Verification Condition Generation via Theorem Proving

John Matthews Galois Connections Inc.
J Strother Moore University of Texas at Austin
Sandip Ray University of Texas at Austin
Daron Vroon Georgia Institute of Technology

Presentation for LPAR 2006

A Sincere Apology

I am sorry I could not be in the conference to give this talk to you personally. My re-entry visa to the United States was unexpectedly refused pending security clearance and background checks and hence I cannot make international travels until the checks are over. I am disappointed at this unfortunate event, and sincerely apologize to the conference organizers and the audience for the inconvenience.

Goals and Motivation

How do we perform efficient mechanical proofs of (assembly or machine level) sequential programs?

There has been two distinct approaches to such reasoning.

- Operational semantics
- Inductive assertions (Floyd-Hoare Style)

Each method has its own advantages and disadvantages.

Our Motivation:

Effectively incorporate the advantages of these two different proof strategies without incurring the disadvantages.

Program Correctness

If the program is executed from a machine state satisfying some precondition, then the state at exit satisfies a requisite postcondition.

Partial correctness:

If the program is executed from a machine state satisfying some precondition **and if it exits** then the state at the first exit satisfies the postcondition.

Total correctness:

If the program is executed from a machine state satisfying some precondition then **it exits and** the state at the first exit satisfies the postcondition.

Operational Semantics

The meaning of a program is defined by its effect on the state vector. --- John McCarthy, 1962

- A state (including the program) is an object (*n*-tuple) in the logic.
- The semantics is specified by defining a language interpreter.
  - *step(s)* returns the state after executing one instruction from state *s*.
- Executions are specified by iterated execution of the interpreter.
  - *run(s,n)* = **if** *(n=0)* **then** *s* **else** *run(step(s), n-1)*
- Program correctness is stated in terms of the interpreter.
  - $\forall s,n: pre(s) \wedge exit(run(s,n)) \Rightarrow (\exists m<=n\ exit(run(s,m) \wedge post(run(s,m)))$

To prove it we use a theorem prover for the logic.

Inductive Assertions

Developed by Floyd and Hoare in the 1960s.

- Program is annotated by assertions at certain cutpoints.
  - Assertions are formulas in a given logic.
- A VCG crawls over the annotated program to generate a collection of proof obligations (VCs).
  - VCs are formulas in the logic.
  - The VCG is typically implemented outside the logic.
- VCs are discharged by a theorem prover for the logic.

We require a theorem prover and a VCG.

Strengths and Weaknesses

Operational Semantics

- Models are often more concrete.
- Can be executed and validated against actual designs.
- All the reasoning can be done in one logic with one trusted tool (a theorem prover).

But

- Traditional code correctness proofs have been cumbersome.
  - One defines stepwise global invariants or a precise characterization of the number of machine steps to exit (clock functions).

Strengths and Weaknesses

Inductive Assertions

- Factors out static analysis so that the user can focus on key algorithmic invariants.
  - Hence individual code proofs are simpler.

But

- Implementing a VCG is complicated and error-prone.
  - Often VCGs do on-the-fly formula simplification (theorem proving).
  - The semantics of the language is not written out formally but indirectly encoded in the VCG implementation.

Getting the Benefit of Both

Work combining inductive assertions with operational semantics has traditionally involved implementing a VCG for the language and verifying it against the operational model.

We want to verify operationally modeled programs, but

- We do not want to annotate program points other than cutpoints.
- We do not want to write clock functions.

And

- We do not want to implement (or verify) a VCG.

We want to be able to

- Do both partial and total correctness proofs.
- Afford compositionality for verified subroutines.
  - Allow subroutines to be verified once rather than at every call-site while verifying the callers.
- Handle recursive subroutines effectively.

VCG via symbolic simulation

What is the guarantee provided by a VCG?

Let p be an annotated cutpoint of the program. Suppose the assertions hold when control reaches the cutpoint p. Let q be the next subsequent cutpoint encountered in the program execution. Then q also satisfies the corresponding assertions.

