A Robust Code Proof System for Improved Secure System Evaluation

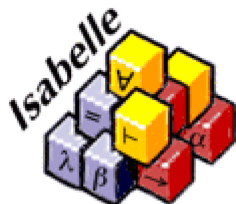
David Hardin

Advanced Technology Center
Rockwell Collins

with contributions by

David Greve and Ray Richards
Rockwell Collins

John Matthews, Mark Shields, and Lee Pike
Galois Connections



Eric Smith
Stanford University

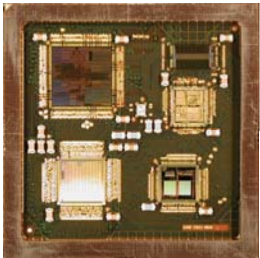
Bill Young
University of Texas at Austin

UNCLASSIFIED

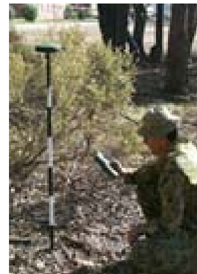
Overview

- We have developed a Robust Code Proof Framework applicable to:
  - Microcode
    - AAMP7G Microprocessor
      - MILS Certification
  - Source Code
    - Separation Kernel
    - VHDL, Verilog (work ongoing, not covered in this talk)
  - Source Code -> Object Code
    - SHADE Program
      - AAMP7G Instruction Set Formal Model
      - Microcryptol Verifying Compiler
      - Compositional Cutpoint Reasoning

A World Leader in Aviation Electronics and Airborne/ Mobile Communications Systems for Commercial and Military Applications



- Communications
- Navigation
- Automated Flight Control
- Displays / Surveillance
- Aviation Services
- In-Flight Entertainment
- Integrated Aviation Electronics
- Information Management Systems



The Problem – High-Assurance for Security Applications

- Flawed implementations can have grave consequences
  - So NSA performs intensive evaluations of critical encryption devices
- Evaluation process is difficult
  - Increasingly numerous crypto implementations
  - Trusted experts are scarce
  - Review process is time-consuming and expensive
  - Optimized crypto algorithms are complex, easy to overlook corner cases
- Highest Evaluation Assurance Level *requires* formal proofs
  - Industry has very little practical experience in this area

Common Criteria

ADVANCED COMPUTING SYSTEMS

- Common Criteria for Information Technology Security Evaluation
  - Internationally recognized standard
  - Provides a common language for vendors and consumers
    - Evaluation Assurance Levels (EALs)
- National Information Assurance Partnership (NIAP)
  - US Common Criteria Certification Authority
- National Security Agency (NSA)
  - Evaluation Authority for formal methods work for ‘high assurance’ certifications

Common Criteria EALs

- EAL 1 – functionally tested
- EAL 2 – structurally tested
- EAL 3 – methodically tested and checked
- EAL 4 – methodically designed, tested, and reviewed
- EAL 5 – semiformally designed and tested
- EAL 6 – semiformally verified design and tested
- EAL 7 – formally verified design and tested

The "EAL scale" is basically logarithmic in evaluation difficulty

Protection Profiles and Security Targets

- These documents tailor the Common Criteria requirements
  - Requirements profiles
- Protection Profiles (PP) specifies requirement profiles for a class of applications
  - e.g., Separation Kernel Protection Profile
- Security Target applies to a specific application
  - Each certification must have a security target

Formal Methods and the CC

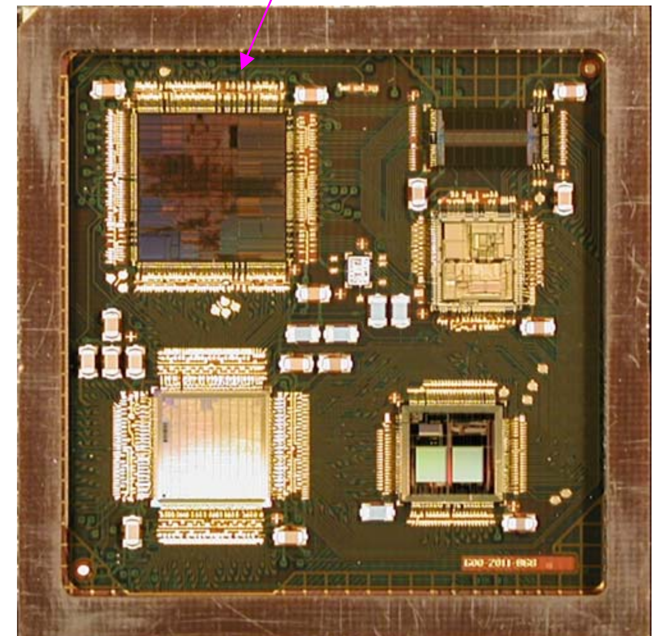
- Formal methods analysis satisfies the following CC sections
  - ADV_FSP (Functional Specification)
  - ADV_HLD (High-Level Design)
  - ADV_LLD (Low-Level Design)
  - ADV_RCR (Representation Correspondence)
  - ADV_SPM (Security Policy Modeling)
- Fundamental properties of the system are proven
- System may be modeled in a formal language
  - Multiple models with a decreasing degree of abstraction
  - Correspondence between levels rigorously proven.
- Properties proven on each model
- Most detailed model shown to correspond to implementation by code-to-spec review

Rockwell Collins AAMP7G CPU

ADVANCED COMPUTING SYSTEMS

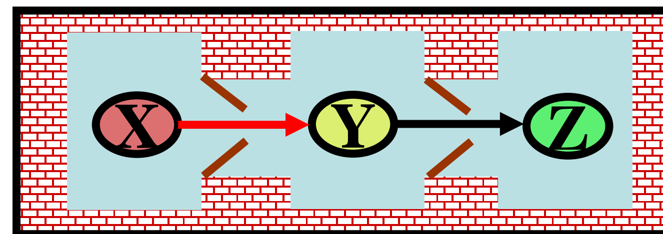
- Developed by RCI Advanced Technology Center
- Used in RCI GPS and Information Assurance products
- High Code Density
- Low Power Consumption (250 mW)
- 100 MHz operation
- Screened for full military temp range
- Implements *intrinsic partitioning*

