the Piton compiler, and **DISPLAY** is a function that extracts a Piton data segment from an FM9001 image. The Piton compiler is a cross-compiler since the compiler

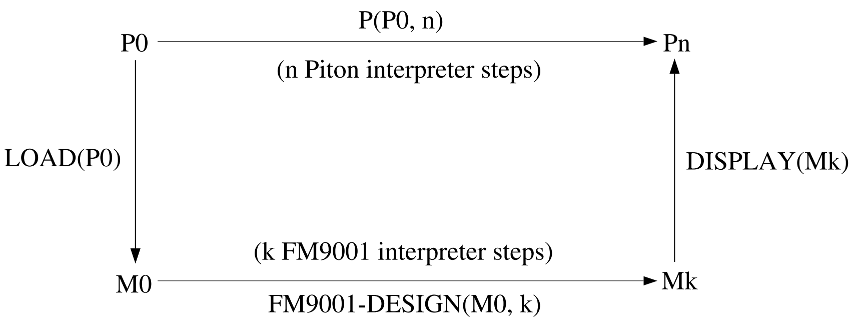

Fig. 1. The Piton correctness proof.

is a function in the Nqthm logic that produces code for another processor, the FM9001. Roughly speaking, the Piton compiler correctness theorem guarantees that the data segment calculated by a Piton program running on the Piton interpreter and the data segment calculated by running a compiled Piton program on the FM9001 are identical. The interpreter function serves as a precise specification for the expected behavior of a system component. This general approach to system verification is described fully in [2]. Interpreter functions can be complex: Piton has 71 instructions and some high-level features, and the definition of **P** in the Nqthm logic requires about 50 pages.

The FM9001 is a fairly conventional microprocessor with an instruction set somewhat like that on a PDP-11 [9]. Unlike most processors the FM9001 has been specified, designed, and proved correct in the sense that the design is shown to meet the specification. The same specification was also used as the target machine in the Piton compiler correctness proof, so the proofs can be "stacked" and we need only assume that the hardware is working properly for a Piton program compiled onto the FM9001 to work according the the formal semantics of Piton. Recently, the FM9001 was fabricated and has run programs, including the compiled Piton program described in this paper [1].

In short, formalized in Nqthm are a language semantics for Piton, a verified compiler to the FM9001, a verified design for the FM9001, and a fabricated chip that implements the FM9001 design. The rest of this paper describes the development of a verified application that runs in this environment.

3 A Specification for a Good Nim Program

Nim is an ancient mathematical game played with piles of stones by two alternating players [5]. On his turn a player removes at least one stone from exactly one pile. The player who removes the final stone loses.

We construct a specification for a program that calculates a good Nim move efficiently. The specification is in the form of several Nqthm predicates listed in this section that introduce five undefined functions related to the implementation. The programmer provides these functions and is obliged to prove that they conform to the constraints of this specification. The five functions are:

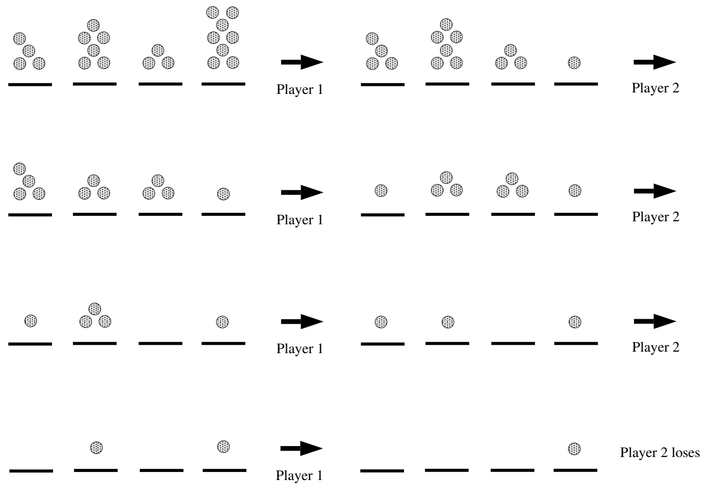

Fig. 2. An example Nim game

- `CM-PROG` : the program that implements a good Nim move generator;
- `COMPUTER-MOVE-CLOCK` : number of instructions the program will execute;
- `NIM-PITON-CTRL-STK-REQUIREMENT` : upper bound on control stack use;
- `NIM-PITON-TEMP-STK-REQUIREMENT` : upper bound on temporary stack use;
- `COMPUTER-MOVE` : modified Nim state representing an optimal move.