A VCG guarantees this by crawling over the annotated program and performing static analysis.

We achieve the same effect by symbolic simulation of the operational semantics without implementing a VCG.

The Basic Method

Assume we are given the functions step, pre, post, together with predicates cut and assert.

Define

csteps(s,i) = if *cut(s)* then *i* else *csteps(step(s),i+1)*

Why is this definition consistent?

The definition is tail-recursive and hence consistent in any logic providing a non-constructive choice operator together with the ability to define recursive equations.

(For a detailed discussion of the issue, refer to Manolios and Moore's 2003 paper "**Partial Functions in ACL2**".)

The Basic Method

csteps(s,i) = if *cut(s)* then *i* else *csteps(step(s),i+1)*

The subsequent cutpoint from a state s is defined by:

nextc(s) = *run(s,csteps(s,0))*, if *cut(run(s,csteps(s,0))*

nextc(s) = *d*, otherwise

where

d = (SOME *s* such that ¬*cut(s)*)

- Easy to prove:
  - **SSR1**: ¬*cut(s)* ⇒ *nextc(s)* = *nextc(next(s))*
  - **SSR2**: *cut(s)* ⇒ *nextc(s)* = *s*
- **SSR1** and **SSR2** cause the theorem prover's simplifier to symbolically simulate machine until the next cutpoint is reached.
  - Symbolic simulation may diverge if no cutpoint is reachable.

Verification Conditions

- These verification conditions imply partial correctness

  C1: $pre(s) \Rightarrow assert(s)$

  C2: $assert(s) \Rightarrow cut(s)$

  C3: $exit(s) \Rightarrow cut(s) \wedge$

  C4: $assert(s) \wedge exit(s) \Rightarrow post(s)$

  C5: $assert(s) \wedge \neg exit(s) \Rightarrow assert(nextc(step(s)))$

- Given a mapping m from the machine states to a well-founded set, this additional verification condition implies total correctness.

  C6: $assert(s) \wedge \neg exit(s) \Rightarrow m(nextc(next(s))) < m(s)$

The verification conditions can be proven by symbolic simulation using the rules **SSR1** and **SSR2**.

- **The process mimics a forward VCG.**

Basic Code Proof

The fact that the verification conditions are sufficient to prove correctness can be mechanically checked independent of the actual definitions of next, pre, post, etc.

- The mechanical proof has been done in both ACL2 and Isabelle theorem provers.

Given a concrete program, we can then

- Define nextc
- Prove SSR1 and SSR2 automatically.
- Use SSR1 and SSR2 as symbolic simulation rules to prove the verification conditions.
- Instantiate the correctness theorems above to establish correctness.

In ACL2 we implemented a macro to carry out these steps.

- No user input other than the annotations for inductive assertions is necessary.
- We derive the correctness theorems based on the operational semantics.

Subroutines and Compositionality

Suppose a procedure P invokes another procedure Q.

C5: $assert(s) \wedge \neg exit(s) \Rightarrow assert(nextc(next(s)))$

Symbolic simulation from a state s in P might encounter an invocation of Q, resulting in symbolic execution of Q.

- If Q has already been verified we will want to skip past such invocations to the exitpoint of Q.

We handle verified subroutines compositionally.

Non-interference Requirement

No cutpoint of P occurs inside an invocation of Q.

Thus

(Non-interference Lemma): If s is poised to invoke Q then the next subsequent cutpoint of P after s is the same as the next subsequent cutpoint of P after the exitpoint s' of Q.

Compositionality

Non-interference lemma is independent of the concrete definitions of P or Q.

- Has been mechanically verified with ACL2 from non-interference requirement and assumption of correctness of Q.

Enables us to skip past a verified subroutine invocation while verifying its caller.

Frame Problem

But

How do we guarantee that we can continue symbolic simulation of P?

- Q might have smashed some of P's state, say its call stack.
- We must ensure (in addition to Q's correctness) that it does not modify unauthorized parts of the machine state.

This is the classic frame problem!