AAMP7 in GPS SAASM MCM



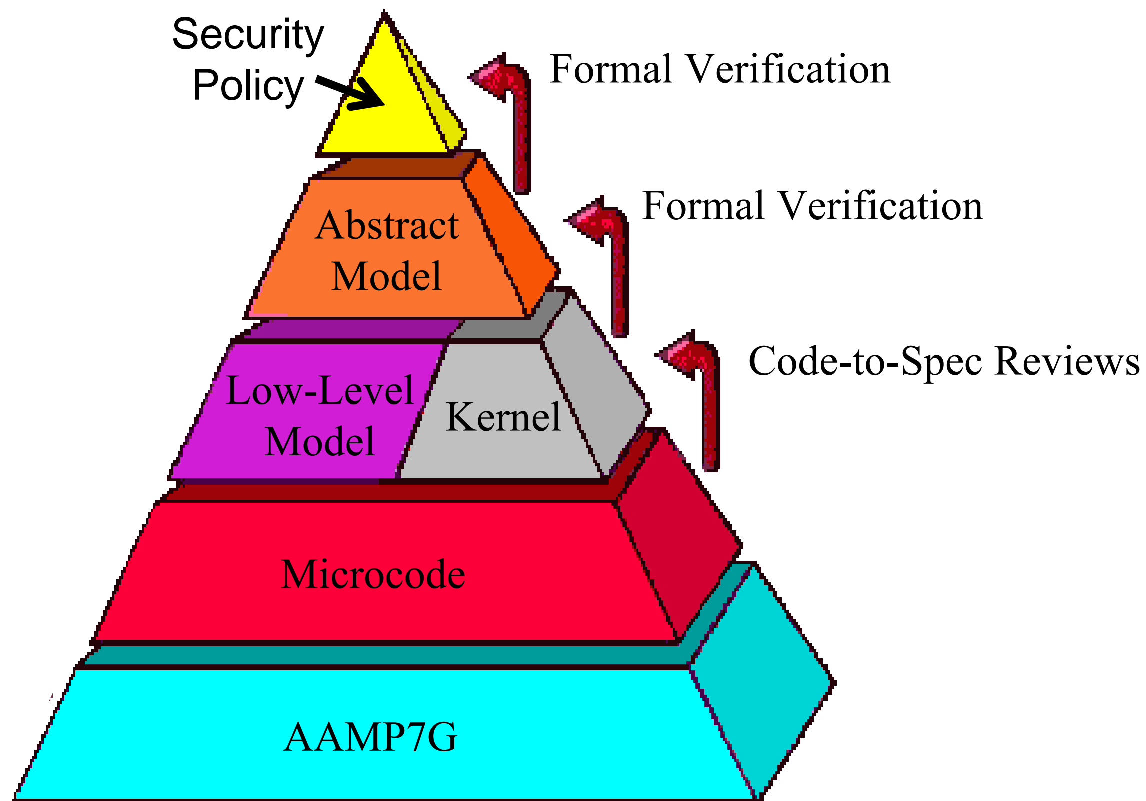
Intrinsic partitioning

- Computing Platform Enforces Data Isolation
- "Separation Kernel in Hardware"



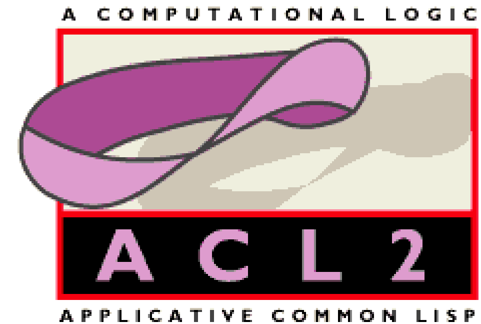
AAMP7G Partitioning Formal Verification

Common Criteria
EAL7 Proof Obligations



The ACL2 Theorem Prover

- A system for the development of machine-checked proofs for theorems expressed in a logic that is an applicative subset of Common Lisp
- Developed by Kaufmann and Moore at the University of Texas and Austin
- *Since ACL2 models are also applicative Common Lisp programs, they can be executed*
- First-order logic
- Proofs are guided by the introduction and proof of lemmas that guide the theorem prover's simplification strategies



ACL2 Details

- Lisp syntax
  - Prefix notation
    - (+ 3 4)
  - Let statements bind variables
    - (let ((x (+ 3 4))) …)
- No loop constructs
  - Looping implemented by recursive functions
    - Obligated to prove termination
- Side-effect free
  - No global variables, all actions are explicit
  - A structure representing system state passed to and returned from most model functions
    - ACL2 single-threaded objects (stobj) – allow state object to be updated ‘in place’ “under the hood”
- Syntactic Sugar
  - Macros can be used to enhance readability and make the resulting models look more like the implementation (C code in many cases)
    - RCI reader macro

```
(%
   (x = (+ 3 4))
   (y = 2)
   (* x y))
```

ACL2 Details (cont'd.)

- Large collection of libraries ("books") to support reasoning about Lisp constructs, data structures, hardware, etc.
- Support for Lisp primitive data types, including mathematical integers, as well as (unsigned-byte n) – very useful for hardware and software modelling
- Proofs are "mostly automated", with the user' s role being more to guide the prover in the right direction rather than to explicitly command every step (although a fully interactive mode also exists)
- Very, very fast rewriter
- Proven to scale to industrial-strength problems (MB's of input text), including processor verifications by
  - AMD
  - IBM
  - Rockwell Collins
- Sophisticated meta-level reasoning, congruence-based rewriting, etc.

Overall AAMP7G Certification Plan

ADVANCED COMPUTING SYSTEMS

Development Process
NSA Certification Rqmnts (TSRD)
Common Criteria (VolII: Functional Rqmnts)
UIC
Protection Profile (Separation Kernel)
TEO
Security Target (AAMP7 SKPS)
Descriptive Top Level AAMP7 Spec
TOC
Descriptive Low Level AAMP7 Spec
AAMP7 SPKS Fail Safe Design Analysis (FSDA)
AAMP7 SPKSSec'ty Verif. Tst (SV Plan, Procedure, Test, Rpt)

Formal Region
Formal Security Policy Model
Correspondence Report
Abstract Machine Model
Correspondence Report
Implementation Machine Model
Validation Report

Evaluation Process
Common Criteria (VOL III: Assurance Rqmnts)
Assurance Plan
Configuration Management Plan
Informal Security Policy Model
Architectural Description
Philosophy of Protection Rpt
Covert Channel Analysis Rpt
Security Features User's Guide
Trusted Facilities Manual

AAMP7G Intrinsic Partitioning Formal Verification

Program Accomplishments

- Developed formal description of separation for uniprocessor, multipartition system
- Modeled trusted AAMP7G microcode
- Constructed machine-checked proof that separation holds of AAMP7G model, using ACL2
- Model subject of intensive code-to-spec review
- Satisfies NSA MILS formal methods evaluation requirements patterned after Common Criteria EAL7+ with respect to ADV
- ***NSA MILS certificate granted in May 2005***
  - AAMP7G can concurrently process Unclassified through Top Secret Codeword information

National Security Agency

Information Assurance Directorate

Information Assurance Directorate
Approval Certificate

The NSA certifies the Advanced Architecture MicroProcessor 7 Government Version (AAMP7G) as a Multiple Independent Levels of Security (MILS) device capable of simultaneously processing unclassified through Top Secret Codeword information, as well as performing JOSEKI decrypt and Type 1 signature verification. The MILS capability of the AAMP7G has been verified using Formal Methods techniques as specified by the EAL-7 level of the Common Criteria. This Certification applies to the AAMP7G which consists of the AAMP7 microprocessor ASIC loaded with the verified boot code that performs the cryptographic functions. This ASIC/boot code combination is packaged in three forms, Rockwell Collins' part numbers 984-3199-001, 984-3200-001 and 984-3201-001.

