Compositional Cutpoint Verification

Eric Smith (Stanford University)

Collaborators:

David Dill (Stanford University)

David Hardin (Rockwell Collins)

Contact ewsmith@stanford.edu

Background

- Based on "**A Symbolic Simulation Approach to Assertional Program Verification**" by Matthews, Moore, Ray, and Vroon (2005, to appear).
- I call that work "MMRV".
- This talk won't presume familiarity with it.

My Contributions

- Handles non-recursive subroutine calls.
- Handles recursive calls more naturally.
- Less user input required.

- Cutpoint idea is not mine!

Eventual Big Picture

- User writes Java code and annotations in JML (the Java Modeling Language).
- We automatically generate an ACL2 book which states and proves the correctness theorem.
  - Probably automatic only for simple properties.

Java Code with JML Assertions

```
class Demo {
   …
   //@ requires n < 2147483647;
   //@ ensures n = \old(n)+1;
   public static int inc(int n) {
      return n+1;
     }
}
```

M5 Byte-code

```
(defconst m5::*inc-method*
  '("inc" (m5::int) m5::nil
    (m5::iload_0)                          ;;;  0
    (m5::iconst_1)                         ;;;  1
    (m5::iadd)                             ;;;  2
    (m5::ireturn)                          ;;;  3
    ))
```

ACL2 Proof

```
(defun inc-precondition (s)
  (and (< (local_0 s) (+ -1 (expt 2 31)))
       (current-method-is m5::*inc-method* (th) s)
       (standard-m5-preconditions s)))

(defun inc-poststate (s0)
  (pop-frame-and-push-return-value
    (+ 1 (local_0 s0))
    (th)
    s0))

(local (prove-it inc))

;redundant: done by prove-it.
(defthm inc-correct
  (implies (and (inc-precondition s)
                (equal (program-counter s)
                       (starting-program-counter))
                (eventually-returns s))
           (equal (run-until-return s)
                  (inc-poststate s))))
```

Eventual Big Picture (cont.)

- The tool to generate ACL2 book from Java/JML is still in the very early stages.
- This talk will focus on the contents of the ACL2 book and how to certify it.

Methodology Is In Use

- Verified about 18 small JVM byte-code programs, including ones with method calls and recursion.
  - Only trivial hints required.
- Verified two programs for Rockwell Collins' AAMP7 microprocessor.
  - Recursive and iterative factorial.
  - AAMP is very complicated.

Rest of the Talk

1. What theorem do we prove?
2. How do we prove it using cutpoints?

“Compositional Verification”

- Verify programs one component at a time.
- After you've verified a component you should never have to re-analyze it.
- No in-lining!

My Methodology

- The components are subroutines.
- I require that the machine track pending subroutine calls using a stack.

Verifying a Whole System

- Useful lie: We verify one subroutine at a time. Before verifying subroutine `foo` we verify all the subroutines that `foo` calls.

Verifying a Whole System (cont.)

- The truth: We verify one recursive clique of subroutines at a time.
- Need a special way to handle recursion and mutual recursion.
  - Won't already have verified callees.

What theorem do we prove?

- “If we are about to run inc, and we run it until it returns, then the post-condition is true when we return, assuming that the pre-condition was true when we started.”

Correctness Theorem

Rough approximation:

```
(defthm inc-correct
  (implies (inc-pre s)
           (inc-post (run-until-return s))))
```

Q: What does **`run-until-return`** do if the routine never returns?

A: We have no idea! It's defined in terms of a partial function, so it's not meaningful in that case.

Partial vs. Total Correctness

We can either:

- Include a termination assumption.
  "If it terminates, it's correct."
  This is partial correctness.

- Prove termination and drop the assumption.
  "It terminates, and it's correct."
  This is total correctness.

Partial Correctness in a Compositional Setting

```
(defthm inc-correct
  (implies (and (inc-precondition s)
                (eventually-returns s))
           (inc-postcondition (run-until-return s))))
```

- **`eventually-returns`** is also defined in terms of a partial function.

Precondition

```
(defun inc-precondition (s)
  (and (< (local_0 s) (expt 2 31))
       (current-method-is m5::*inc-method* (th) s)
       (standard-m5-preconditions s)))
```

Applies to the "pre-state“.

The state just after the new frame is pushed on behalf of the subroutine and just before the routine's code starts execution.

Postcondition

- Applies to the first state in which the stack height has decreased.
  - The state just after the method returns.
- Specifies the final state in terms of the prestate.

