A Robust Machine Code Proof Framework for Highly Secure Applications

David S. Hardin
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA USA
dshardin@rockwellcollins.com

Eric W. Smith
Department of Computer Science
Stanford University
Palo Alto, CA USA
ewsmith@cs.stanford.edu

William D. Young
Department of Computer Sciences
University of Texas at Austin
Austin, TX USA
byoung@cs.utexas.edu

ABSTRACT

Security-critical applications at the highest Evaluation Assurance Levels (EAL) require formal proofs of correctness in order to achieve certification. To support secure application development at the highest EALs, we have developed techniques to largely automate the process of producing proofs of correctness of machine code. As part of the Secure, High-Assurance Development Environment program, we have produced in ACL2 an executable formal model of the Rockwell Collins AAMP7G microprocessor at the instruction set level, in order to facilitate proofs of correctness about that processor's machine code. The AAMP7G, currently in use in Rockwell Collins secure system products, supports strict time and space partitioning in hardware, and has received a U.S. National Security Agency (NSA) Multiple Independent Levels of Security (MILS) certificate based in part on a formal proof of correctness of its separation kernel microcode. Proofs of correctness of AAMP7G machine code are accomplished using the method of "compositional cutpoints", which requires neither traditional clock functions nor a Verification Condition Generator (VCG). In this paper, we will summarize the AAMP7G architecture, detail our ACL2 model of the processor, and describe our development of the compositional cutpoint method into a robust machine code proof framework.

Categories and Subject Descriptors

B.1.2 [**Control Structures and Microprogramming**]: Control Structure Performance Analysis and Design Aids – *formal models, simulation*; D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs, formal methods, reliability*.

General Terms

Reliability, Security, Verification.

Keywords

ACL2, high-assurance, certification, cryptography, processor modeling, symbolic simulation, theorem proving.

ACL2'06, August 15-16, 2006, Seattle, Washington, USA.

1. INTRODUCTION

Security-critical applications developed for use by the U.S. government must be certified according to the Common Criteria at high Evaluation Assurance Levels [2]. At the highest EAL, EAL7, the application must be formally specified, and the application must be proven to implement its specification. This can be a very expensive and time-consuming process. One of the main research goals of the Secure, High-Assurance Development Environment (SHADE) program is to improve secure system evaluation -- measured in terms of completeness, human effort required, time, and cost -- through the use of highly automated formal methods. In support of this goal, we have developed practical techniques for creating executable formal computing platform models that can both be proved correct, and also function as high-speed simulators [4], [7]. This allows us to both verify the correctness of the models, as well as validate that the formalizations accurately model what was actually designed and built.

In this paper, we will present a code proof framework for highly secure applications targeting the Rockwell Collins AAMP7G embedded microprocessor [1], [13], built upon an executable formal instruction set model of the AAMP7G written in ACL2 [8]. The AAMP7G is of particular interest because it supports strict time and space partitioning in hardware, and has received an NSA MILS certificate to handle Unclassified through Top Secret codeword information concurrently, based in part on a formal proof of correctness of its separation kernel microcode.

We will begin by describing the formally verified partitioning features of the AAMP7G. We will introduce the AAMP7G instruction level model, and describe how it has been validated through the execution of AAMP binaries. The model has also been informally validated by using the same user interface to control the AAMP7G model that is also used to control the actual AAMP7G chip. We will then summarize the compositional cutpoint code proof technique, and show how it is coupled with ACL2's efficient symbolic simulation support to produce a robust code proof framework for highly secure applications that are compiled to AAMP7G machine code. This secure application code can be generated from a number of sources, including traditional compilers, but also including a certifying compiler for the µCryptol cryptographic programming language [14]. The certifying compiler for µCryptol generates correctness statements for intermediate transformations of the compiler that are then checked automatically by a theorem prover [11].

2. THE AAMP7G

The AAMP7G is the latest in the line of Collins Adaptive Processing System (CAPS) processors and AAMP microprocessors developed by Rockwell Collins, Inc. (RCI) for use in military and civil avionics since the early 1970s [1]. Over the years, RCI has been able to tailor each implementation to embedded avionics and communication product requirements, accruing size, weight, power, cost, and specialized feature advantages over alternate solutions. Each new AAMP makes use of the same multi-tasking stack-based instruction set, while adding state of the art technology in the design of each new CPU and peripheral set. AAMP7G adds built-in partitioning technology among other improvements.

AAMP processors feature a stack-based architecture with 32-bit segmented, as well as linear, addressing. AAMP supports 16/32-bit integer and fractional, as well as 32/48-bit floating point operands. The lack of user-visible registers improves code density (many instructions are a single byte), which is significant in embedded applications where code typically executes directly from slow Read-Only Memory. AAMP provides a unified call and operand stack, and the architecture defines both user and executive modes, with separate stacks for each user “thread”, as well as a separate stack for executive mode operation. The transition from user to executive mode occurs via traps; these traps may be programmed, or may occur as the result of erroneous execution (illegal instruction, stack overflow, etc.). The AAMP architecture also provides for exception handlers that are automatically invoked in the context of the current stack for certain computational errors (divide by zero, arithmetic overflow). The AAMP instruction set is of the CISC variety, with over 200 instructions, supporting a rich set of memory data types and addressing modes.