This certificate allows the use of the AAMP7G in cryptographic applications. The use of the AAMP7G in a system must be accomplished in accordance with the Security Features Users Guide for AAMP7G, 0N686613.

NSA does not make, by virtue of this certification, any statement or other representation regarding the use of the security features of the AAMP7G within a system context. The decision to use the AAMP7G must be made in a total system context employing an Information System Security Engineering process.

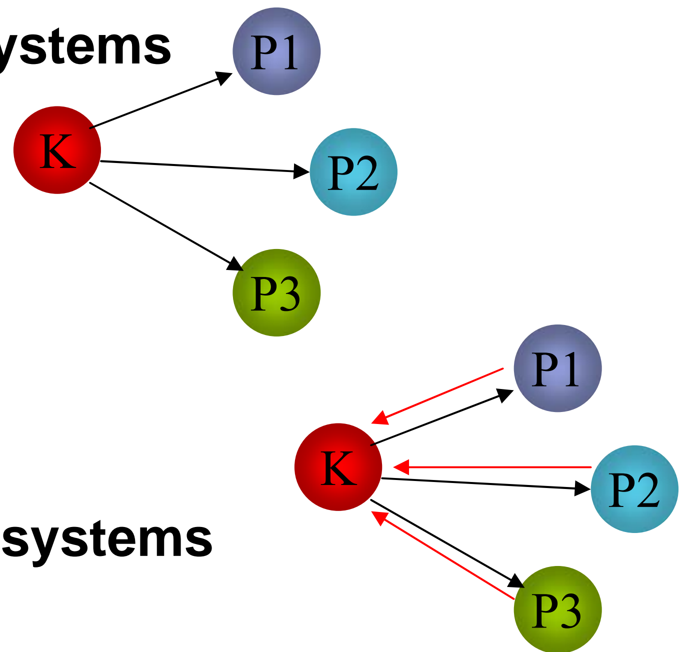
Daniel G. Wolf
Information Assurance
Director

9 May 2005
Date

- RCI IR&D funded
- Capability developed in multiyear RCI formal methods research program

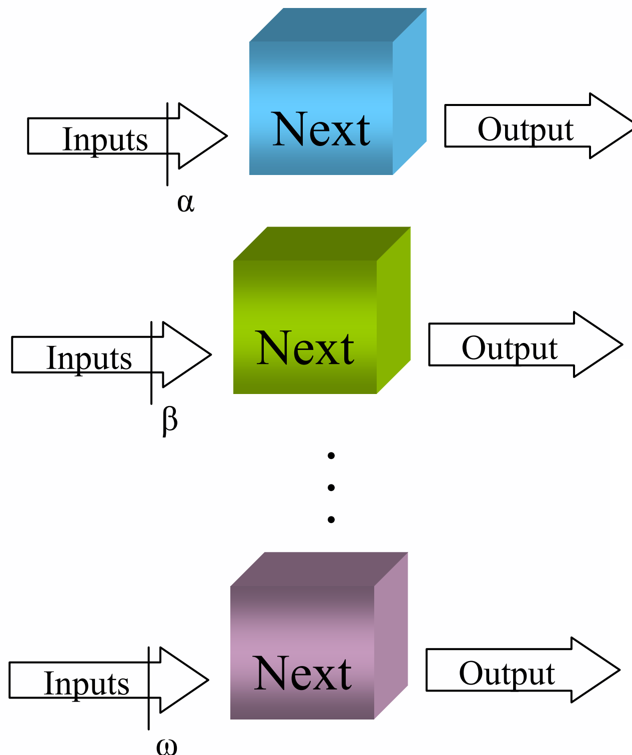
The GWV Formal Security Policy

- GWV security policy developed for AAMP7 verification
  - NSA Type 1 Certification with CC-like formal methods requirements
  - Named after its authors: Greve (RCI), Wilding (RCI), and van Fleet (NSA)
- GWV validated by use in proof of firewall system exhibiting desired infiltration, exfiltration, mediation properties
- GWV only applicable to a narrow class of systems
  - Strict temporal partitioning
  - Kernel state cannot be influenced by execution of code within partitions



- Later generalized to cover a wider range of systems
  - GWVr2

GWVr2



Consider a model that takes n inputs and produces m outputs, where the inputs and outputs are all components of system state.

We could make m copies of this model, each computing a different component of the output.

For each model, only its needed inputs are defined. The remainder of each input state is undefined.

Let's create m input states, and copy into each state only the necessary values to calculate one output, execute the model on each input state, and collect the outputs back into a single state. We will call this Next*

Inputs

Next*

Inputs → Next → Output (α)

Inputs → Next → Output (β)

. . .

Inputs → Next → Output (ω)

Outputs

GWVr2


```
(defthm GWVr2
    (equal
       (next st)
       (next* graph st)))
```

- **The 'next*' function requires an argument that defines the dependencies between inputs and outputs**
  - **Data flow graph (high-level design model)**
- **Paper on GWVr2 presented at SSTC 2005**
- **We have employed GWVr2 to prove information flow properties for a commercial Real-Time Operating System (RTOS) kernel as part of an EAL6+ evaluation**
  - **AFRL funded, at the request of various US DoD programs, including JSF**

Separation Kernel Architecture

user

user

user

user

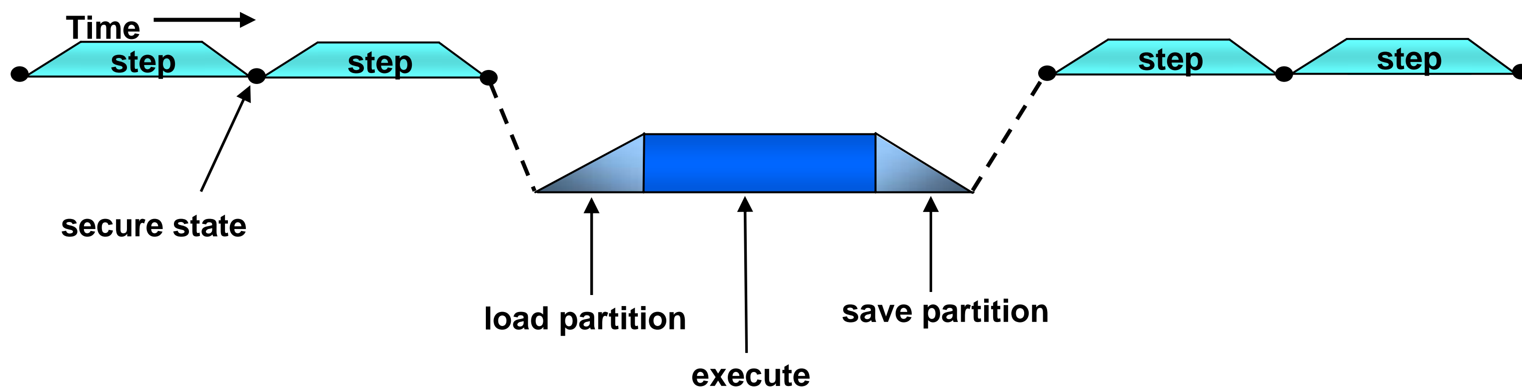
Kernel

ASP/BSP

Processor

Modeling a Separation Kernel

- System state evolves over a series of discrete steps
  - Each step start/ends with the system in a secure state
- The security policy is the notion that the only data that flows between partitions is that which is explicitly authorized by the system's configuration
- Proof of the security policy is complicated by the fact that user partitions can influence kernel state