The following subsections present the constraints the specification imposes on the programmer. We describe each constraint informally and then provide the Nqthm term that characterizes it. The conjunction of these terms is the program specification. We have omitted here a precise description of the Nqthm logic and the definitions of several functions used in the constraints which the reader requires in order to understand these terms precisely. Although the straightforward Nqthm logic and suggestive function names convey the intent of these terms, the interested reader is referred to [4] for a description of the Nqthm logic, [8] for the definition of functions related to the Piton interpreter, and [12] for the definition of the specification's other defined functions.

3.1 Algorithm Legality

We require that the function `COMPUTER-MOVE` returns valid Nim moves, which is expressed in the Nqthm logic with the term below. (Note that `COMPUTER-MOVE` is one of the undefined functions being constrained in the specification, and that `GOOD-NON-EMPTY-NIM-STATEP` and `VALID-MOVEP` are defined in [12].)

```
(implies
 (good-non-empty-nim-statep state)
 (valid-movep state (computer-move state)))
```

3.2 Algorithm Optimality

A Nim state consists of a list of numbers that represents the number of stones in each of the piles. A *strategy* maps non-empty Nim states to Nim states in a way consistent with the notion of a legal Nim move. A *winning strategy* is a strategy that guarantees for a particular Nim state that the player who is about to take a turn will win. An *optimal move* for a particular non-empty Nim state is a legal move that is the first step in a winning strategy if one exists.

For any non-empty Nim state, there either is one stone left, or there is a winning strategy for the next player, or there will be a winning strategy for the opponent on his next move. We can therefore search for an optimal move from any Nim state. Figure 3 depicts the search tree for an optimal move from the state `'(1 2 1)`. The nodes in the tree in bold type are *losing states* from which there is no winning strategy.

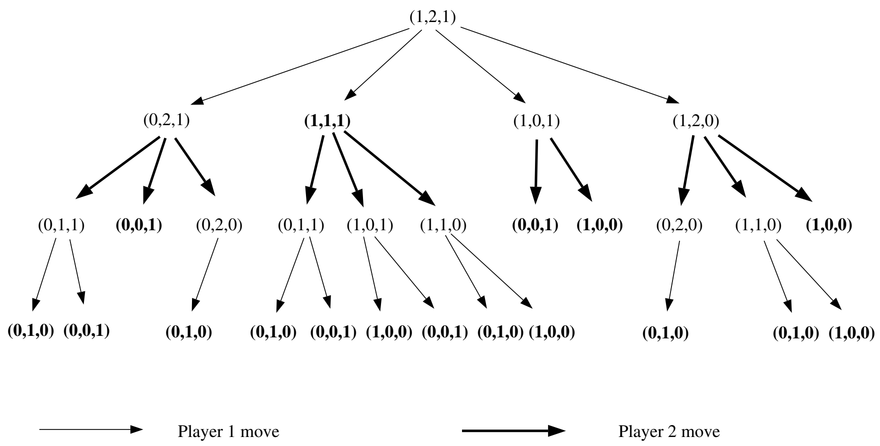

Fig. 3. State-space search for an optimal move

We formalize search by defining an Nqthm function `WSP` as a recursive function that blindly searches all possible moves for an optimal move. `(WSP state)` returns `false` if state is a losing state and an optimal move otherwise. For example, `(WSP '(1 2 1))` returns `'(1 1 1)` and `(WSP '(1 1 1))` returns `false`.

