A Mechanical Analysis of Program Verification Strategies

Sandip Ray

Department of Computer Sciences
University of Texas at Austin

Email: sandip@cs.utexas.edu
web: http://www.cs.utexas.edu/users/sandip

Joint work with **Warren A. Hunt Jr**, **John Matthews**, and **J Strother Moore**

Program Verification

Program verification entails proving the following theorem.

Correctness Theorem: If the program is initiated in a machine at a state satisfying a certain **precondition**, then the state reached on **termination** of the program satisfies a desired **postcondition**.

Program verification is one of the earliest and most fertile application areas of formal reasoning, and mechanical theorem proving.

Summary

We formally analyze three verification strategies for deterministic sequential programs modeled with operational semantics.

- Stepwise Invariants
- Clock functions
- Inductive Assertions

We show that each strategy is both sound and complete.

Completeness means that if there is *any* correctness proof of the program then there is one using any of the strategies.

- The completeness result has been surprising (at least to me and others I have shown it).

But the proofs are not mathematically deep!

- A careful formalization of the questions essentially leads to the answers.

Talk Outline

- **Operational Semantics and Correctness Theorems**
- Proof Strategies
- Analysis of Strategies
- Discussion and Conclusion

Operational Semantics

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

- Model states of the machine executing the program as objects (n-tuples) in a logic.
  - Two special state components are the *pc* and the *program* being executed.
- A program is an object, e.g., a list of instructions.
  - The semantics of a program is given by defining a **language interpreter**, which is a function on states.

Modeling with Operational Semantics

- Function $\mathit{effect}(s, i)$ specifies the state obtained by executing instruction i on state s.
- Let $curinst$ be the instruction in $\mathit{program}(s)$ pointed to by $\mathit{pc}(s)$. Then $\mathit{step}(s)$ is defined to be $\mathit{effect}(s, curinst)$.
- We define the concept of running the machine for n steps from state s by the function *run*:

$$\mathit{run}(s, n) \triangleq \begin{cases} s & \text{if } \mathit{zp}(n) \\ \mathit{run}(\mathit{step}(s), n - 1) & \text{otherwise} \end{cases}$$

- Predicate $\mathit{exit}(s)$ holds if s is a terminating state.

Correctness Statement

There are two formal notions of correctness, **Partial** and **Total**.

Partial Correctness:

For any state p satisfying the *pre* condition, if an *exit* state q is reachable from p, then the *post* condition holds for the first such *exit* state.

$$\begin{aligned}\forall s, n : \mathit{pre}(s) \;&\wedge \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)) \\ &\wedge (\forall m : \mathit{natp}(m) \wedge (m < n) \Rightarrow \neg \mathit{exit}(\mathit{run}(s, m))) \\ &\Rightarrow \mathit{post}(\mathit{run}(s, n))\end{aligned}$$

Total Correctness:

Total Correctness = **Partial correctness** + **Termination**

Termination:

$$\forall s : \mathit{pre}(s) \Rightarrow (\exists n : \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)))$$

Each of the formulas can be expressed in the ACL2 logic.

Talk Outline

- Operational Semantics and Correctness Theorems
- **Proof Strategies**
- Analysis of Strategies
- Discussion and Conclusion

A Running Example

1: X:=0
2: Y:=10
3: if (Y $\leq$ 0) goto 7
4: X:=X+1
5: Y:=Y-1
6: goto 3
7: ....

- $\mathit{pre}(s) \triangleq \mathit{prog\text{-}loaded}(s) \wedge (\mathit{pc}(s) = 1)$
- $\mathit{post}(s) \triangleq (X(s) = 10)$
- $\mathit{exit}(s) \triangleq (\mathit{pc}(s) = 7)$

Stepwise Invariants

[Origin somewhat unknown, but work of **Goldstein and von Neumann (1961)**, and **Turing (1949)** can be viewed as instances of this approach.]

Partial Correctness:

Construct a predicate *inv* such that the following are theorems:

$I1$: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{inv}(s)$

$I2$: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{inv}(\mathit{step}(s))$

$I3$: $\forall s : \mathit{inv}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

Total Correctness:

Additionally define m such that the following are theorems:

$I4$: $\forall s : \mathit{inv}(s) \Rightarrow \mathit{o\text{-}p}(m(s))$

$I5$: $\forall s : \mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow m(\mathit{step}(s)) \prec m(s)$

Conditions $I2$ and $I5$ require that *inv* and m explicitly characterize each reachable state.

Stepwise Invariants

The predicate *inv* must characterize every *pc* value.