Detailed Model

- How detailed does the model need to be?
  - The Lowest level model should be
    - At a level that captures the design decisions of the security function of the Target of Evaluation
    - At or near the level of the implementation
      - Facilitate the code-to-spec review

Detailed Modeling in ACL2

ADVANCED COMPUTING SYSTEMS

- Develop syntactic sugar to give a look and feel similar to the implementation
  - Set of macros provides imperative look and feel
  - Makes code-to-spec review easier
- Loops modeled by recursive functions
  - Must prove termination

Detailed Model Example

ADVANCED COMPUTING SYSTEMS

```
(defun RemoveFromList (TheList Element st)
 (%
  (NextInList = (Element -> next))
  (ifx (= NextInList 0)
              st)
  (if (= Element NextInList)
              (% ((& TheList -> First) = (NULL)))
    (%
     (if (= (* TheList -> First) Element)
                ((& TheList -> First) = NextInList)
              st)
     (PrevInList = (Element -> prev))
     ((& PrevInList -> next) = NextInList)
     ((& NextInList -> prev) = PrevInList)))
    ((& Element -> next) = (NULL))
    ((& Element -> prev) = (NULL))))
```

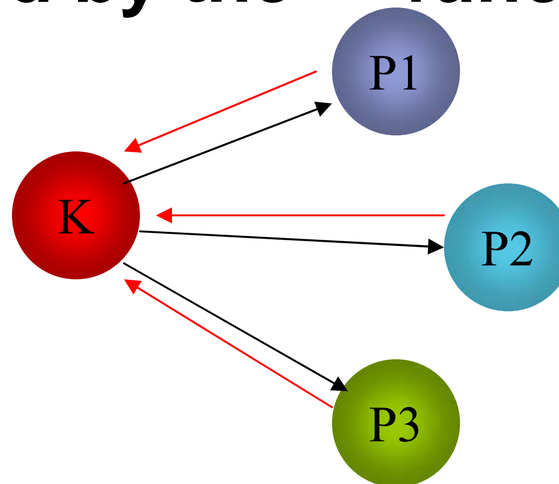
ACL2 Model

```
void RemoveFromList (LIST *TheList, ELE * Element)
 {
  ELE *PrevInList, *NextInList = Element -> next;
  if (!NextInList )
     return;

  if (Element == NextInList)
     TheList -> First = NULL;
  else if (TheList -> First == Element)
     TheList->First = NextInList;

  PrevInList = Element->prev;
  PrevInList->next = NextInList;
  NextInList->prev = PrevInList;
  Element->next = NULL;
  Element->prev = NULL;
```

RTOS Code

Abstract Model

- Abstract model representation is a data flow graph
  - Nodes are pieces of state
  - Edges represent allowed flow of data (unidirectional)
- Maps to CC ADV_HLD
  - Tagged data-flow graph shows breakdown into sub-systems and describes how data flow among them
- Graphs are used by the '*' functions in GWVr2



Proof Outline

- GWVr2 theorem for each model function
  - Is the graph correct?
    - Model function:
      (defun model-fn (a b c st) => st)
    - Graph function:
      (defun model-fn-graph (a b c st g) => (st g))
    - GWVr2 function:
      (defun model-fn* (g a b c st) => st)
    - Theorem:

```
(dethm model-fn-corr
  (equal
    (model-fn a b c st)
    (model-fn* (nth 1 (model-fn-graph a b c st)) a b c st)))
```

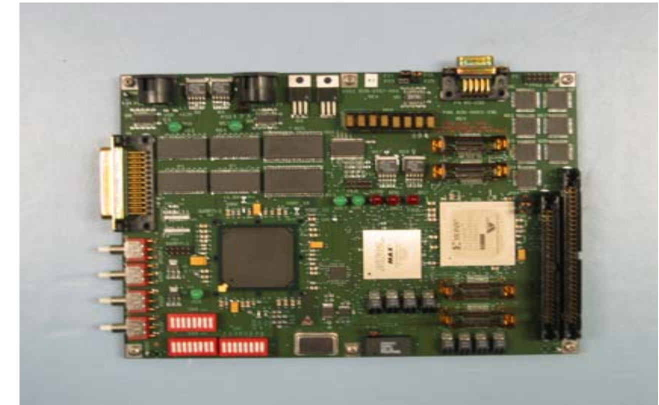
Results

- Successfully modelled the machine-independent kernel code for a commercial Real-Time Operating System
  - ~6000 lines of C
  - Machine-dependent code was analyzed with traditional inspections
- Created information flow graphs for the kernel
- Proved GWVr2 for the kernel
- Rough level of effort: 3 FM researchers for a year, plus 2 engineers to inspect machine-dependent code
- Completed code-to-spec review in May
- NIAP labs testing, etc. underway
- Will be able to say more when the certification is complete!

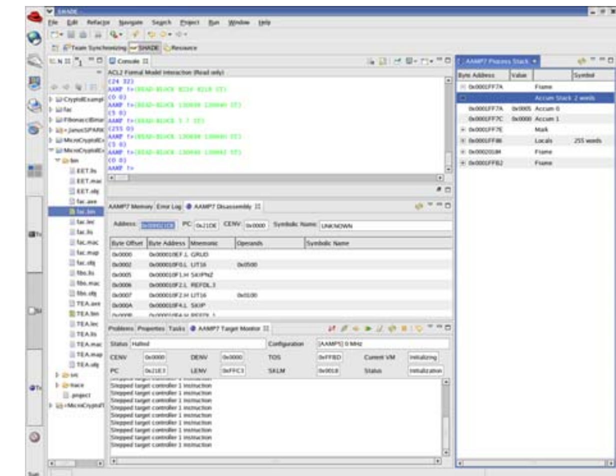
Secure, High Assurance Development Environment (SHADE)

Program Objectives

- Provide a “nuts-and-bolts” partitioned development environment.
- Develop tools and techniques to provide formal analysis at the instruction level for the AAMP7 processor
- Develop a verifying compiler for an “embeddable” subset of the Cryptol cryptographic language targeting the AAMP7
- Demonstrate a convenient, high-assured toolchain path from high-level algorithm description to load image.