We require that the function `COMPUTER-MOVE` return an optimal move.

```
(implies
 (and
   (good-non-empty-nim-statep state)
   (wsp state))
 (not (wsp (computer-move state))))
```

3.3 Algorithm Implementation

We require that a program that produces the same result as `COMPUTER-MOVE` be implemented in a Piton program. We represent the Nim state by an array of naturals and a length that are passed to the program as parameters. We require that when the Piton subroutine `computer-move` in a list of programs returned by the function `CM-PROG` is executed using the Piton interpreter on a "reasonable" Piton state for `COMPUTER-MOVE-CLOCK` "ticks", the resulting state has an incremented program counter, the program status word set to 'run, and the naturals array representing the Nim state replaced by a new array with the same value as that calculated by `COMPUTER-MOVE`.

```
(implies
 (and
  (equal p0 (p-state pc ctrl-stk (cons wa (cons np (cons s temp-stk)))
             (append (cm-prog word-size) prog-segment) data-segment
             max-ctrl-stk-size max-temp-stk-size word-size 'run))
  (equal (p-current-instruction p0) '(call computer-move))
  (computer-move-implemented-input-conditionp p0))
 (let ((result
        (p p0 (computer-move-clock
                (untag-array (array (car (untag s)) data-segment))
                word-size))))
   (and
    (equal (p-pc result) (add1-addr pc))
    (equal (p-psw result) 'run)
    (equal (untag-array (array (car (untag s)) (p-data-segment result)))
           (computer-move
              (untag-array (array (car (untag s)) data-segment)))))))
```

`COMPUTER-MOVE-IMPLEMENTED-INPUT-CONDITIONP` above identifies "reasonable" Piton states from which we expect our program to work properly, including that `NIM-PITON-CTRL-STK-REQUIREMENTS` bytes be available on the Piton control stack and at least `NIM-PITON-TEMP-STK-REQUIREMENTS` bytes be available on the Piton temporary stack.

3.4 Predictable Response Time

We require that the program return a calculated move within a window of time. The program must execute between 10,000 and 20,000 Piton instructions. We assume the word size is at most of length 32, and the Nim state has no more than 6 piles.

```
(implies (and (nat-listp state ws)
              (lessp 0 ws) (not (lessp 32 ws))
              (lessp 1 (length state)) (not (lessp 6 (length state))))
         (and (lessp 10000 (computer-move-clock state ws))
              (lessp (computer-move-clock state ws) 20000)))
```

Note that this part of the specification eliminates some possible implementations. One is the blind search implementation suggested by the definition of WSP, since the first level of the search tree has as many as $6 * 2^{32}$ nodes, and there are as many as $6 * 2^{32}$ levels to the tree.

3.5 Realistic Memory Use

We require the implementation use little stack space.

```
(lessp (plus (nim-piton-ctrl-stk-requirement)
             (nim-piton-temp-stk-requirement))
       1000)
```

This part of the specification eliminates, for example, a table-driven implementation since there are 2^{177} distinct possible Nim states.

3.6 FM9001 Loadability

We require that the program work on an FM9001 and that it meet the requirements of the compiler correctness proof of [8]. This requires among other things that the compiled Piton programs fit into the FM9001 address space and that the Piton programs be well-formed. The interested reader is referred to [8] for details. This part of the specification allows us to apply the compiler correctness proof.

This constraint also eliminates some possible undesirable implementations. An immensely long program that uses alternation to solve the Nim problem by cases will not fit into the FM9001 address space when compiled, and will therefore not meet the requirements of this part of the specification.

4 The Nim Implementation

In this section we develop an algorithm for efficient calculation of optimal moves, and present a Piton program that implements this algorithm. In Section 5 we discuss the mechanical proof that this implementation meets its specification.

Since a formal specification has been developed for this program as well as a mechanical proof that the program meets the specification, a reader interested only in the behavior of the Nim software and not the topic of verification in general might choose to skip this section. In contrast to conventional program development efforts where the program source code is the only place where an absolutely dependable description of the behavior of the system can be found, in this effort the specification in Section 3 is dependable because the mechanically-generated correctness proof outlined in Section 5 guarantees that it is a correct description of the program's behavior. We present the algorithm because it implements a clever trick and because it makes clearer the wide gap between the specification and the implementation and thus demonstrates the usefulness of proving that the specification is met.