To compositionally deal with the frame problem, Q's postcondition must be strong enough that we can deduce the control flow of P after its exit.

- We essentially deal with it by requiring that the postcondition must specify the global effects of the execution of Q on the state components visible to P.
  - If Q uses some scratch space during then the modification of such space need not be specified.

Recursive Procedures

How do we compositionally verify a procedure which invokes itself?

- No verified callee!!

We achieve this by induction.

Key insight: If the caller terminates then the callee must terminate in a smaller number of steps.

Proof Outline:

There is no caller cutpoint inside the callee invocation. If the caller exits, the exitpoint is a cutpoint. This is possible only after the callee has exited.

The use of compositionality in recursive procedures often drastically simplifies code proofs.

See the paper for an illustrative example.

Applications

The paper discusses several applications of the method on two operational machine models, TINY and M5.

- TINY is a small stack-based machine modeled in ACL2 by Rockwell-Collins.
- M5 models a fairly large fragment of the JVM.

The method has been successfully used to verify small to medium-sized assembly-level programs in ACL2.

- Functional correctness of JVM bytecodes for a CBC-mode encryption and decryption algorithm (600 bytecodes).

Only non-trivial user inputs are the loop invariants for the associated procedures.

Applications

The method is being deployed and used by others

- At NSA, proof of multiplier on Mostek 6502 processor using our macro. (Legato)
- At Galois Connections, verification of AAMP7 programs using our macro. (Pike)
- At Rockwell Collins, use of the method (with independent implementation) for verification of AAMP7 assembly language programs. (Hardin, Smith, Young)
- At Stanford University, developing infrastructure for Java program verification using the method (with independent implementation). (Smith)
- At University of Cambridge, porting the method to HOL4 for reasoning about ARM assembly language programs. (Fox)

Discussion

The method is very simple conceptually.

- Tail-recursive functions for symbolic simulation
- Frame conditions and non-interference for composition
- Induction for recursive procedure

But the ideas seem to be powerful and robust in practice.

- Just configure the theorem prover to obtain the advantage of the VCG without implementing one.

Discussion

But wait….

Isn't the infrastructure (macros and all) just a VCG?

Maybe. But with some **very important differences**.

- It is state-based, not formula based.
- You do not need to trust the "VCG" code. All that you trust is the theorem prover.
  - Yet you get the benefits of inductive assertions and the concreteness and executability of operational semantics.
- Any operational semantics can be plugged into this "VCG".

Discussion

Of course this is just a start.

- Practical VCGs might perform substantial static analysis to reduce (for example) case blow-up on the join points of conditionals.
  - Can we do such analysis in this dynamic setting?
  - We believe this might be possible by appealing to ACL2's so-called "meta rules", but that is for the future.

Related Work

- **Inductive assertions have been applied in several code verification projects, usually with a VCG.**
  - Praxis proofs of Spark programs
  - ESC/Java, Java Certifying Compiler
- **Verification of VCGs has been an active topic of research in the HOL and PVS communities.**
- **In ACL2:**
  - Moore observed in 2003 that tail-recursive equations can be used to simulate VCG reasoning for partial correctness proofs.
  - Matthews and Vroon in 2004 came up with a related method for proving termination.

  This work unifies these two ideas into a single framework and extends the method to achieve compositionality and handle recursive procedures.
- **There are many parallels between our work and Foundational PCC approaches.**

See the paper for details.

Conclusion

With an expressive theorem prover and operational semantics

- One can simulate inductive assertions proofs for deriving partial and total correctness.
- One can apply the methods compositionally.
- One can reason about recursive procedures efficiently

All this without implementing (let alone verifying) a VCG for the language.

We are planning to apply this method on larger code proofs.

A Key target application: The SHADE Verifying Compiler Project (jointly by Galois Connections Inc. and Rockwell Collins Inc.)

Questions?

Please feel free to email us questions at:

matthews@galois.com

moore@cs.utexas.edu

sandip@cs.utexas.edu

vroon@cc.gatech.edu

Again, I apologize for not being able to give this talk in person.