AAMP7G development board



Eclipse-based AAMP7G development environment

RCI subcontractors: Galois Connections, University of Texas at Austin

SHADE Summary

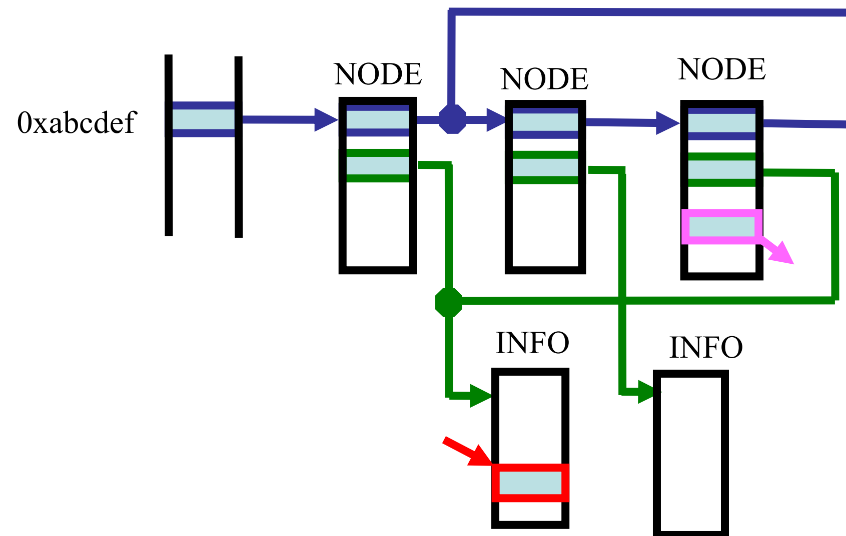
AAMP7 Instruction-Set Formal Model

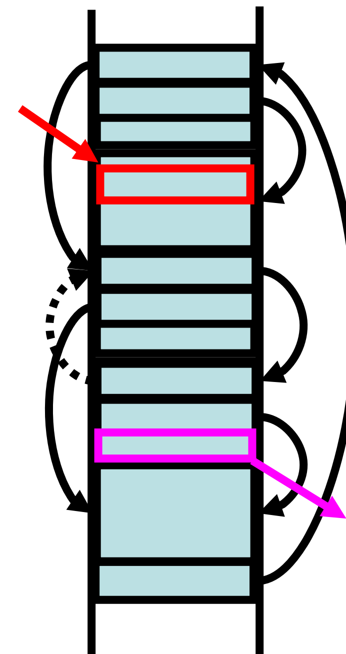
- Provides instruction-level simulator for the AAMP7
- Written in ACL2
  - *~100 KSLOC with all RCI support books*
  - *~500 MB Lisp heap required*
- Can be used as a processor simulator, as well as a vehicle for proof
  - Validated by loading AAMP processor diagnostic tests into (simulated) memory, and running the model
- Utilizes ACL2 single threaded object (stobj) to model CPU state; stobj updates are performed "in place", greatly reducing garbage generation at model execution time
- GACC (Generalized Accessor) library used to model memory, same as used in AAMP7 separation proofs
- New bitvector library, "super-ihs", extends ACL2 Integer Hardware Specification (IHS) library

Data Structure Representation

Programmer's view -- "boxes and arrows"

Reality – mapped into a single linear address space





We must "face reality" in order to verify a compilation

GACC: Generalized Accessor Library

ADVANCED COMPUTING SYSTEMS

- A means of describing linearized data structures
  - Really just a list of addresses
  - Distinguishes pointer and data locations
- Rules for resolving read/write operations
  - (read list1 (write list2 values ram)) = (read list1 ram)
  - (read list1 (write list1 values ram)) = values
- Rules for preserving structure
  - Writes to data locations don't change data structure shape
- Efficient rules for disjoint/subset/unique relations
  - Linear Time/Space
  - Free-variable matching, Meta-rules
  - Useful for alias analysis

Call graph with "hot spots" for ACL2 AAMP Model

AAMP7G Partition Views

SHADE - Eclipse SDK

File Edit Refactor Navigate Search Project CodePro Run Window Help

SHADE

AAMP7 Disassembly | AAMP7 Memory | AAMP7 Partition Schedules

Schedule Name	VCE Address	Partition Name	Next VCE	VM #	Time Count
Cold Reset 0					
	0x00000040	mini_rte__startup	0x0000007C	0	0
	0x0000007C	mini_rte__startup	0x000000B8	1	2000
	0x000000B8	mini_rte__startup	0x000000F4	2	2000
	0x000000F4	mini_rte__startup	0x0000007C	3	2000
Cold Reset 1					
Cold Reset 2					
Warm Reset					
Power Down					

Console | AAMP7 History | AAMP7 Partition Status

Partition Name	VM #	Status	Location	CENV	PC	DENV	LENV	Time Count	Control Block	State
mini_rte__startup	0	Continue (suspen...		0x0000	0x98CA	0x0000	0xBA50	0	0x00008000	0x00020020
			Boot:_ada_boot line 6	0x0000	0x98CA		0xBA50			
			Ada_Main:ada_main__main line 34	0x0000	0x9886		0xBA56			
			Mini_Rte:mini_rte__startup line 300	0x0000	0x9117		0xBA5E			
mini_rte__startup	1	Continue (suspen...		0x0000	0xAB7C	0x0001	0x26B3	2000	0x0000A000	0x0002004C
mini_rte__startup	2	Continue (suspen...		0x0000	0x6B80	0x0000	0xE23C	2000	0x00006000	0x00020078
mini_rte__startup	3	Continue (current)		0x0000	0xEB84	0x0001	0x6235	2000	0x0000E000	0x000200A4

Problems | Properties | Tasks | AAMP7 Partition Access Rights | AAMP7 Target Monitor

Partition Name	VM #	Region#	Low Address	High Address	Source Mode	Type Mode	Execute Mode
mini_rte__startup	0	0	0x00008000	0x000098DF	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	0	1	0x00013000	0x0001759B	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	1	0	0x0000A000	0x0000BED3	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	1	1	0x00023000	0x0002759F	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	2	0	0x00006000	0x00007E77	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	2	1	0x00018000	0x0001C59F	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	3	0	0x0000E000	0x0000FD2B	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	3	1	0x00010000	0x0001051B	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	3	2	0x00028000	0x0002C59F	TAU/Data	Write/Read	Err/Exec/User

SHADE -

File Edit Refactor Navigate Search Project Run Window Help

Team Synchronizing SHADE Resource

Console

ACL2 Formal Model Interaction (Read only)

```
(24 32)
AAMP !>(READ-BLOCK 8216 8218 ST)
(0 0)
AAMP !>(READ-BLOCK 130938 130940 ST)
(5 0)
AAMP !>(READ-BLOCK 5 7 ST)
(255 0)
AAMP !>(READ-BLOCK 130938 130940 ST)
(5 0)
AAMP !>(READ-BLOCK 130940 130942 ST)
(0 0)
AAMP !>
```

ACL2 session

AAMP7 Process Stack

Byte Address	Value		Symbol
0x0001FF7A		Frame	
		Accum Stack	2 words
0x0001FF7A	0x0005	Accum 0	
0x0001FF7C	0x0000	Accum 1	
0x0001FF7E		Mark	
0x0001FF86		Locals	255 words
0x00020184		Frame	
0x0001FFB2		Frame	