{**T**}	1:	X:=0
{**(X=0)**}	2:	Y:=10
{ *natp*(**Y**) $\wedge$(**X+Y=10**) }	3:	if (Y $\leq$ 0) goto 7
{ *natp*(**Y**) $\wedge$(**Y $>$0**) $\wedge$(**X+Y=10**)}	4:	X:=X+1
{ *natp*(**Y**) $\wedge$(**Y $>$0**) $\wedge$ (**X+Y=11**)}	5:	Y:=Y-1
{ *natp*(**Y**) $\wedge$ (**X+Y=10**)}	6:	goto 3
{**(X=10)**}	7:	...

A similar comment can be made about m in case of total correctness proof.

Clock Functions

[Widely used in Boyer-Moore community, usually for total correctness.]

Total Correctness:

Construct a function *clock* such that the following are theorems:

$C1$: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{natp}(\mathit{clock}(s))$

$C2$: $\forall s, n : \mathit{pre}(s) \wedge \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{clock}(s) \leq n$

$C3$: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

$C4$: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

Partial Correctness:

Weaken $C1$, $C3$ and $C4$ to add $\mathit{exit}(\mathit{run}(s, n))$ in the hypotheses.

$C1'$: $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{natp}(\mathit{clock}(s))$

$C3'$: $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{exit}(\mathit{run}(s, \mathit{clock}(s)))$

$C4'$: $\forall s, n : \mathit{pre}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{post}(\mathit{run}(s, \mathit{clock}(s)))$

Clock functions characterize the number of steps from initiation to exit.

- **But this is a characterization of time complexity!**

Clock Functions

```
1:   X:=0
2:   Y:=10
3:   if (Y ≤ 0) goto 7
4:   X:=X+1
5:   Y:=Y-1
6:   goto 3
7:   ....
```

We define the clock function by looking at the control structure of the program.

$$\textit{loop-taken}(s) \triangleq (\textit{pc}(s) = 3) \wedge \textit{natp}(Y(s)) \wedge (Y(s) > 0) \wedge \textit{prog-loaded}(s)$$

$$\textit{loop-clock}(s) \triangleq \begin{cases} 0 & \textit{if } \neg\textit{loop-taken}(s) \\ 4 + \textit{loop-clock}(\textit{run}(s, 4)) & \text{otherwise} \end{cases}$$

$$\textit{clock}(s) \triangleq 2 + \textit{loop-clock}(\textit{run}(s, 2)) + 1$$

Inductive Assertions

This is the most commonly advocated approach to program verification.

[Based on early observations by **Goldstein and von Neumann (1961)**, and **Turing (1949)**, later refined and generalized by classic works of **Floyd (1967)**, **Hoare (1969)**, **Manna (1969)**, and **Dijkstra (1975)**.]

- Annotate program with **assertions** at certain **cutpoints**.
- A VCG derives from these a set of **Verification Conditions** (VCs).
- The VCs are proven by a theorem prover.

Key Features:

- Requires annotations only at cutpoints.
- Requires both a VCG and a theorem prover.

(Aside) **King (1969)** wrote the first mechanized VCG.

Inductive Assertions

Recall the stepwise invariant proof.

$\{\mathbf{T}\}$	1:	X:=0
$\{\mathbf{(X=0)}\}$	2:	Y:=10
$\{\ \mathit{natp}\mathbf{(Y)} \wedge \mathbf{(X+Y=10)}\ \}$	3:	if (Y $\leq$ 0) goto 7
$\{\ \mathit{natp}\mathbf{(Y)} \wedge \mathbf{(Y\ >0)} \wedge \mathbf{(X+Y=10)}\}$	4:	X:=X+1
$\{\ \mathit{natp}\mathbf{(Y)} \wedge \mathbf{(Y\ >0)}\ \wedge\ \mathbf{(X+Y=11)}\}$	5:	Y:=Y-1
$\{\ \mathit{natp}\mathbf{(Y)} \wedge\ \mathbf{(X+Y=10)}\}$	6:	goto 3
$\{\mathbf{(X=10)}\}$	7:	...

Inductive Assertions

In inductive assertions, we only require annotations at the cutpoints, namely loop tests and program entry and exit.

{**T**}	1:	X:=0
	2:	Y:=10
{ *natp*(**Y**) ∧(**X+Y=10**) }	3:	if (Y ≤ 0) goto 7
	4:	X:=X+1
	5:	Y:=Y-1
	6:	goto 3
{**(X=10)**}	7:	...

A VCG explores the paths in the annotated program to generate the VCs.

- Language semantics encoded in VCG implementation as formula transformation.