Postcondition for inc

```
(defun inc-postcondition (s0 s)
   (equal (top (stack (top-frame (th) s)))
          (+ 1 (top (stack (top-frame (th) s0))))))
```

Seems reasonable…

Frame Conditions

- But what about the other pieces of the state?
  - How does inc affect the heap?
  - The class table?
  - Stack frames farther down in the stack?
- Not allowed to reexamine inc later.
- Need to handle these “frame conditions” (old term from AI).

Frame Conditions (cont.)

- We could list out all the state components that don't change.
- Tedious.
- Impractical: Too many memory addresses to list them all.

Post-state vs. Postcondition

- Trick: We phrase the postcondition as an equality.

```
(equal (run-until-return s0)
       (pop-frame-and-push-return-value
          (+ 1 (m5::local_0 s0)))
          (th)
          s0))
```

- "Final state is the initial state, modified in some way."
- Much stronger!
  - Anything we don't explicitly list must stay the same.

Possible Objection #1

- I don’t want to specify every state component.
  - Ex: temporary registers
  - Ex: junk left in de-allocated memory
- Solution: “Don't-care Specification".
  - Dave Greve calls it "wormhole abstraction".

Don’t-care Specification

- For some state components, the program just "does whatever it does".

```
(equal (run-until-return s0)
       (modify s0
               :top-elem (fact (arg s0))
               :temp (get-temp (run-until-return s0))))
```

- The proof for the **:temp** state component is trivial.
- We’d better not need to know anything about **temp**!

Three Types of Things

1. Things that change in ways we care about.

   Everybody handles those.

2. Things that must not change.

   Handled by equality phrasing.

3. Things we don’t care about.

   Handled by don’t-care specification.

Aside: Don't-care Specification in Loop Invariants

- Sometimes we really can't say what a state component is.
- Q: What is the loop counter on an arbitrary loop iteration?
  - We'd like to use equality phrasing for the invariant.
- A: The loop counter is whatever it is!
- But we can add extra statements about it:
  - Ex: "… and the loop counter's factorial is on top of the stack."

Possible Objection #2

- What if I can only give a predicate?
- Ex: “Foo returns an even integer.”
- Trick: Use a “fix function”.

```
(equal (run-until-return s0)
       (pop-frame-and-push-return-value
          (even-fix (m5::local_0 (run-until-return s0))))
          (th)
          s0))
```

Possible Objection #2 (cont.)

- Postcondition phrasing and equality phrasing have the same logical strength.

- Proof:

  (postcondition s)

  iff

  (equal s (if (postcondition s) s (not s))).

Correctness Theorem for **`inc`**

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

- This rule rewrites **`(run-until-return s)`**.

Proving the Theorem Using Cutpoints

- Annotate the program with assertions at certain cutpoints.
- Prove that assertions are preserved from one cutpoint to the next.

- Control flow graph has many arrows.
- Prove one arrow at a time.
- In return we get a proof of the whole routine.

Proving the Theorem Using Cutpoints (cont.)

- I'll only talk about partial correctness.
- Total correctness is similar but requires showing some ranking function decreases from one cutpoint to the next.

MMRV Provides

- A generic partial correctness result.
- A macro (defsimulate) to help you verify programs by using a :functional-instance of that result.
  - Functional instantiation lets you instantiate a result about some generic constrained functions to get a result about your functions -- as long as your functions satisfy the constraints.

Generic Functions

Pre(s) - precondition

Post(s) - postcondition

Cut (s) - cutpoint recognizer

Assert(s) - assertion that must be true at the cutpoints

- If PC=0, then assertion-for-pc-0 holds.
- If PC=7, then assertion-for-pc-7 holds.
- ….

Exit(s) - exit state recognizer

Next(s) - step the state

Nextc(s) - step the state until you find a cutpoint

- Defined using a partial function.

5 Constraints (cont.)

Partial correctness follows from:

1. pre(s) => cut(s)
2. pre(s) => assert(s)
3. cut(s) ^ assert(s) ^ not(exit(s)) =>
   assert(nextc(next s))
4. exit(s) => cut(s)
5. exit(s) ^ assert(s) => post(s)

Symbolic Simulation

- Need to prove the hard constraint:
  cut(s) ^ assert(s) ^ not(exit(s)) => assert(nextc(next s))
- Start at a cutpoint and assume the assertion holds there.
- (Also get to assume we're not already at an exit point.)
- Must show that if we run until the next cutpoint, the corresponding assertion is true there.

- Proof splits into cases, one for each non-exit cutpoint.
- We do symbolic simulation in each case.

Symbolic Simulation (cont.)

