Application of Logic to Digital System Design

Paul S. Miner
May 10, 1995

Supporting text

The purpose of this talk is to illustrate what activities comprise the application of formal methods to digital system design. The examples given in this presentation are all drawn from real applications of formal methods.
A much better introduction to the issues involved is:
John Rushby: *Formal Methods and their Role in the Certification of Critical Systems*, Technical Report CSL-95-1, Computer Science Laboratory, SRI International, March 1995.
http://www.csl.sri.com/csl-95-1.html

Outline

- Formal Specification
- Formal Proof Activities
- Degree of Rigor
- Obstacles to Getting Started

Supporting text

- The first part of the talk will address some of the issues involved in developing a formal specification. This will include illustrations of different styles of specification using examples drawn from real applications.
- The next section of the talk will address some of the different styles of refinement and proof in a formal development process.
- Formal methods can be applied with varying degrees of rigor. A taxonomy of levels of rigor will be presented.
- There are a number of obstacles for first time users to overcome. This section of the talk will enumerate some of the obstacles; hopefully, the remainder of this workshop will provide insights for overcoming them.

Formal Specification

Formal Specification: Use of notations derived from formal logic to describe

- *assumptions* about the world in which a system will operate
- *requirements* that the system is to achieve
- the intended *behavior* of the system

Styles of Specification Different approaches are used for these descriptions:

- *Properties*—enumeration of assumptions and requirements
- *Functions*—express desired behavior or design descriptions
- *State-machines*—express desired behavior or design descriptions
- ...

Assumptions at one level become requirements at a lower level.

Supporting text

There are a number of different forms a formal specification can take. Often the style of specification will reflect the problem domain. Specifications describe some combination of *assumptions*, *requirements*, and *intended behavior* of the system. The description of the behavior can be high-level and abstract, or it may reflect a level in a design heirarchy.

Example of Property-Based Specification (Fault-tolerant clock synchronization)

1. There is a $\rho \ll 1$ such that for any clock C_p that is non-faulty during the interval from t_1 to t_2:

$$(1-\rho)(t_2 - t_1) \leq C_p(t_2) - C_p(t_1) \leq (1+\rho)(t_2 - t_1)$$

2. There is a δ such that if clocks C_p and C_q are non-faulty at time t, then:

$$|C_p(t) - C_q(t)| < \delta$$

Supporting text

The first property states that the function C_p is a clock, that is, it provides a reasonable measure of the passage of time. The second states that any two non-faulty clocks in the collection are synchronized.

Example of Functional Specification (Floating-point Operations)

IEEE floating-point arithmetic requires that each arithmetic operation *be performed as if it first produced a result correct to infinite precision and unbounded range, and then coerced this result to fit in the destination's precision.* [ANSI/IEEE STD 854-1987]

Supporting text

This is an example of an informal requirement. The easiest way to define arithmetic operations is to use a function based specification. With a sufficiently expressive specification language, this is a rather straightforward exercise.

Floating-Point Operations (DECLARATIONS)

$$
\begin{aligned}
FP &= \text{the set of floating-point numbers} \\
Fin &= \text{the set of finitely valued floating-point numbers, } Fin \subset FP \\
M &= \text{the set of rounding modes} \\
\mathbf{R} &= \text{the set of real numbers} \\
fin1, fin2 &\in Fin \\
mode &\in M
\end{aligned}
$$

`fp_to_real`: $Fin \rightarrow \mathbf{R}$
`real_to_fp`: $R \times M \rightarrow FP$

Supporting text

The definition of floating-point operations assume that we have the following:

- A set of objects that represent the floating-point numbers
- A proper subset of the floating-point numbers that represents the finitely valued floating-point numbers
- A finite set of rounding modes
- Real Numbers
- Function `fp_to_real` is defined to convert any finitely valued floating-point number to a real number.
- Function `real_to_fp` takes two arguments: a real number and a rounding mode. It coerces the real into a floating-point representation in accordance with the rounding mode.

Floating-Point Addition (Partial Specification)

For floating-point addition of two finitely valued arguments, $fin1$ and $fin2$, the expression

$$\texttt{fp\_to\_real}(fin1) + \texttt{fp\_to\_real}(fin2)$$

defines the infinitely precise result.

Supporting text

We can then define arithmetic operations *correct to infinite precision and unbounded range* using the corresponding operator in real arithmetic.

Floating-Point Addition (CONTINUED)

The formal specification of floating-point addition for finitely valued arguments is:

$$\texttt{fp\_add\_finite}(fin1, fin2, mode) \;\hat{=}\; \texttt{real\_to\_fp}(\ (\texttt{fp\_to\_real}(fin1) + \texttt{fp\_to\_real}(fin2)),\ mode)$$

Supporting text

This specification states precisely what a realization of floating-point addition for finite arguments must compute. It does not in any way determine how this function may be implemented. This definition also formally captures, in a natural way, the informally stated requirements of the standard.