2.1 AAMP7G Intrinsic Partitioning

The transition from multiple CPUs to a single multi-function CPU is shown in Figure 1. On the left, three federated processors provide three separate functions, A, B, and C. It is straightforward to show that these three functions have no unintended interaction.

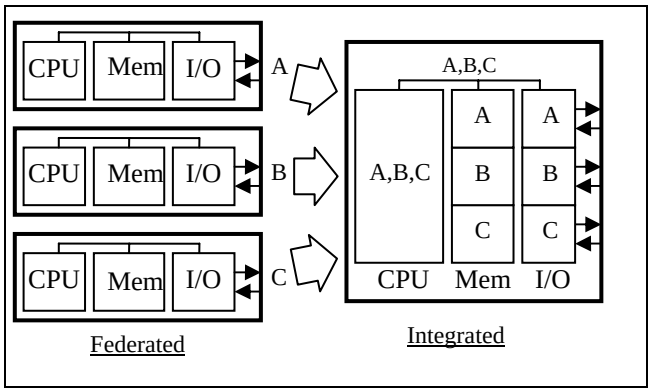

Figure 1. Transition to Multi-Function CPU.

On the right of Figure 1, an integrated processor provides for all three functions. The processor performs code from A, B, and C; its memory contains all data and I/O for A, B, and C. A partition is a container for each function on a multi-function partitioned CPU like the AAMP7G. AAMP7G follows two rules to ensure partition independence:

1. TIME PARTITIONING. Each partition must be guaranteed a time slice to execute the intended function.
2. SPACE PARTITIONING. Each partition must have exclusive-use space for storage.

2.1.1 Time Partitioning

Each partition must be guaranteed a time slice to execute the intended function. The AAMP7G uses strict time partitioning to ensure this requirement. Each partition is allotted certain time-slices during which time the active function has exclusive use and control of the CPU and related hardware.

For the most secure systems, time slices are allocated at system design time and not allowed to change. For dynamic reconfiguration, a "privileged" partition may be allowed to set time slices. AAMP7G supports both of these cases, as determined by the system designer.

The asynchronous nature of interrupts poses interesting challenges for time partitioned systems. AAMP7G has partition-aware interrupt capture logic. Each interrupt is assigned to a partition; the interrupt is only recognized during its partition's time slice. Of course, multiple interrupts may be assigned to a partition. In addition, an interrupt may be shared by more than one partition if needed.

System-wide interrupts, like power loss imminent or tamper detect, also need to be addressed in a partitioned processor. In these cases, AAMP7G will suspend current execution, abandon the current list of partition control, and start up a list of partition interrupt handlers. Each partition's interrupt handler will run performing finalization or zeroization as required by the application.

2.1.2 Space Partitioning

Each partition must have exclusive-use space for storage. The AAMP7G uses memory management to enforce space partitioning. Each partition is assigned areas in memory that it may access. Each data and code transfer for that partition is checked to see if the address of the transfer is legal for the current partition. If the transfer is legal, it is allowed to complete posthaste. If the transfer is not legal, the AAMP7G Partition Management Unit (PMU) disallows the CPU from seeing read data or code fetch data; the PMU also preempts write control to the addressed memory device.

Memory address ranges may overlap, in order to enable inter-partition communication. In this case, interaction between partitions is allowed since it is intended by system design. For maximum partition independence, overlapping access ranges should be kept to a minimum.

As with time slices, memory ranges may be allocated at system design time and not allowed to change. Or, for dynamic reconfiguration, a "privileged" partition may be allowed to set memory ranges. AAMP7G supports both of these cases, as determined by the system designer.

2.2 Partition Control

Only a small amount of data space is needed for partition control structures. The data space is typically not intended to be included in any partition's memory access ranges. Each partition's control includes time allotment, memory space rights, and initial state, stored in ROM. Each partition's saved state is stored in RAM. Partition control blocks are linked together defining a partition activation schedule. AAMP7G partition initialization and partition switching are defined entirely by these structures and performed entirely in microcode. Thus, no software access is needed for AAMP7G partitioning structures. This limits verification of AAMP7G partitioning to proving that the partitioning microcode performs the expected function and no other microcode accesses the partitioning structures.

3. AAMP7G PARTITIONING MICROCODE PROOFS

Rockwell Collins has performed a formal verification of the AAMP7G partitioning system using the ACL2 theorem prover. This work was part of an evaluation effort which led the AAMP7G to receive a MILS Certificate from NSA in May 2005, enabling a single AAMP7G to concurrently process Unclassified through Top Secret codeword information. We first established a formal security policy, as described in [6]. We produced an abstract model of the AAMP7G's partitioning system, as well as a low-level model that directly corresponded to the AAMP7G microcode. We used ACL2 to automatically produce the following:

1. Proofs validating the security model
2. Proof that the abstract model enforces the security policy
3. Proof that the low-level model corresponds to the abstract model.

Richards et al. [12] discuss the use of ACL2 to meet high-assurance Common Criteria requirements. One interpretation of the requirement for low-level design models is that the low-level design model be sufficiently detailed and concrete so that an implementation can be derived from them with no further design decisions. Because there are no design decisions remaining, one can easily validate the model against the implementation. In the case of the AAMP7G partitioning microcode, this validation was provided in the form of a code-to-spec review conducted by NSA evaluators. Note that this low level of abstraction of a model, while making the code-to-spec review process easier, makes proofs about it more challenging.