Proof for the cutpoint at pc 7:

- Need to prove:

  pc(s)=7 ^ assert-for-pc-7(s) => assert(nextc(next s))

- The first call to next just gets us past pc 7.
- Opening the call to next we get:

  assert(nextc(modify… s))

Symbolic Simulation (cont. 2)

- Now need to simplify: assert(nextc(modify… s))
- Symbolic simulation rules:
  - Rule 1: cut(s) => nextc(s) = s
  - Rule 2: not(cut(s)) => nextc(s) = nextc(next s)
- Each call to **`next`** expands into a **`modify`**.
- Nested **`modify`**s get simplified.

Symbolic Simulation (cont. 3)

- Eventually we should hit a cutpoint, and the simulation then stops.
- But consider a loop which does not contain a cutpoint.
  - Might symbolically simulate forever without reaching a cutpoint!
- So we require that every loop contain a cutpoint.

Symbolic Simulation (cont. 4)

- What if we some loop doesn't terminate?
- Hit infinitely many cutpoint states.
- Program never terminates.
- No problem. Program Partially correct!
  - Any non-terminating program is partially correct.

Subroutine calls with MMRV

- What about subroutine calls?
- Two kinds: non-recursive calls and recursive calls.

Non-recursive Calls in MMRV

- Recall:
  - Rule 2: not(cut(s)) => nextc(s) = nextc(next s)
- Might step through the call instruction and start executing the called method.
- Amounts to in-lining: Violates compositionality.
- Instead, we want to invoke the correctness theorem for the called method.

Recursive Calls in MMRV

- Callee and caller are the same.
  - No problem deciding which routine to verify first -- they're the same!
  - But harder to verify the routine.
- Symbolic executions may step through recursive calls and corresponding returns.

Recursive Calls in MMRV (cont.)

- After popping off a frame we're typically in another call of the same method.
- Need to know everything is okay in that call.
- Need to know that the next return will leave us in an okay state.
- And so on…

Recursive Calls in MMRV (cont.)

- Requires us to characterize the stack frames all the way down.
- Also leads to more cutpoints.
  - Need a cutpoint after the recursive call.
  - Otherwise we'll have to simulate through an arbitrary number of returns.

My Methodology

- I handle both types of subroutine call.
- Also do a lot of things automatically for the user.
- I phrase lots of things in terms of the height of the call stack.
- Mostly, I hide the stack height stuff from the user.

Layered Approach

0. Partial Correctness Result from MMRV
   - Generic machine and generic program.
   - 5 constraints to prove.
1. Partial Correctness Result with Stack Height
   - Still generic machine and generic program.
   - 1 constraint to prove. (Others true by definition.)
2. Machine-specific Result
   - Specific Machine (JVM or AAMP7).
   - Still generic program.
3. Program-specific Result
   - Macro to help generate this.

Level 0

- Recall: Partial correctness follows from 5 constraints about exit, cut, pre, post, assert, etc.
- I reuse partial-correctnes.lisp from MMRV.
- Aside: can use lambdas with :functional-instance to increase the number of parameters on the functions involved.
  - Lets me add stack height parameter, etc.

Level 1

- I start with generic functions that don't use the stack height.
- I build my own functions on top of them that are stack-height aware.
- I define my functions so that all constraints but #3 are true by definition.

Non-stack-aware Functions

- precondition(s0)
- poststate(s0)
  - Not using don't-care specification.
- starting-program-counter()
  - Automatically treated as a cutpoint
  - For Java is always 0.
- non-start-cutpoints ()
  - A list of integers.
- assertion-for-non-start-cutpoints(s0 s)
  - (Assertion for start point will be precondition.)

Requirements on the Operational Semantics

Must specify two things:

- Stack-height(s)
  - Stack means “call stack”, not operand stack.
  - Must increase for calls and decrease for returns.
- Program-counter(s)
  - Location of next instruction to be executed by top routine on call stack.

Exit State Recognizer

```
(defun stack-height-less-than (sh s)
  (< (stack-height s) sh))
```

- Recognizes the state just after we pop off a stack frame for a return.
- Takes a stack-height parameter.
- Not program-specific.

Cutpoint Recognizer

```
(defund my-cutpoint (sh pc-vals s)
  (or (stack-height-less-than sh s)
      (and (equal sh (stack-height s))
           (member (program-counter s) pc-vals))))
```

- Takes a stack-height parameter and a list of cutpoint PCs.
- Not program-specific.
- Constraint 4 was: exit(s) => cut(s).
  - True by definition for my functions.

Full Cutpoint List

```
(defun cutpoints ()
  (cons (starting-program-counter)
        (non-start-cutpoints)))
```

We automatically include starting-program-counter.

My Precondition

```
(defun my-precondition (sh s0 s)
  (and (precondition s)
       (equal sh (stack-height s))
       (equal (program-counter s)
              (starting-program-counter))
       (equal s s0)))
```

- Constraint 1 was: pre(s) => cut(s).
  - True by definition for my functions.