State-machine Specification

Design layers formalized as state machines

- State represents memory contents and hardware status
- Transition *function* defines state transitions

Interpretation maps lower level states into higher level states

$$
\begin{tikzcd}
s_i \arrow[r, "N_i"] & t_i \\
s_{i+1} \arrow[u, "Map"] \arrow[r, "N_{i+1}"] & t_{i+1} \arrow[u, "Map"']
\end{tikzcd}
$$

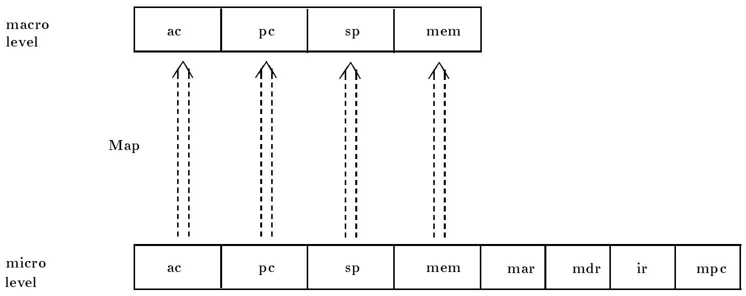
Need to show that diagram "commutes" to establish that layer $i+1$ correctly implements layer i:

$$Map(N_{i+1}(s_{i+1})) = N_i(Map(s_{i+1}))$$

Supporting text

State-machine based specifications are useful for describing the behavior of digital systems. Systems are specified as a collection of states and a transition function. Different levels in a design hierarchy are related by abstraction mappings. Verification consists of showing that diagrams commute.

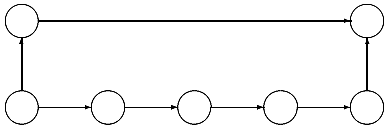
Abstraction Mappings (Data Abstraction)



Supporting text

This slide illustrates a data abstraction similar to what would be used in a verification of a microprocessor. The macro-level defines the programmers view of the state; this include the accumulator, program counter, stack pointer and memory. The micro-level introduces some additional storage locations necessary for a correct realization of the macro-level specification; this includes the memory address register, memory data register, the instruction register, and microprogram counter.

Temporal Abstraction



Supporting text

This diagram illustrate a simple temoral abstraction. The high-level state transition function is realized using a sequence of lower level state transitions.

Formal Proof Activities

Use of methods from formal logic to

1. *analyze* specifications for certain forms of consistency, completeness
2. *prove* that specified behavior will satisfy the requirements, given the assumptions
3. *prove* that a more detailed design *implements* a more abstract one

Supporting text

A verification effort can consist of a number of different proof activities. There are a number of formal techniques for analyzing specifications. These help provide assurance that the specification describes what we intend. Proof techniques are useful for showing that specified behavior exhibits desired prpoerties. Similarly, proof can be used to relate different levels in a design hierarchy.

Formal Analysis of a Specification (1)

One property we might want to prove about the functions `real_to_fp` and `fp_to_real` from the floating-point addition specification is that for every finitely valued floating-point number and every rounding mode:

`real_to_fp(fp_to_real(`fin`),`$mode$`)` $= fin$

Supporting text

In order to ensure that the specification captures what we intend, we postulate certain properties that should be true of the specification. Formal techniques allow us to prove a spcification posseses these properties. This is sometimes referred to as formal validation.
The example presented here states that the function converting real numbers to floating-point representations should invert `fp_to_real`, irrespective of the rounding mode. As stated, the property is not true. Additional restrictions are required, because the IEEE-854 floating-point stadard allows redundant encodings.

Verification of Fault-Tolerant Algorithms (2)

Top-level: Properties that algorithm should possess

Lower-level: Abstract description of the algorithm and underlying assumptions

Prove: The algorithm satisfies desired properties given the assumptions

Supporting text

As an example, the top-level requirement for fault-tolerant clock synchronization is an enumeration of properties. The verification of an algorithm consists of showing that the algorithm, given certain assumptions about the environment, ensures all the properties.

Design Verification (3)

Top-level: Abstract description of system (and assumptions)

Lower-level: Detailed description of system (and assumptions)

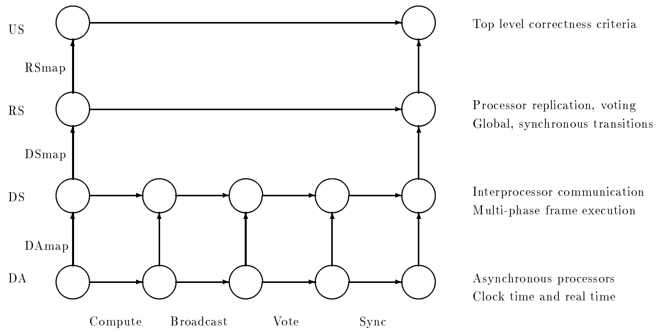
Prove: The detailed system description has the same behavior as the abstract description given the assumptions and an abstraction function relating the two systems.