For the path $1 \rightarrow 2 \rightarrow 3$ the obligation is $\mathtt{T} \Rightarrow \mathit{natp}(10) \wedge (0 + 10) = 10$.

- The obligations are discharged by a theorem prover.

Inductive Assertions in Operational Semantics

The inductive assertions method can be used with operational semantics, without a VCG.

- It is possible to emulate VCGs by symbolic simulation on operational models.

Possibility first suggested by a cute method due to **Moore (2003)**, which could handle partial correctness.

- Consolidated by **Matthews, Moore, Ray, Vroon (2006)** to handle total correctness.
- Further consolidated by independent efforts of **MMRV** and **Hardin, Smith, Young (2006)** to allow compositionality, effective reasoning about recursive procedures, etc.
- Recently significantly automated for crypto proofs about by **Smith and Dill (2007)**.

How to do all this is not part of this talk, but we'll use the **MMRV** formalization of inductive assertions.

- **HSY** and **SD** use a variant with trivial difference.

Formalizing Inductive Assertions

Suppose we are given predicates *cut* and *assert* specifying the cutpoints and corresponding assertions.

Define:

$$\mathit{csteps}(s, i) \triangleq \begin{cases} i & \text{if } \mathit{cut}(s) \\ \mathit{csteps}(\mathit{step}(s), i+1) & \text{otherwise} \end{cases}$$

Crucial Observation:

- Definition of *csteps* is tail-recursive
- **Manolios and Moore (2003)** show that any tail-recursive equation can be admitted in ACL2 by encapsulation.

(Aside) The admissibility of tail-recursive equations was also used in Moore's initial formulation.

Formalizing Inductive Assertions

Suppose we are given predicates *cut* and *assert* specifying the cutpoints and corresponding assertions.

Define:

$$\mathit{csteps}(s,i) \triangleq \begin{cases} i & \text{if } \mathit{cut}(s) \\ \mathit{csteps}(\mathit{step}(s), i+1) & \text{otherwise} \end{cases}$$

$$\mathit{nextc}(s) \triangleq \begin{cases} \mathit{run}(s, \mathit{csteps}(s,0)) & \text{if } \mathit{cut}(\mathit{run}(s, \mathit{csteps}(s,0))) \\ \mathbf{d} & \text{otherwise} \end{cases}$$

where $\mathit{cut}(\mathbf{d}) \Leftrightarrow (\forall s : \mathit{cut}(s))$

For any state s, $\mathit{nextc}(s)$ returns the closest reachable cutpoint from s (if such a cutpoint exists), otherwise it does not return a cutpoint.

Formalizing Inductive Assertions

Partial Correctness:

Given an operational model and a set of cutpoints specified by *cut* **(that include** *pre* **and** *exit* **states), define** *assert* **such that the following are theorems.**

$V1$: $\forall s : \mathit{pre}(s) \Rightarrow \mathit{assert}(s)$

$V2$: $\forall s : \mathit{assert}(s) \Rightarrow \mathit{cut}(s)$

$V3$: $\forall s : \mathit{exit}(s) \Rightarrow \mathit{cut}(s)$

$V4$: $\forall s : \mathit{assert}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$

$V5$: $\forall s, n : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{step}(s)))$

Total Correctness:

Additionally define a function *rank*, **weaken** $V5$ **to** $V5'$, **and prove** $V6$ **and** $V7$.

$V5'$: $\forall s : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{step}(s)))$

$V6$: $\forall s : \mathit{assert}(s) \Rightarrow \mathit{o\text{-}p}(\mathit{rank}(s))$

$V7$: $\forall s : \mathit{assert}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{nextc}(\mathit{step}(s))) \prec \mathit{rank}(s)$

Talk Outline

- Operational Semantics and Correctness Theorems
- Proof Strategies
- **Analysis of Strategies**
- Discussion and Conclusion

What do We Analyze?

Soundness of a strategy:
The obligations involved logically imply the correctness statements.

Completeness of a strategy:
If a program is partially (totally) correct, then the corresponding proof obligations for each strategy can be met.

- Often called **Cook Completeness**.

In this talk we just outline the completeness proofs, since they are more surprising!

Completeness: Some elaboration

We need to answer the following questions:

Suppose a program is partially (totally) correct. Then can we **always**

- define a *clock*?
- define the appropriate *inv* and m?
- given a predicate *cut* recognizing the cutpoints, define *assert* and *rank*?

The answer in each case is **Yes**, and in each case the proof is essentially trivial!

Completeness of Clock Functions

Given: The partial (resp., total) correctness theorem.