Recall from Section 3 that `(WSP state)` is `false` if and only if no winning strategy exists for state and that this calculation is performed by traversing the tree of all possible legal moves from an initial state. Let `(BIGP state)` = number of piles with at least 2 stones. Let the bit-vector representation of a number be the conventional low-order-bit-first base 2 representation for some large number of bits. Let `(XOR-BVS state)` = bitwise exclusive-or of the bit-vector representations of the number of stones in each pile. Let `0-vector` be a vector composed entirely of zeroes. Let `(GREEN-STATEP state)` be an abbreviation for `(BIGP state)`$>0 \leftrightarrow$ `(XOR-BVS state)`$\neq$`0-vector`.

Theorem: `(GREEN-STATEP state)` $\leftrightarrow$ `(WSP state)`.

This remarkable property was rediscovered for this project, but has in fact been known at least since 1901 [3]. The most obvious proof uses an induction on the search tree. Figure 3 is the search tree for a small Nim state and illustrates the theorem above. As guaranteed by the theorem, the nodes from which there is a winning strategy (in non-bold type) are exactly the nodes that are green states.

We exploit this theorem in the following algorithm that computes optimal moves efficiently. If `(BIGP state)` < 2 and there are an even number of non-empty piles, remove all the stones from a largest pile. If `(BIGP state)` < 2 and there are an odd number of non-empty piles, remove all but one stone from a largest pile. If `(BIGP state)` > 1, find a pile whose binary representation has a 1-bit in the position of the highest 1-bit in `(XOR-BVS state)`, and remove enough stones so that the new pile's binary representation is the `XOR` of the binary representations of the other piles.

From the previous theorem one can prove that this algorithm efficiently generates optimal moves.

The five functions left undefined in the specification presented in Section 3 have been defined in the implementation. The Piton implementation of Nim (returned by function `CM-PROG` described in the specification) implements the efficient algorithm described above and contains approximately 300 lines of Piton code. Figure 4 lists two example Piton routines from the implementation that convert natural numbers to bit vectors and natural number arrays to bit vector arrays. [12] contains a complete program listing.

5 The Nqthm Correctness Proof

Nqthm has been used to construct proofs that correspond to the six specification constraints given in Section 3, so our Piton implementation has been proved correct. Approximately 750 lemmas were proved in order to guide Nqthm to the proof of these main theorems. Most of these lemmas fall into one of the following four categories.

- Some lemmas relate the behavior of Piton loops and subroutines when interpreted by the Piton interpreter to an Nqthm function. Typically, a special Nqthm function that calculates a result in precisely the same manner as the

```
     SUBROUTINE NAT-TO-BV
     (VALUE) (CURRENT-BIT TEMP)
       CALL PUSH-1-VECTOR
       POP-LOCAL CURRENT-BIT
       CALL PUSH-1-VECTOR
       RSH-BITV
LOOP:  PUSH-LOCAL VALUE
       TEST-NAT-AND-JUMP ZERO DON
       PUSH-LOCAL VALUE
       DIV2-NAT
       POP-LOCAL TEMP
       POP-LOCAL VALUE
       PUSH-LOCAL TEMP
       TEST-NAT-AND-JUMP ZERO LAB
       PUSH-LOCAL CURRENT-BIT
       XOR-BITV
LAB:   PUSH-LOCAL CURRENT-BIT
       LSH-BITV
       POP-LOCAL CURRENT-BIT
       JUMP LOOP
DONE:  RET
```

```
     SUBROUTINE NAT-TO-BV-LIST
     (NAT-LIST BV-LIST LENGTH) (I 0)
LOOP:  PUSH-LOCAL NAT-LIST
       FETCH
       CALL NAT-TO-BV
       PUSH-LOCAL BV-LIST
       DEPOSIT
       PUSH-LOCAL I
       ADD1-NAT
       SET-LOCAL I
       PUSH-LOCAL LENGTH
       EQ
       TEST-BOOL-AND-JUMP T DONE
       PUSH-LOCAL NAT-LIST
       PUSH-CONSTANT (NAT 1)
       ADD-ADDR
       POP-LOCAL NAT-LIST
       PUSH-LOCAL BV-LIST
       PUSH-CONSTANT (NAT 1)
       ADD-ADDR
       POP-LOCAL BV-LIST
       JUMP LOOP
DONE:  RET
```