Supporting text

The next example will give a high-level overview of design verification.

Design Refinement and Proof (Example: Reliable Computing Platform)

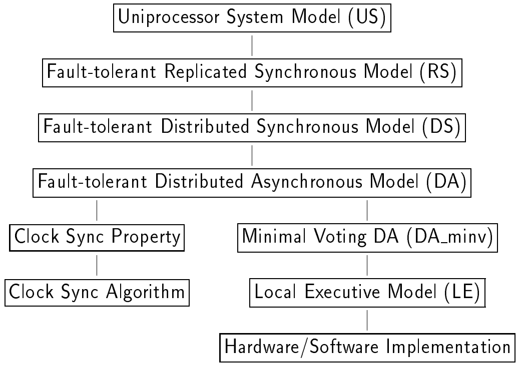
Single-frame state transition divided into four phases



Supporting text

This diagram illustrates four levels in the design hierarcy of the Reliable Computing Platform. The top-level specification provides the applications programmer a high-level view of a single, ultra-reliable computer. Lower levels in the diagram introduce the necessary replication and redundancy management to mask physical failures. These levels all use the state-machine approach.

Hierarchical Specification of the Reliable Computing Platform



Supporting text

This slide shows all the levels in the RCP design verification effort. The top four levels are the same as the previous slide. This slide also illustrates how the clock synchronization is addressed. The top four levels assume the clock synchronization properties. At a lower level, a proof is given showing that an algorithm guarantees the properties. This hierarchy illustrates how different techniques may be employed in one large verification effort.

Correctness of Specification

How do we know that a specification is what we intend?

- At intermediate levels in the design hierarchy, the specification is a refinement of higher-level requirements
  - If the higher levels are related by proof, then the requirements at this level are sufficient to ensure the high level requirements
- At the top-level, we must resort to review.
  - The presence of a formal specification enables formal challenges. A review can identify additional properties that a design must possess. Formal verification techniques can then be used to either prove the design already possesses the property or reveal its absence.
- At the bottom levels in a design hierarchy, we must show that the assumptions are consistent.

Supporting text

Formal techniques can ensure that intermediate steps in the design process are carried out correctly, but we are still faced with the problem of correctly stating the top-level specification. Formal specification techniques enable us to analyze the top level specification in a more rigorous fashion, thus providing confidence that the specification is what we intend. However, it is still necessary to review the top-level to ensure that it matches expectations.

Levels of Rigor

Level 1: Specification (and proof) using conventional mathematical notation.

Level 2: Specification using a formal specification language with manual proofs.

Level 3: Specification in a formal language with automatically checked or generated proofs.

Supporting text

- Level 1 formal techniques allow freedom of expressiveness, but very little potential for mechanized support. The logical system need not be precisely defined and proofs need not explicitly address all the details.
- In Level 2, the specification language has a well defined syntactic structure and proofs proceed according to well defined rules. There is potential for some mechanized support to ensure that specifications adhere to syntactic rules and that some proof rules are applied correctly.
- In Level 3. the language and proof system have tightly integrated mechanized support. Sometimes, expressiveness of logic is restricted to allow more efficient/automatic mechanized support. Makes it feasible to formally analyze systems with greater precision than is possible with paper-and-pencil approaches.

Mechanized Support for Formal Methods

General Purpose Theorem Provers: Specification and Proof using the logic supported by theorem prover. Proofs are semi-automatic. Many of the steps are automated, but some require insight.

Examples include: PVS, HOL, Nuprl, Nqthm/Acl2, IMPS, ...

Specialized Approaches:

Model Checking: Fully automatic proofs that state machine description of hardware possess specified properties. Specifications given in a decidable temporal logic. An example system is SMV. Tools employing related techniques include COSPAN and Murϕ.

Design Derivation: Design proceeds from a behavioral level description to a hardware design via a series of *correctness preserving refinements*. Example systems include DDD and DRS.

Software: Machine checked verification of Ada code (Penelope). Decision tables for specification of software requirements (TableWise).

Supporting text

There are a number of chioces for level 3 application of formal methods. Specialized tools are useful for addressing some areas of the design space, but general purpose systems allow for addressing a number of different verification techniques within a single framework. Current research efforts include developing approaches to combine general purpose theorem provers with the more specialized techniques.

Obstacles to getting started

- Diversity of tools
  - Choice of proper tool is difficult
  - Over 50 notations, methods, and tool listed on WWW Formal Methods Virtual Library
    http://www.comlab.ox.ac.uk/archive/formal-methods.html
  - Many domain specific tools and techniques
  - Little support for combining results from different tools
  - Some efforts currently in progress
- Immaturity of Tools
  - Most tools are research prototypes
- Lack of sufficient Libraries
- Education

Supporting text

The diversity of tools and techniques is both a blessing and a curse. There are a number of domain specific approaches that are appealing, but a significant amount of expertise is required to make an intelligent choice.