4. AAMP7G INSTRUCTION SET MODEL

Having established the correctness of the AAMP7G's partitioning system, we next wished to provide a formal model of the instruction set processing that occurs within a partition's time slice. Having such a model would enable us to perform machine code proofs of correctness that could be used in high-assurance evaluations. The instruction set model and all the necessary support books consists of some 100,000 lines of ACL2 code. The architecture of the AAMP7G instruction set model is shown in context in Figure 2. The layers identified in italics in Figure 2 are those that we explicitly model. We begin with a concrete instruction model, written in a sequential manner that reflects how the machine actually operates. The AAMP memory model is based on the linear address space book previously used in the AAMP7G partitioning proofs [5], which in turn is built on a bags library described in [16]. The AAMP7G machine state, including the architecturally-defined registers, is represented as an ACL2 single-threaded object (stobj) for performance reasons.

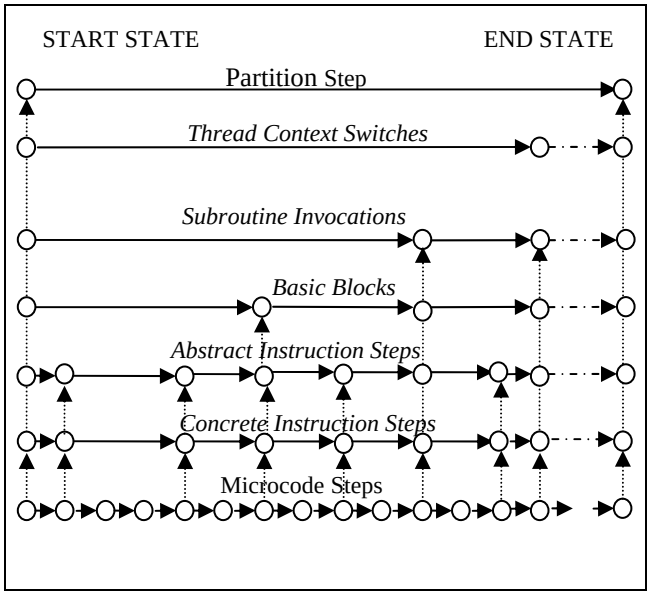

Figure 2. AAMP7G instruction set model architecture.

We prove correspondence between the concrete model and a more abstract model, which is described in detail below. Sequences of abstract instruction steps form basic blocks; a machine code subroutine is made up of a collection of basic blocks. Subroutine invocations are performed in the context of an AAMP thread, and multiple user threads plus the executive mode constitute an AAMP7G partition. Our model supports the entire context switching machinery defined by the AAMP architecture, including traps, outer procedure returns, executive mode error handlers, and so on.

4.1 Running AAMP7G Machine Code

Since we model the AAMP7G instruction set in its entirety, we can analyze AAMP7G machine code from any source. This includes traditional compilers and assemblers, but also notably includes a certifying compiler for the μCryptol cryptographic programming language, also developed under the SHADE program. Since we directly model memory, we merely translate the binary file for a given AAMP7G machine code program into a list of (address, data) pairs that can be loaded into ACL2. We load the code, reset the model, and the execution of the machine code then proceeds, under the control of an Eclipse-based [3] user interface that was originally written to control the actual AAMP7G, as shown in Figure 3.

4.2 Model Validation

We validate the AAMP7G instruction set model by executing the same instruction set diagnostics on the model that are used for AAMP processor acceptance testing. A typical diagnostic exercises each instruction, plus context switching, exception