Process Stack

CryptolExampl
fac
FibonacciBinar
> JanusSPARK
MicroCryptolE>
MicroCryptolE>
bin
EET.lis
EET.mac
EET.obj
fac.axe
fac.bin
fac.lec
fac.lis
fac.mac
fac.map
fac.obj
fibs.lis
fibs.mac
fibs.obj
TEA.axe
TEA.bin
TEA.lec
TEA.lis
TEA.mac
TEA.map
TEA.obj
src
trace
.project
> MicroCryptolT

AAMP7 Memory | Error Log | AAMP7 Disassembly

Address: 0x000021DE PC: 0x21DE CENV: 0x0000 Symbolic Name: UNKNOWN

Byte Offset	Byte Address	Mnemonic	Operands	Symbolic Name
0x0000	0x000010EF.L	GRUD		
0x0002	0x000010F0.L	LIT16	0x0500	
0x0005	0x000010F1.H	SKIPNZ		
0x0006	0x000010F2.L	REFDL.3		
0x0007	0x000010F2.H	LIT16	0x0100	
0x000A	0x000010F4.L	SKIP		
0x000B	0x000010F4.H	REFDL.1		

Disassembly

Problems | Properties | Tasks | AAMP7 Target Monitor

Status	Halted			Configuration	[AAMP5] 0 MHz		
CENV	0x0000	DENV	0x0000	TOS	0xFFBD	Current VM	Initializing
PC	0x21E3	LENV	0xFFC3	SKLM	0x0018	Status	Initialization

```
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
Stepped target controller 1 instruction
```

Console

AAMP7G ACL2 Formal Model Integration with Eclipse AAMP7G Tools

Sun

Cryptol

- **Galois' domain-specific language for cryptography algorithms**
  http://www.cryptol.net

- **Cryptol features:**
  - **Purely functional**
  - **Size-indexed bitvector types, no limits on bitvector size**
  - **Lazy infinite streams**
  - ***Not* Turing-complete**

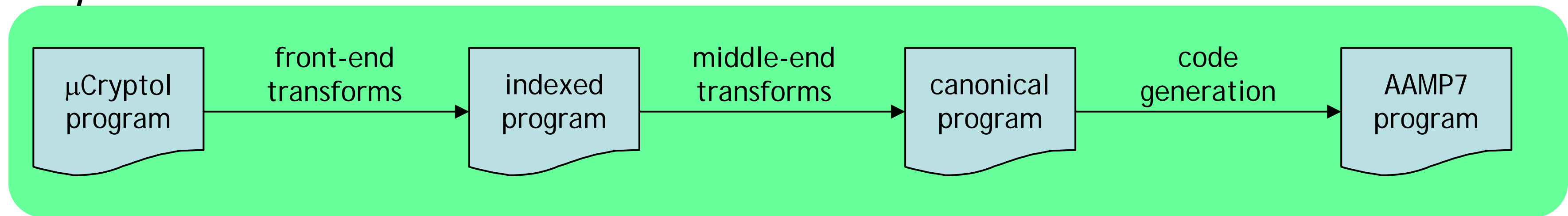
- **µCryptol**
  - **Cryptol subset, tailored for systems with constrained memory**
  - **Formal semantics**
  - **Designed for verification**
  - **Creating a verifying compiler targeting the AAMP7G**
  - **See paper in HCSS06 Proceedings**

Why a verifying compiler for µCryptol?

- Cryptographic systems need to be correct
  - NSA is a demanding customer
- Cryptographic systems are difficult, expensive to certify
  - A verifying compiler could markedly reduce code-to-spec review costs and reduce time-to-market for cryptographic devices
- Reference Cryptol specifications for common crypto algorithms are available
- A domain-specific language, such as Cryptol, seems to present lower risk than attempting a verifying compiler for a general-purpose programming language
- Cryptol is a Galois Connections design, so we can state its specification precisely
- The AAMP7G is an “easy” code generation target (think JVM)
- The AAMP7G is a Rockwell Collins design with a precise specification
- Theorem prover technology has matured sufficiently to make this program feasible

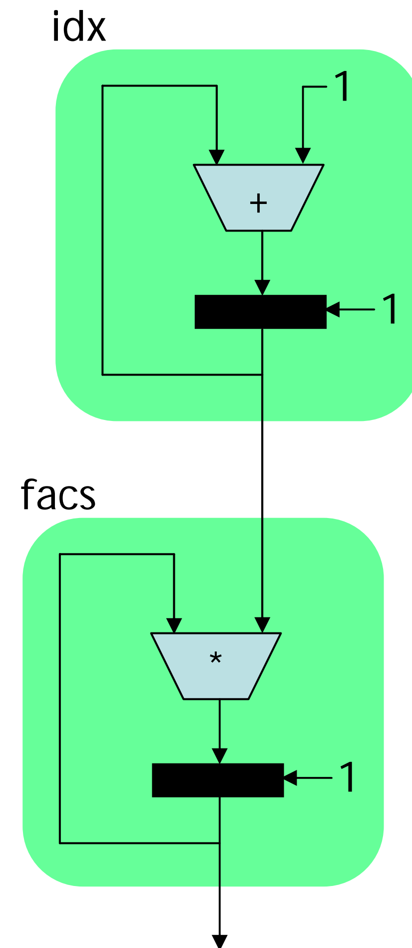
Compiler Architecture

SHADE Compiler



Example: factorial (mod 2^8)


```
fac : B^32 -> B^8;
fac i = facs @@ i
  where {
    rec
      idx : B^8^inf;
      idx = [1] ## [x + 1 | x <- idx];
    and
      facs : B^8^inf;
      facs = [1] ## [x * y | x <- facs
                           | y <- idx];
  };
```



Stream values:

```
idx  = [1, 2, 3, 4,  5,   6,   7,   8, …]
facs = [1, 1, 2, 6, 24, 120, 208, 176, …]
```

Stage 1: Compile to indexed form

SHADE Compiler

µCryptol program → (front-end transforms) → indexed program → (middle-end transforms) → canonical program → (code generation) → AAMP7 program

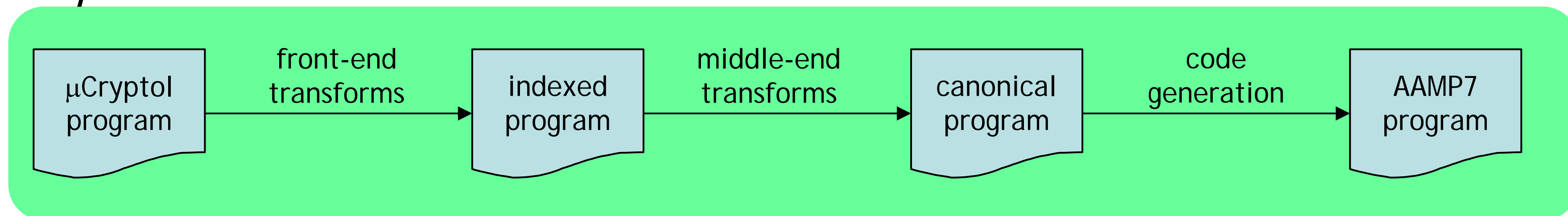
Stage 1: Compile to indexed form

- Each stream represented as first-order function taking index to stream element
- Nested definitions lambda-lifted to top-level
- Pattern-matching and stream/vector comprehensions compiled away
- Program can now be shallowly embedded into ACL2

```
idx : nat -> B^8;
idx n =  if n = 0 then 1
         else idx (n-1) + 1;

facs : nat -> B^8;
facs n =  if n = 0 then 1
          else facs (n-1) * idx (n-1);

fac : B^32 -> B^8;
fac i = facs (toNat i);
```