My Postcondition

```
(defun my-postcondition (s0 s)
   (equal s (poststate s0)))
```

- Uses the equality phrasing.

My Assertion

```
(defun assertion-at-cutpoint (s0 s)
   (if (equal (program-counter s)
              (starting-program-counter))
       (and (equal s s0)
            (precondition s0)) ;user's
      (assertion-for-non-start-cutpoints s0 s)))

(defun assertion (sh s0 s)
   (if (stack-height-less-than sh s)
       (equal s (poststate s0))
      (assertion-at-cutpoint s0 s)))
```

- Constraint 5 was: exit(s) ^ assert(s) => post(s).
- Constraint 2 was: pre(s) => assert(s).
- True by definition for my functions.

One Constraint Remains

Constraint 3 was:

cut(s) ^ assert(s) ^ not(exit(s)) => assert(nextc(next s))

My version:

```
(defthm cutpoint-to-cutpoint
  (implies (and (member (program-counter s) (cutpoints))
                (equal sh (stack-height s))
                (assertion-at-cutpoint s0 s)
                (eventually-returns-from-stack-height
                     (stack-height s)
                     (next s)))
           (assertion
              sh
              s0
              (run-until-cutstate k (cutpoints) (next s)))))
```

Proving `cutpoint-to-cutpoint`

- Use the two MMRV symbolic simulation rules.
- New wrinkle: Subroutine calls.
- 3 new symbolic simulation rules.

Simulation Rule 3

```
(defthm simulation-rule-3
  (implies (and (< sh (stack-height s))
                (eventually-returns-from-stack-height sh s))
           (equal (run-until-cutstate sh pc-vals s)
                  (run-until-cutstate sh pc-vals (run-until-return s)))))
```

- Looking for a cutpoint at stack height sh.
- Stack height is now greater than sh.
  - (Have just pushed a frame for a call.)
- Start by running until stack height decreases.
- Then resume looking for a cutpoint.

Simulation Rule 3 (cont.)

```
(defthm simulation-rule-3
  (implies (and (< sh (stack-height s))
                (eventually-returns-from-stack-height sh s))
           (equal (run-until-cutstate sh pc-vals s)
                  (run-until-cutstate sh pc-vals (run-until-return s)))))
```

- Introduces a call to run-until-return.
- The theorem about the subroutine triggers.

One Wrinkle: Termination

- We're doing partial correctness.
- Theorem about the called method has a hypothesis: "It terminates."
- Intuitively okay:
  - Proving partial correctness of caller.
  - So can assume caller terminates.
  - At that point, callee must have terminated too!

Advancing the Termination Fact

- At each cutpoint we assume termination
  - (eventually-returns …)
- But we need that fact later in the simulation, when we get to the call.
  - May be many simulation steps after the cutpoint.
- Advance the termination hypothesis along in a parallel simulation.

Advancing the Termination Fact (cont.)

- Might be several subroutine calls between cutpoints.
- What if termination simulation outruns the real simulation?
- Trick: Leave a copy of the termination fact at each call.
- Special -runner function.
  - Name with -runner is just an alias.
  - Runner function is what advances.
  - Leaves non-runner function at each call.

Advancing the Termination Fact (cont.)

```
(defthm rule-4
  (implies
    (equal sh (stack-height s))
    (equal (eventually-returns-from-stack-height-runner sh s)
           (eventually-returns-from-stack-height-runner sh (next s)))))


(defthm rule-5
  (implies
    (< sh (stack-height s))
    (equal
      (eventually-returns-from-stack-height-runner sh s)
      (and (eventually-returns-from-stack-height sh s)
           (eventually-returns-from-stack-height-runner sh (run-until-return
   s))))))
```

Recursion

- Need to know that subroutine calls are correct when we get to them.
- Non-recursive
  - We verify the called method first.
- That won't work for recursion!

Recursion (cont.)

- Do the same thing as non-recursive case, but do it in an inductive step.
- Induction on number of steps to terminate.
  - Assume calls which terminate within n-1 steps are correct.
  - Show calls which terminate within n steps are correct.

Recursion (cont. 2)

```
(defun-sk correct-for-calls-which-return-within-n-steps (n)
  (forall (s)
  (implies (and (my-precondition (stack-height s) s s)
                (returns-from-stack-height-within-n-steps
                   (stack-height s)
                   n
                   s))
           (postcondition s (run-until-return s)))))
```

Conclusion and Future Work

- Framework to check assertions
  - Now handles subroutine calls.
  - Handles recursion nicely.
  - Little user input required for simple programs.
- Now, how do we get the assertions?
  - From programmer annotations
    - Ex: JML
  - From static analysis tools
    - Ex: Linear invariant detection