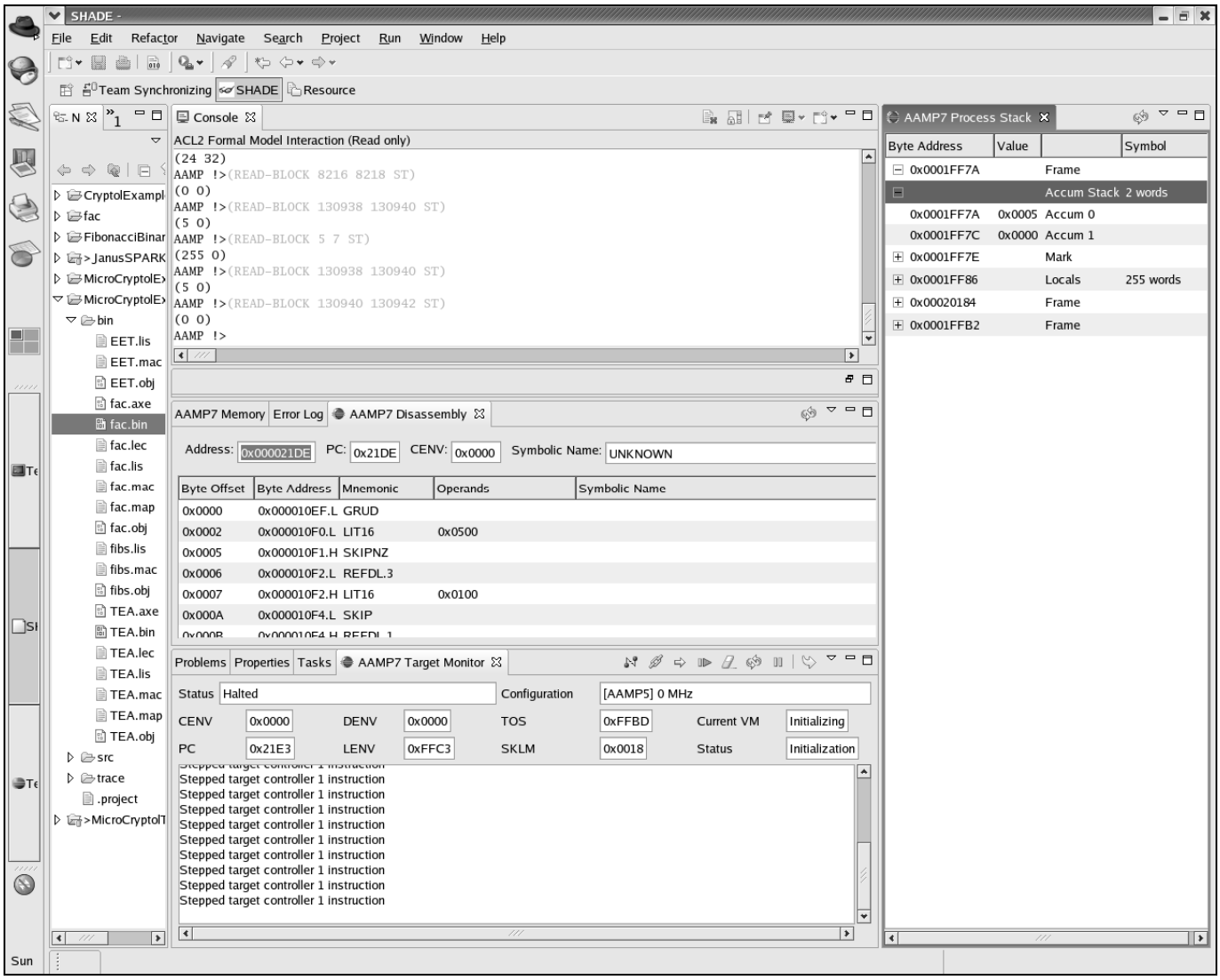

Figure 3. Eclipse-based user interface to ACL2 model of the AAMP7G.

handling, etc., and can run to many thousands of AAMP7G instructions each.

4.3 Instruction Abstraction and Symbolic Simulation

The goal of proving properties of AAMP7G programs requires an ACL2 formalization of the AAMP7G operation semantics amenable to formal analysis. When we began this work, the available formalization had been adapted from the model of the Rockwell AAMP5 and was designed to be executable and faithful to the operation-level semantics. However, it was far from friendly to formal analysis [17].

In particular, the semantics is defined in terms of a complex macro mechanism that embeds in ACL2 a simple assembly like language. On its face, this notation provides an extremely perspicuous characterization of the semantics. For example, below is the definition of the ADDU instruction, which takes two 16-bit unsigned values from the top of the stack, adds them using modular unsigned integer arithmetic, and pushes the result back onto the stack.

```
(AAMP *state->state*
        (POP ux)
        (POP uy)
        (PUSH (UWord16 (+ ux uy)))
        st)
```

Here, `AAMP` is a macro defined within ACL2 that interprets its arguments as follows: The first argument specifies that this function is a state to state (as opposed to a value-returning) transformation. The effect on the state is equivalent to executing the listed pseudo-instructions in sequence. Local variables such as `ux` and `uy` are introduced where needed. Finally, the resulting state is returned.

This form belies the underlying complexity of the model. For example, the semantics must check that any of several anomalous conditions relating to resets, traps, interrupts, and memory violations have not been signaled in the current state; and that the AAMP7G Partition Management Unit (PMU) is configured to allow fetching the current code byte, reading the top two locations on the stack, and writing the second location on the stack.

Each potential "error" leads to a different state being returned. In addition, the semantics assumes that there is no overlap between the code and data segments referenced by the operation. The simple expression above macro-expands into a complex Lisp expression of around 125 lines (or over 1000 lines when the nested LET structure is fully expanded). ADDU is a particularly simple operation, not involving any of the exception conditions that arise, for example, with signed integer addition. Certain AAMP7G operations expand into expressions that are literally thousands of lines of almost incomprehensible Lisp code.

Moreover, because ACL2 is an applicative language, expansion of the macro emulates this imperative notation by translating it into an applicative form. The result is a set of nested updates to a system state stobj with some two dozen fields. Because the macro emulates sequential computation, many of these updates are redundant, cumulative, or offsetting.

We manage this complexity in several steps:

We identify a set of preconditions adequate to eliminate most execution paths corresponding to anomalous conditions not expected to arise in a typical execution.

We rewrite the nest of state updates to reorder them, combine multiple updates to the same state component, and eliminate redundant or offsetting updates.