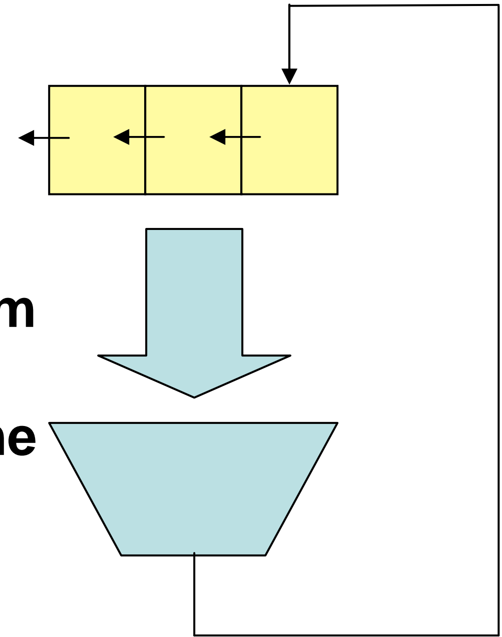
Stage 2: Compile to canonical form

SHADE Compiler



Stage 2: Compile to canonical form

- **Each clique of mutually recursive stream functions represented by single tail-recursive function**
- **Each tail-recursive function takes an extra tuple of *history buffers***
- **Stream dependency analysis calculates minimum length of each history buffer**
- **Complex Cryptol primitives left unchanged, some simple ones are inlined**

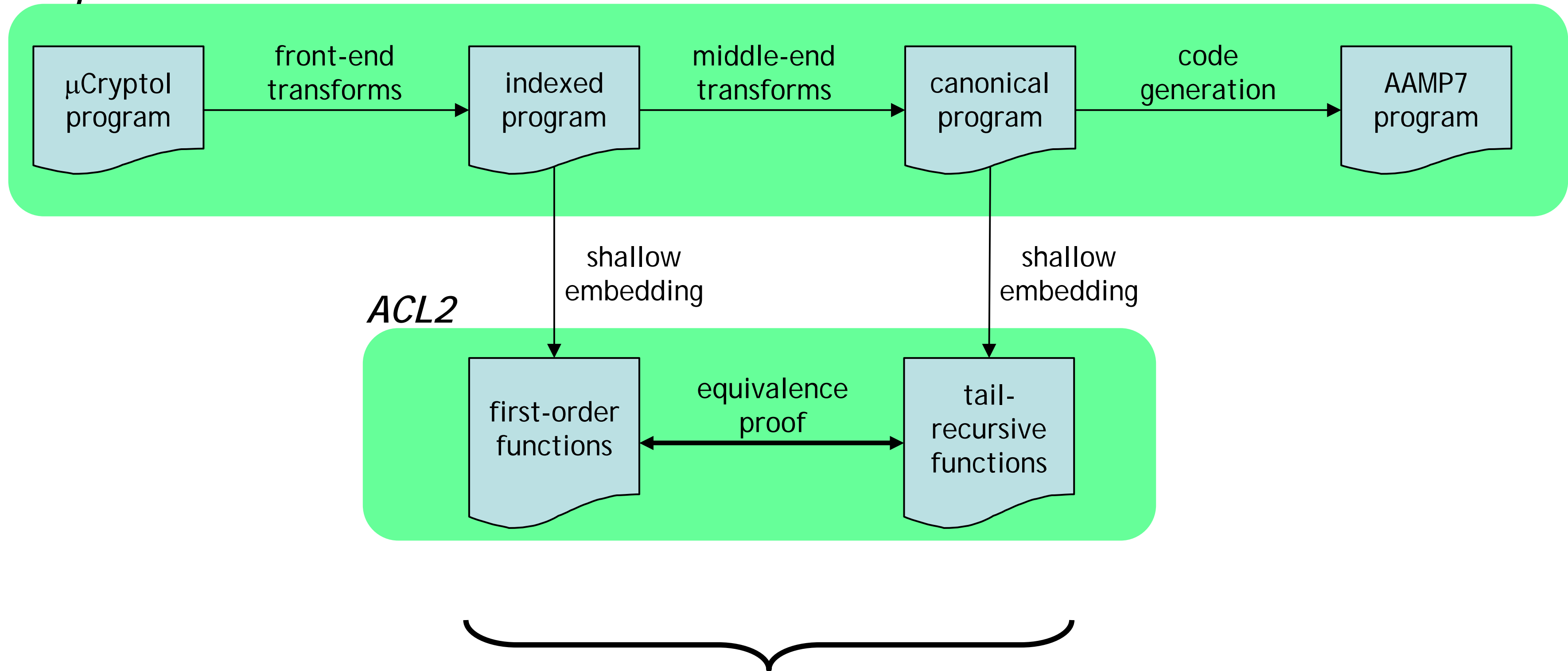


- **Detailed Steps in Indexed -> Canonical:**

1. Push stream applications
2. Collapse arms
3. Align arms
4. Takes/segments to indexes
5. Convert to iterator form
6. Eliminate simple primitives
7. Eliminate zero-sized values
8. Inline and simplify
9. Introduce temporaries
10. Eliminate nested definitions
11. Share top-level value definitions
12. Box top-level definitions
13. Eliminate shadowing

Stage 2: Verification Architecture

- **Use ACL2 to verify compiler middle-end transformations**

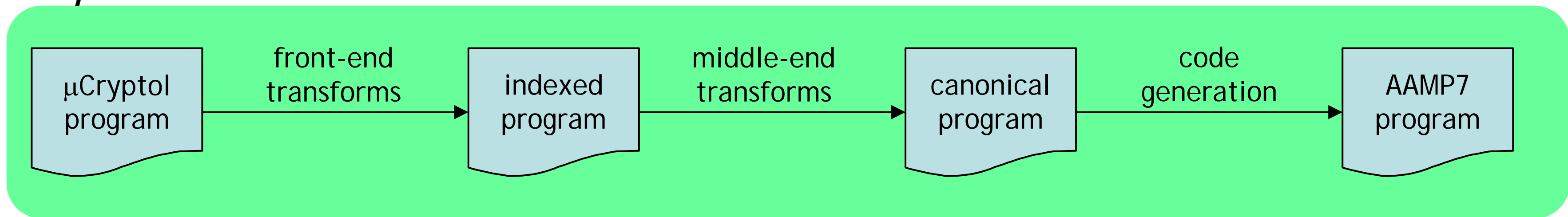


Results

- ACL2 macro that can automatically prove equivalence of indexed to canonical forms, for all examples
  - factorial, alt-factorial
  - Fibonacci, 3-Fibonacci, 5-Fibonacci
  - TEA, AES, RC6
- AES proof takes about 20 minutes on a 1.5 GHz G4 Powerbook
- See Pike, Shields, and Matthews paper in ACL2'06 Proceedings for more details

Stage 3: Generate machine code

SHADE Compiler



Stage 3: Generate machine code

- History buffers represented as circular imperative arrays
  - Optimized away if history length is small
- Compiler statically allocates history buffers
- Calls library routines for multiple-word Cryptol primitives such as arithmetic, shift, rotate, etc.