Fig. 4. Two example Piton subroutines

Piton program in question is defined. A clock function that computes the number of Piton instructions the loop or subroutine will execute is defined. A correctness lemma for a loop or subroutine states that the Piton interpreter running the loop or subroutine for the number of instructions computed by the clock function yields the same result as the Nqthm function.
We call proofs of this kind of lemma *code correctness proofs.*

- Some lemmas demonstrate the equivalence of Nqthm functions that mimic Piton programs to Nqthm functions defined more naturally and are easier to reason about.
  We call proofs of this kind of lemma *specification proofs.*
- Some lemmas are used to prove the optimality of `COMPUTER-MOVE`. That is, that the algorithm outlined in Section 4 works.
  We call proofs of this kind of lemma *algorithm proofs.*
- Some lemmas establish bounds on the clock functions.
  We call proofs of this kind of lemma *timing proofs.*

The timing proofs were done using PC-Nqthm [7], the interactive enhancement to Nqthm. All other proofs require only Nqthm. The proof that the Piton program meets the specification uses the default arithmetic library [10] and the Piton interpreter definitions [8].

```
(prove-lemma correctness-of-nat-to-bv (rewrite)
        (implies
         (and
          (equal p0 (p-state pc ctrl-stk (cons v temp-stk) prog-segment
                     data-segment max-ctrl-stk-size max-temp-stk-size
                     word-size 'run))
          (equal (p-current-instruction p0) '(call nat-to-bv))
          (nat-to-bv-input-conditionp p0))
         (equal
          (p p0 (nat-to-bv-clock num))
          (p-state
            (add1-addr pc) ctrl-stk
            (cons (list 'bitv (nat-to-bv (cadr v) word-size)) temp-stk)
            prog-segment data-segment max-ctrl-stk-size
            max-temp-stk-size word-size 'run))))
```

Fig. 5. An Nqthm prove-lemma event

The correctness of `nat-to-bv` listed in Figure 4 is described by the prove-lemma command in Figure 5. We use the proof of this lemma to illustrate what it takes to do these kinds of proofs. First, we accomplish a code correctness proof the loop in `nat-to-bv`. We introduce a recursive function `FOO1` in the Nqthm logic that computes the values calculated in the loop, and use Nqthm to prove inductively that the interpreter applied to a Piton state that is about to execute that loop calculates the same values as `FOO`. Several trivial subsidiary lemmas must be proved first so as to guide Nqthm to this proof. We then introduce a function `FOO2` that computes values in the same manner as `nat-to-bv`, and use Nqthm to generate a code correctness proof of this. Next we have Nqthm generate a specification proof of the equivalence of `FOO2` to `NAT-TO-BV`, a conventional Nqthm definition of a natural number to bit vector conversion function. This requires dozens of subsidiary lemmas about arithmetic. Finally, we use Nqthm to prove the lemma shown in Figure 5, which it does using the previously proved lemmas.

The correctness lemma is even more complex than it may appear at first glance. `NAT-TO-BV-INPUT-CONDITIONP` contains additional preconditions that this subprogram requires in order to run correctly. There must be enough stack space to do the calculation, the value at the top of the stack must be a natural number representable on the machine, and the program segment must have the needed programs loaded. `NAT-TO-BV-CLOCK` is a function that computes the number of instructions the Piton subroutine `nat-to-bv` executes when called.

Like most conjectures presented to Nqthm with the prove-lemma command for which Nqthm successfully generates a proof, `CORRECTNESS-OF-NAT-TO-BV` has two important properties. First, since it is accepted by the Nqthm prover, we believe that it represents mathematical truth. Second, it is an instruction to the prover about how to prove future theorems. By constructing the exact form of the theorem mindful of the theorem's interpretation in later proof efforts, we add to the prover's ability to reason soundly about Piton programs that call `nat-to-bv`.