After rewriting has stabilized, we enable a new set of rewrite rules that replace commonly occurring locutions by more intuitive abstractions. For example, the expression

```
(read-data-word (nth *aamp.denvr* st)
                (+ 3 (nth *aamp.tos* st))
                (nth *aamp.ram* st))
```

rewrites to

```
(get-stack-word 3 st).
```

Finally, we package the resulting state into a very readable macro form, which is merely syntactic sugar for a set of updates on the state.

The completed abstraction of the ADDU instruction is given below:

```
(defun op-addu-preconditions (st)
  (and
    (standard-preconditions st)
    (allowed-to-fetch-current-code-byte st)
    (allowed-to-read-top-n-stack-words 2 st)
    (allowed-to-write-second-word-on-
      stack st)))
```

```
(defun vm-addu-expected-result (st)
  (modify st
    :pc (inc-pc 1 st)
    :tos (inc-tos 1 st)
    :memtmp8 *addu-opcode*
    :memtmp (get-stack-word 1 st)
    :ram (modify-ram st
                  :stack-word 1
                  (+ (get-stack-word 0 st)
                     (get-stack-word 1 st))
         )))

(defthm vm-addu-rewrite
  (implies
    (equal (current-op st) *addu-opcode*)
    (equal (aamp_vm st)
           (if (op-addu-preconditions st)
               (vm-addu-expected-result st)
             (aamp_vm-you-shouldnt-see-this
               st)))))
```

The function OP-ADDU-PRECONDITIONS collects those conditions that need to hold for "normal" execution of the ADDU operation. VM-ADDU-EXPECTED-RESULT gives the expected output state as a modification of the input state. Unlike the earlier 'assembly' language form, this doesn't characterize the *steps* of the computation, but rather the *result* in a very compact and readable form.

Finally, proving the theorem VM-ADDU-REWRITE establishes that stepping the AAMP virtual machine on this instruction will yield exactly the expected result, assuming the preconditions are satisfied. If not, the result is characterized by AAMP_VM-YOU-SHOULDNT-SEE-THIS, which is defined to be exactly AAMP_VM. That is, in all non-normal cases the operation simply does whatever it does, but we don't need to look at it. If the instruction semantics involves interesting exceptions, such as overflow or divide-by-zero, these are characterized by additional expected result functions and additional branches in the right hand side of this rewrite rule.

We have treated each of 192 instructions from the AAMP7G processor model similarly. Assuming that we can relieve the preconditions at each step, this allows us efficiently to symbolically step through even very long sequences of AAMP7G instructions. After each step, the rewriter effectively canonicalizes the result into a very compact and readable form.

5. COMPOSITIONAL CODE PROOF INFRASTRUCTURE

Our verification of AAMP7G programs is done compositionally. That is, we verify programs one subroutine at a time. We try to ensure that, after we verify a subroutine, we never have to analyze it again. Thus, the correctness theorem (or theorems) for a routine R must be strong enough to support the verification of any routine that calls R, without the need to analyze R again.

Before we verify a subroutine R, we must verify all of the routines that R calls. Thus, the order in which we verify a

program's subroutines is a topological order on its call graph. Clearly, this scheme does not work for recursive routines, so we handle recursion specially. (A further complication arises in the case of mutual recursion, which we do not yet handle. The solution will be to verify one mutually-recursive clique of routines at time.)

To prove the correctness theorem for a subroutine we use a proof methodology called "compositional cutpoints." Our method borrows parts of the method put forth in [9]; both methods are improvements of an earlier method described by those same authors.

Cutpoint proofs require annotating the subroutine to be verified by placing assertions at some of its program locations; those locations are called "cutpoints". Every cutpoint has a corresponding assertion which is taken to apply to those states that arise just before the instruction at the cutpoint is executed.

Consider the AAMP7G subroutine FACT-ITER (see Figure 4), which iteratively computes the low 32 bits of the factorial of its argument. We verify a subroutine F by giving it a precondition and a postcondition. Roughly speaking, the correctness theorem for a subroutine F will say:

"If we are about to start executing F, and if the precondition holds, then if F eventually returns, the post-condition will be true upon return from F."

This is a partial correctness result, because it has a hypothesis that asserts that F eventually returns. We could extend the partial correctness result to a total correctness result by proving separately that F terminates, and we believe such a termination proof can be done using a "compositional cutpoints" method similar to the one we use for partial correctness. However, we have yet to implement the proof machinery for termination (this is current work in progress). In the rest of this paper we will consider only partial correctness.

The precondition of a subroutine R applies to its "prestate," that is, the state just after a frame was pushed onto the call stack on behalf of R and just before the first instruction of R is executed.

Among the assertions typically found in the precondition of routine R are:

1. that the expected code for R is indeed loaded at the correct address.
2. that the arguments to R satisfy some constraints (e.g., "argument x is a positive integer")
3. that the machine is in a "normal" state (e.g., no memory access violations have occurred)
4. that there is enough space on the call stack for any frames that will be pushed by R or its callees.
5. that the AAMP7G Partition Management Unit (PMU) is setup to give permission for all the memory accesses performed by R and its callees.