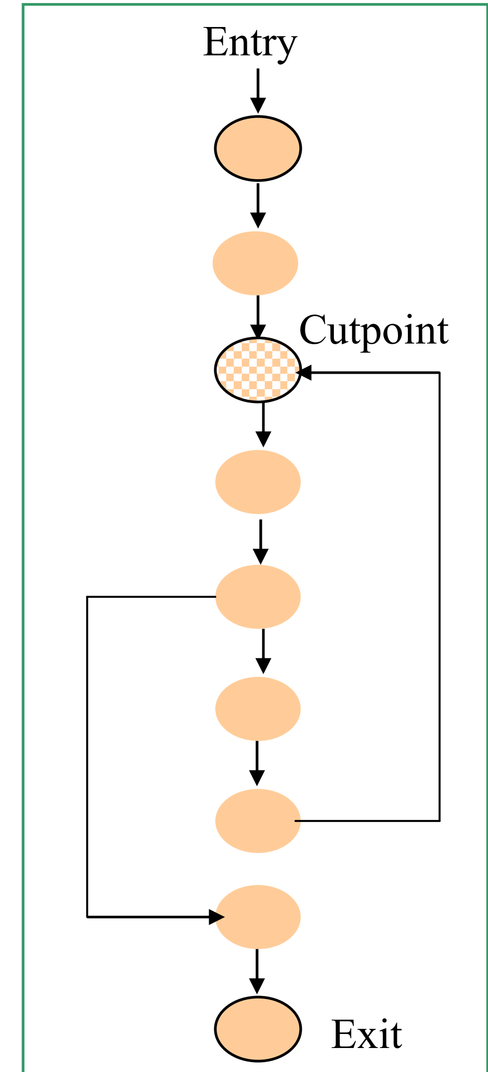
Stage 3: Verification Architecture

Desired Theorems (in general)

- If machine starts at a state satisfying program's precondition (entrypoint assertion), then
  - *Partial correctness*: if the machine ever reaches an exitpoint state, then the first exitpoint reached satisfies the program's postcondition (exitpoint assertion).
  - *Termination*: the machine will eventually reach an exitpoint
- However, we don't want to
  - write and verify a VCG
  - manually define a *clock function*
    - computes for each program state exactly how many steps are needed to reach the next exitpoint

Underlying Verification Method – Compositional Cutpoint Technique

- Sound and automatic theorem proving technique for generating verification conditions from a small-step operational semantics
- Inspired by J Moore presentation at HCSS 2004
- Cutpoints and their state assertions for a given subroutine must be specified
- Symbolic simulation of processor model takes us from cutpoint to cutpoint, until we reach subroutine exit
- Compositionality: Once cutpoint proof is done for a given subroutine, we don't have to reason about it again if it's called by another subroutine
- No Verification Condition Generator required
- See *Verification Condition Generation via Theorem Proving*
  John Matthews, J Moore, Sandip Ray, Daron Vroon, 2006 (LPAR'06, to appear)
- Has been used it to verify a 600-line JVM program implementing a generic CBC-mode encryption



AAMP7G Machine Code Proofs using Compositional Cutpoint Method

- Preconditions, e.g.
  - Code to be proved is loaded into memory
  - Input parameter is within range for a given algorithm
- Postconditions
  - e.g., fact(x) on top of stack after running AAMP7G machine code for factorial
- Frame Conditions
  - e.g., Only local variables and operand stack memory needed to implement factorial are modified by executing AAMP machine code for factorial
- Compositional Cutpoint Proof Technique
  - No Verification Condition Generator required
- Generation of the above information can be done mostly automatically
- See Hardin, Smith, and Young paper in ACL2'06 Proceedings for more details

Example Program – Iterative Factorial


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

Machine Code Proofs – Preconditions Example


```
(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)))))
```

Machine Code Proofs – Postconditions Example


```
;; 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))
```

Machine Code Proofs – Assertions at Cutpoint


```
(prove-it  ;; Proof driver macro
  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))))))))

         ;;  Precondition still holds (e.g., code has not been modified)
         (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>)
```


Rockwell Collins

Issue: Verifying compiler front-end

SHADE Compiler

µCryptol program → front-end transforms → indexed program → middle-end transforms → canonical program → code generation → AAMP7 program

indexed program → shallow embedding → first-order functions

canonical program → shallow embedding → tail-recursive functions

AAMP7 program → deep embedding → AAMP7 state machine

ACL2

first-order functions ↔ equivalence proof ↔ tail-recursive functions ↔ equivalence proof ↔ AAMP7 state machine

Extended Verification Architecture

ADVANCED COMPUTING SYSTEMS

- **Use Isabelle/HOLCF to verify front-end compiler transformations**

SHADE Compiler

µCryptol program → (front-end transforms) → indexed program → (middle-end transforms) → canonical program → (generate code) → AAMP7 program

µCryptol program → (shallow embedding) → HOLCF functions

indexed program → (shallow embedding) → first-order functions (HOLCF)

canonical program → (shallow embedding) → tail-recursive functions (ACL2)

AAMP7 program → (deep embedding) → AAMP7 state machine (ACL2)

HOLCF

HOLCF functions ↔ first-order functions

first-order functions → (translate) → first-order functions (ACL2)

ACL2

first-order functions ↔ tail-recursive functions ↔ AAMP7 state machine

deep embedding of ACL2 in HOL

Translating ACL2 to Isabelle

- Mike Gordon, Matt Kaufmann, Warren Hunt and James Reynolds are building an ACL2 external oracle for HOL4
  - Defined ACL2 universe as `sexp` datatype in HOL
  - Used ACL2 axioms to define ACL2 primitives in HOL
  - Expressions and formulas defined over `sexp` invoke ACL2
  - Result of ACL2 is trusted
- John Matthews has developed a prototype `sexp` datatype for Isabelle, and proved equivalence between a shallow embedding of the μCryptol factorial example into Isabelle/HOLCF and a translated version of indexed form using the `sexp` datatype

Summary

Rockwell Collins and partners have developed robust techniques and tools to improve high-assurance system evaluations by:

- Making use of automated theorem provers to provide formal proofs as required by EAL7
- Producing executable formal models of computing platforms that can also be validated by execution of production tests
- Pioneering techniques for automating hardware, microcode, object code and source code verification
- Designing and implementing a verifying compiler for a subset of the Cryptol language