`CORRECTNESS-OF-NAT-TO-BV` is useful because it equates the behavior of a Piton subprogram as defined by the Piton interpreter to an Nqthm term that does not involve the interpreter. By applying this lemma Nqthm can reason about this program without regard to the semantics of Piton. This makes proofs achievable since, as a practical matter, proofs involving Piton programs running on the very complicated Piton interpreter are much more complex than proofs about Nqthm functions that compute similar results. The lemma is stored in Nqthm as a replacement rule, and has been constructed so that later proofs can apply this lemma automatically during proofs. The subroutine `nat-to-bv-list` contains several kinds of Piton instructions, and the proof of the correctness of `nat-to-bv-list` depends on the semantics of these instructions as defined in the Piton interpreter. For example, `PUSH-LOCAL` is defined in the Piton interpreter and the Nqthm theorem prover uses the definitions that describe the effect of executing a `PUSH-LOCAL` instruction automatically when constructing a proof. Similarly, the instruction `CALL NAT-TO-BV` in the subprogram is reasoned about by Nqthm automatically using `CORRECTNESS-OF-NAT-TO-BV`.

Once a carefully-constructed correctness theorem such as this one about a subroutine has been added to the database of proved lemmas, a call to that subroutine in a Piton program is reasoned about automatically just as any built-in Piton instruction.

The development of the correctness proof required about 3 man-months. (This does not include time to develop the events of the Piton compiler or arithmetic library or to accomplish an earlier proof related to Nim [11].) Approximately 40% of the man-hours were spent on code correctness proofs, 30% on specification proofs, 20% on algorithm proofs, and 10% on timing proofs. Nqthm generates the proof in approximately 10 hours on a Sun Sparcstation IPC.

6 Conclusions

Mechanical verification of programs in this manner is time-consuming and difficult. Nevertheless, and quite remarkably, the experience of building the Nim program suggests that development time scales linearly with program length. Once a Piton subroutine has been proved correct, a call to this subroutine can be reasoned about as easily as any basic Piton statement.

A modest but non-trivial application has been constructed that makes use of a verified compiler and microprocessor. Its functional correctness has been verified using Nqthm. Mechanically-checked proofs of bounds on the number of executed instructions have been constructed. We hope to verify more complex software in the future using our mechanical proof tools as we pursue our goal of building error-free computer systems. Programs that work in real-time are particularly attractive since the formalization of a programming language with an interpreter as in [8] is well-suited to proving program timing properties.

An FM9001 was fabricated and runs a compiled version of the Nim program. The fabricated FM9001 microprocessor, the Piton compiler, and the Nim program were never tested in a conventional manner during development or after completion. Even so, each worked the first time and we would have been surprised if any had not.

References

1. Ken Albin, Warren Hunt, and Matthew Wilding. FM9001 fabrication (in preparation). Technical report, Computational Logic, Inc., 1993.
2. William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
3. Charles L. Bouton. Nim, a game with a complete mathematical theory. In *Annals of Mathematics*, volume 3, 1901-02.
4. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
5. Martin Gardner. *Mathematical Puzzles and Diversions*. Simon and Schuster, New York, 1959.
6. Warren A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.
7. Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 60, Institute for Computing Science, University of Texas at Austin, Austin, Texas, August 1987.
8. J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.
9. Warren A. Hunt Jr. and Bishop Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, April 1992.
10. Matthew Wilding. Proving Matijasevich's lemma with a default arithmetic strategy. *Journal of Automated Reasoning*, 7(3), September 1991.
11. Matthew Wilding. A verified nim strategy. Internal Note 249, Computational Logic, Inc., November 1991.
12. Matthew Wilding. A proved application with simple real-time properties. Technical Report 78, Computational Logic, Inc., October 1992.

Acknowledgments: I thank Bill Bevier and J Moore for many very valuable suggestions during this project, and David Goldschlag, Chris Rath, Bill Young, and three anonymous referees for reading drafts of this paper. This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.