This list is not exhaustive. The precondition for FACT-ITER is given in Figure 5.

```
    #x04           ;; Proc Header --
    #x00           ;; 4 words of locals
;
    #x10            ;; LIT4 0
    #x11            ;; LIT4 1
; local0 is a counter from 1 up to N
    #xc0            ;; ASNDL 0
; local2 is initialized to 1
    #x10            ;; LIT4 0
    #x11            ;; LIT4 1
    #xc2            ;; ASNDL 2
; L2: loop top ------------------- CUTPOINT
    #x30            ;; REFDL 0
    #x34            ;; REFDL 4
; if local0 > N, goto L
    #xa5
    #x0e            ;; GRUD
    #x5b            ;; SKIPNZI
    #x0e            ;;  L (+14)
; local2 = local2 * local0
    #x30            ;; REFDL 0
    #x32            ;; REFDL 2
    #xa5
    #x2a            ;; MPYUD
    #xc2            ;; ASNDL 2
; increment local0
    #x30            ;; REFDL 0
    #x10            ;; LIT4 0
    #x11            ;; LIT4 1
    #xa5
    #x28            ;; ADDUD
    #xc0            ;; ASNDL 0
; go to L2
    #x19            ;; LIT8N
    #x13            ;;  L2 (-20)
    #x59            ;; SKIP
; L: return local2
    #x32            ;; REFDL 2
    #x16            ;; LIT4 6
    #x5f            ;; RETURN
```

Figure 4. AAMP7G machine code for 32-bit unsigned factorial.

The postcondition of a routine applies to its "poststate," that is, the state that results when the stack frame for the routine is popped off at the end of the routine's execution. Typically a postcondition will specify the value returned by the routine (e.g,

```
(defun fact-iter-max-words-of-operand-stack () (declare (xargs :guard t)) 4)
;from analysis of the code

(defund fact-iter-precondition (s)
    (declare (xargs :non-executable t))
    (and (standard-precondition (fact-iter-address)
                                (fact-iter-code)
                                (fact-iter-max-words-of-operand-stack)
                                s)
         ;; The routine doesn't work if the argument is the maximum 32-bit
         ;; unsigned value, since in that case the loop never terminates:
         (not (equal 4294967295 (aamp::read-two-local-words 4 s)))))
```

Figure 5. Precondition specification for factorial subroutine.

that the routine returns the low 32 bits of the factorial of the input). However, specifying the return value alone is rarely sufficient. Recall that the correctness theorem for a routine R must be sufficient to verify any routine that calls R. Often R's callers will rely on many properties of R in addition to its return value. For example, R's callers will typically need to know that R doesn't modify memory that it shouldn't, and that R doesn't cause any traps or memory access violations.

Since we don't want to re-analyze R again after verifying it, R's correctness theorem must capture these "frame conditions". Writing a predicate that explicitly lists all of the frame conditions for R (including all the state components and memory ranges that R leaves unchanged) can be tedious. Instead we phrase R's postcondition using an equality; we specify that the poststate is equal to some modification of the initial state. The modification can be phrased using the MODIFY macro. The modification typically involves popping off a stack frame, pushing on a return value, and perhaps making changes to memory. This equality formulation is quite strong; for the equality to hold, any state component not explicitly mentioned in the call to MODIFY must remain unchanged. In keeping with this approach, the user usually supplies a postcondition that includes a "poststate". The poststate for FACT-ITER is given in Figure 6.

A typical correctness theorem for a subroutine F is:

```
(defthm f-correct
  (implies
    (and
      (f-precondition s0)
      (equal (program-counter s0)
             (f-starting-program-counter))
      (eventually-returns s0))
    (equal (run-until-return s0)
           (f-poststate s0))))
```

Here, F-STARTING-PROGRAM-COUNTER indicates the location of the beginning of the code for F. F-CORRECT corresponds to the rough English-language correctness specification given above.

One objection to requiring a poststate rather than a postcondition is that a poststate is "too strong". Perhaps there are parts of the poststate that the user doesn't care to specify. For example, FACT-ITER changes the AAMP's "memory temporary" registers and leaves “garbage” above the stack pointer. No caller of FACT-ITER will rely on any of those values, and it would be tedious to specify them exactly as a function of FACT-ITER’s prestate. The solution is to characterize the poststate in terms of the prestate and the poststate itself. This seems counter-intuitive but allows one to easily make trivially-true characterizations of unimportant state components.

For example, we specify the MEMTMP component of FACT-ITER's poststate by just saying that it equals the MEMTMP component of FACT-ITER's poststate! This allows us to enjoy the strength of phrasing postconditions in terms of equality with a poststate but without the tedium of specifying uninteresting components. We call this approach "wormhole abstraction" or "don't care specification".