We must define a *clock* that returns the number of steps from a *pre* state to the first *exit*.

$$\begin{array}{lcl} \mathit{esteps}(s,i) & \triangleq & \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{esteps}(\mathit{step}(s), i+1) & \text{otherwise} \end{cases} \\ \mathit{clock}(s) & \triangleq & \mathit{esteps}(s,0) \end{array}$$

Exercise:

Use this definition to prove

$$\begin{array}{l} \forall s, n : \mathit{exit}(\mathit{run}(s,n)) \Rightarrow \\ \qquad \mathit{exit}(\mathit{run}(s, \mathit{clock}(s))) \wedge \\ \qquad \mathit{natp}(\mathit{clock}(s)) \wedge \\ \qquad (\mathit{natp}(n) \Rightarrow (\mathit{clock}(s) \leq n)) \end{array}$$

(Aside) The proof is not completely trivial. Can you see why?

Completeness of Stepwise Invariants

Given: The partial (resp., total) correctness theorem.

Assume wlog that we have the corresponding clock function proof, with the weird *clock*.

$$
\begin{array}{lcll}
\mathit{inv}(s) & \triangleq & (\exists p, m: & \mathit{pre}(p) \wedge \\
 & & & \mathit{natp}(m) \wedge \\
 & & & (s = \mathit{run}(p, m)) \wedge \\
 & & & ((\exists \alpha : \mathit{exit}(\mathit{run}(p, \alpha))) \Rightarrow (m \leq \mathit{clock}(p)))) \\
m(s) & \triangleq & \mathit{clock}(s) &
\end{array}
$$

Completeness of Stepwise Invariants

Given: The partial (resp., total) correctness theorem, and the predicate *cut*.

Assume wlog that we have the corresponding clock function proof, with the weird *clock*.

- Define the weird *inv*.

$$\begin{aligned} \mathit{assert}(s) &\triangleq \begin{cases} \mathit{inv}(s) & \text{if } \mathit{cut}(s) \\ \texttt{NIL} & \text{otherwise} \end{cases} \\ \mathit{rank}(s) &\triangleq \mathit{clock}(s) \end{aligned}$$

Talk Outline

- Operational Semantics and Correctness Theorems
- Proof Strategies
- Analysis of Strategies
- **Discussion and Conclusion**

Why Care about Formalizing Strategies?

The proofs are really trivial, but only after we carefully formalized the correctness statements and the essence of each strategy.

Without the formalization it is easy to design flawed strategies.

A Flawed Strategy!

The following proof strategy was suggested by **Manolios and Moore (2003)** as a way to prove "partial correctness".

Define:

$$\mathit{stepw}(s) \triangleq \begin{cases} s & \text{if } \mathit{halted}(s) \\ \mathit{stepw}(\mathit{step}(s)) & \text{otherwise} \end{cases}$$

where $\mathit{halted}(s) \triangleq (\mathit{step}(s) = s)$.

Define: $(s_0 \hookrightarrow s) \triangleq (\mathit{stepw}(s_0) = \mathit{stepw}(s))$

Let $\mathit{modify}(s)$ represent the modification of s after executing the program of interest. Then the the strategy is to prove:

$$\mathit{pre}(s) \Rightarrow (s \hookrightarrow \mathit{modify}(s))$$

Can you see the problem with this strategy?

Random Remarks

The proofs of soundness and completeness are generic.

- Done in ACL2 using encapsulation.

For a specific set of concrete definitions of *step*, *pre*, *post*, etc., we can translate proofs from one strategy to another by functional instantiation.

- I have a macro that can translate between strategies.

The macro can be used to go back and forth between strategies while developing correctness proofs of a large system.

- A component can be verified with that strategy that is best suited for it.
- Potentially useful in developing generic verification frameworks.
- I have occasionally wanted this while doing cutpoint proofs.

However, I have never used the translation in practice yet.

- This work was principally a result of curiosity.

Conclusions

We proved that the three strategies are logically equivalent in spite of differences, and each is in fact complete.

- At least in a logic allowing recursive definitions and quantification.

Perhaps indicates that the complexity in program verification does not arise from the proof strategy used but rather the inherent complexity involved in reasoning about correctness of code.

- Maybe an automation breakthrough will also carry over among strategies?

The immediate takeaway message is perhaps that it is very useful sometimes to reason about and think in terms of quantification, especially when developing generic techniques and strategies.

Acknowledgements

Thank You!

Special Thanks To:

- **Jeff Golden** for significant early discussions that clarified my understanding of operational semantics and program correctness.
- **Matt Kaufmann** for many discussions on ACL2 logic and quantifiers, and in particular for a comment made during a presentation of some preliminary results in 2004, that started me on track for the completeness results in the first place.