With wormhole abstraction, the correctness theorem for a subroutine F is:

```
(defthm f-correct
  (implies
    (and
      (f-precondition s0)
      (equal (program-counter s0)
             (f-starting-program-counter))
      (eventually-returns s0))
    (equal
      (run-until-return s0)
      (f-poststate s0
        (run-until-return-wormhole s0)))))
```

Note that F-POSTSTATE now takes two arguments, the initial state, S0, and `(run-until-return-wormhole s0)`. Here RUN-UNTIL-RETURN-WORMHOLE is just RUN-UNTIL-RETURN, renamed to prevent F-CORRECT from causing a rewrite loop.

```
;; Factorial, defined in the traditional recursive style
(defun fact (n)
  (if (zp n) 1
    (* n (fact (1- n)))))


(defun fact-iter-words-of-locals-and-args () (declare (xargs :guard t)) 6)
;from dealloc count pushed just before return

(defun fact-iter-words-of-return-values () (declare (xargs :guard t)) 2)
;from height of operand stack just before return


(defun fact-iter-poststate (s0 s)
    (declare (xargs :non-executable t))
    (standard-poststate ((0 ;; top return value
                            2 ;; takes up 2 words
                            ;;the mathematical factorial of the argument:
                            (fact (gacc::read-data-words 2 (aamp::aamp.denvr s0)
                                                           (+ 4 (aamp::aamp.lenv s0))
                                                           (aamp::aamp.ram s0)))
                            ))
                         (fact-iter-max-words-of-operand-stack)
                         (fact-iter-words-of-locals-and-args)
                         (fact-iter-words-of-return-values)
                         s0
                         s))
```

Figure 6. Poststate for factorial subroutine.

Figure 7 shows the main command invoked as part of the proof of the correctness theorem for FACT-ITER, FACT-ITER-CORRECT. The prove-it macro causes ACL2 to attempt a "cutpoint to cutpoint" proof for the subroutine and then appeals to a generic result that states that the "cutpoint to cutpoint" property is sufficient to ensure partial correctness. The remainder of this section describes the process in more detail.

The prove-it macro automatically considers a routine's starting program location to be a cutpoint, with the routine's precondition as its corresponding assertion. In addition the user can specify a set of "user cutpoints," each paired with a corresponding assertion. User cutpoints often correspond to the continuation tests of loops.

The resulting full set of cutpoints is sufficient if it "cuts every loop," that is, if every cycle in the routine's control flow graph contains a cutpoint. We need not consider cycles in the code of called subroutines; any subroutine call should either be a call to an already-verified routine, or be a recursive call (which we handle separately, as described below).

For code emitted by a compiler for an imperative language with standard looping constructs like "for" and "while," it is usually sufficient to put a cutpoint at the continuation test of each loop, even in the presence of break and continue statements. Sometimes we can do even better. For example, a single cutpoint sometimes suffices to verify a program with two nested loops.

A routine with no loops typically requires no user cutpoints; that is, the starting program location usually sufficies as the only cutpoint. This is true even if the routine contains branches or subroutine calls, including recursive calls.

If the set of cutpoints for a routine is insufficient to cut all the loops in its control flow graph, the symbolic simulation described below may loop forever -- in which case the proof will fail. The "cutpoint to cutpoint" proof for a routine involves symbolic simulation of the machine model. The simulation starts at a cutpoint and assumes that the assertion for that cutpoint holds. We simulate the machine until it either reaches another cutpoint or exits by executing a return instruction. At the resulting state, we must show that the corresponding assertion holds. Recall that each cutpoint has a corresponding assertion. The corresponding assertion for a state in which the routine has just exited is the routine's postcondition. (The above description is a bit of an oversimplification because the simulation from a cutpoint may encounter a conditional branch. In such a case the simulation will split into several simulation branches. What we really must prove is that the state at the end of every simulation branch satisfies its corresponding assertion.)

The proof proceeds by symbolic simulation using several symbolic simulation rules, which are described fully in [15].

Among the rules are ones that:

```
(prove-it
 fact-iter ;the name of the routine
 :wormhole t
 :subroutine-calls nil ;makes for faster proofs
 :user-cutpoints
 ;; List of (PC byte offset . assertion) pairs
 ((6 . (and
          ;; First comes an equality claim about the current state, s,
          ;; in terms of the initial state, s0.
          (equal s
              (standard-cutpoint-state
                  :pc 6
                  :locals (
                      (4 2 (aamp::read-two-local-words 4 s0))
                      (2 2 (fact (+ -1 (gacc::read-data-words 2
                                                (aamp::aamp.denvr s0)
                                                (aamp::aamp.lenv s0)
                                                (aamp::aamp.ram s))))))))

       ;; We claim that the precondition held at state s0.
       ;; So, roughly speaking, any state component (e.g., the code)
       ;; still unchanged from s0 to s still satisfies whatever the
       ;; precondition said about it.
       (fact-iter-precondition s0)

       ;; Asserts that the loop counter at local slot 0 is at most one more
       ;; than the input argument, N (accessed on the AAMP stack at local slot 4)
       (<= (aamp::read-two-local-words 0 S)
           (+ 1 (aamp::read-two-local-words 4 S)))

       ;; Asserts that the loop counter is positive
       ;; (it starts at 1 and goes upward).
       (< 0 (aamp::read-two-local-words 0 S))
       ))) <hints elided>)
```

Figure 7. Assertions at cutpoint.

- Step the state when neither a cutpoint nor a subroutine exit has been reached.
- Stop the simulation when such a point has been reached.
- Handle subroutine calls.

The simulation process is fairly natural. It does not require the user to write a special-purpose program to do the simulation. Rather, the symbolic simulation rules drive ACL2's normal rewriter to do the simulation when the prove-it macro generates a carefully crafted theorem.

The full "cutpoint to cutpoint" proof requires doing the above symbolic simulation for every cutpoint. Once we show that an execution starting from any cutpoint is well-behaved, we can conclude that the routine is partially correct. The prove-it macro automates that reasoning.

As noted above, we handle recursion separately; the full details are given in [15]. Essentially we wrap the "cutpoint to cutpoint" proof inside an induction, on the variable n, of the claim "All calls which terminate within n steps are correct." Thus, when simulation encounters a recursive call we can assume from the inductive hypothesis the call operates correctly. (Since the caller terminates, the callee must also terminate, and it must do so in fewer steps.)

In Figure 7 we specify one user cutpoint for the non-recursive routine FACT-ITER, namely the location which is 6 bytes past the first instruction. We pair the cutpoint with a loop invariant whose most interesting part says that the top stack element is the factorial of one less than the loop index.

The prove-it macro does most of the heavy lifting of the proof, allowing the user to focus on the interesting part: formulating a set of assertions that is inductively strong, and proving helper lemmas about the notions with which the specific program deals. Typically, prove-it proceeds automatically, without very many hints, due in part to a large and growing library of general-purpose lemmas about the AAMP7G.

6. CONCLUSION

We have presented a code proof framework for highly secure applications targeting the Rockwell Collins AAMP7G embedded microprocessor, built upon an executable formal instruction set model of the AAMP7G written in ACL2. We have presented the formally verified partitioning features of the AAMP7G, and shown how the AAMP7G instruction level model is validated through the execution of AAMP binaries. The model is controlled using the same user interface that is used to control the actual AAMP7G chip. We summarized the compositional cutpoint code proof technique, and showed how it is coupled with ACL2's efficient symbolic simulation support to produce a robust code proof framework for AAMP7G machine code. This code can be generated from a number of sources, including traditional compilers, but also including a certifying compiler for the μCryptol cryptographic programming language.

7. ACKNOWLEDGMENTS

Many thanks to the AAMP7G development team at Rockwell Collins, and especially to Matt Wilding, Dave Greve, and Ray Richards for their pioneering work on the AAMP7G partitioning microcode proofs. Special thanks go to SHADE team members: Tom Johnson at Rockwell Collins; RCI co-ops Daron Vroon and Jared Davis; and John Matthews, Lee Pike, and Mark Shields at Galois Connections. Thanks also to Sandip Ray for his work in the area of compositional cutpoint reasoning. We appreciate the comments of the anonymous reviewers, which resulted in a stronger paper. Finally, many thanks to J Moore for his initial work on inductive assertions and operational semantics that directly led to the SHADE cutpoint research, and to J and Matt Kaufmann for their inspiring work on ACL2. This work was performed under contract MDA904-03-C-1781 with the United States Department of Defense. A portion of the material described herein is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.

8. REFERENCES

[1] Best, D., Kress, C., Mykris, N., Russell, J., Smith, W.: An advanced-architecture CMOS/SOS microprocessor. *IEEE Micro*, Aug. 1982, 11–26.

[2] Common Criteria for Information Technology Security Evaluation (CCITSE), Mar. 1999. Available at http://www.radium.ncsc.mil/tpep/library/ccitse/ccitse.html.

[3] Eclipse community, http://www.eclipse.org.

[4] Greve, D., Wilding, M., Hardin, D.: High-speed, analyzable simulators. In Kaufmann, M., Manolios, P., Moore, J.S., eds.: *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, 2000, 89–106.

[5] Greve, D.: Address enumeration and reasoning over linear address spaces. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[6] Greve, D., Richards, R., and Wilding, M.: A summary of intrinsic partitioning verification. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[7] Hardin, D., Wilding, M., and Greve, D.: Transforming the theorem prover into a digital design tool: from concept car to off-road vehicle. In Hu, A. and Vardi, M., eds.: CAV'98. Volume 1427 of LNCS, Springer-Verlag, 1998, 39-44.

[8] Kaufmann, M., Manolios, P., Moore, J S.: *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. ISBN 0792377443.

[9] Matthews, J., Moore J S., Ray, S., and Vroon, D.: Verification condition generation via theorem proving. In Proceedings of LPAR'06, to appear.

[10] Moore, J S.: Inductive assertions and operational semantics. In Geist, D., ed.: CHARME 2003. Volume 2860 of LNCS, Springer-Verlag, 2003, 289–303.

[11] Pike, L., Shields, M., and Matthews, J.: A verifying core for a cryptographic language compiler. Proceedings of HCSS'06, Apr. 2006.

[12] Richards, R., Greve, D., Wilding, M, and van Fleet, M.: The Common Criteria, formal methods, and ACL2. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[13] Rockwell Collins, Inc.: *AAMP7r1 Reference Manual*, 2003.

[14] Shields, M.: A language for symmetric-key cryptographic algorithms and its implementation, Jan. 2006. Available at http://www.cartesianclosed.com/pub/mcryptol/.

[15] Smith, E. Compositional cutpoints for automated machine code proof. In preparation.

[16] Smith, E., Nelesen, S., Greve, D., Wilding, M, and Richards, R.: An ACL2 library for bags (multisets). In Proceedings of ACL2'04, Nov. 2004.

[17] Young, W.: Introducing abstractions via rewriting. In Borrione, D. and Paul, W., eds.: CHARME 2005. Volume 3725 of LNCS, Springer-Verlag, 2005, 402-405.