Abstract State Machines for modeling and verifying hardware circuits†

Vinod Viswanath

April 14, 2003

†Presentation at CERC group meeting.
Outline

• Introduction: What are ASMs?

• Example: The dining philosophers problem

• ASMs for Hardware: Scope

• DLX Processor: Proof of correctness

• Why ASMs?
Introduction

- ASMs formalise algorithms or systems in a state-based method
- Notation formalises all possible states and state transitions
- Can express a system structure as well as its dynamics within the same notational framework
- Notion of state transitions is attached to an algebraic method
- Dynamics of a system is described by means of a set of transition rules
Introduction

• “An improved Church-Turing thesis”

• Started by Yuri Gurevich as dynamic structures in 1984

• “A general kind of abstract computational device, called dynamic structures”

• Later called evolving structures or evolving algebras

• Combination of algebras (first order structures) and transition systems
  – algebras represent data, control structures and configurations (states)
  – transition systems represent system behavior
ASM characteristics

• Precise

• Faithfulness of the model (no notational coding)

• Clarity (unlike denotational semantics)

• Executable specification (unlike Z or process algebras)

• Scalable (many-layered abstraction model)

• Generality (across domains)
Example: Dining Philosophers Problem
Example: Dining Philosophers Problem

- Modeled by *multi-agent ASMs*
- An *agent* describes a process (a philosopher)
- Behavior described by a set of transition rules
- Transition rules are parametrized
- State space consists of domains and functions
  - static functions have fixed interpretation (constants)
  - dynamic functions can change interpretations during a run
  - external functions are like dynamic functions, but their interpretation cannot be changed by the system
  - dependent functions are defined in terms of other functions in the system (macros)
Dining Philosophers ASM: State Space

static function maxPhil = 5

PHILOSOPHER = \{P_0, P_1, \ldots, P_{maxPhil-1}\}

FORK = \{fork_0, fork_1, \ldots, fork_{maxPhil-1}\}

STATUS = \{taken, released\}

static function left_fork : PHILOSOPHER \rightarrow FORK
= \{P_i \mapsto fork_i \mid 1 \leq i \leq maxPhil\}

static function right_fork : PHILOSOPHER \rightarrow FORK
= \{P_i \mapsto fork_j \mid 1 \leq i \leq maxPhil,\}
   \quad j = (i + 1) \mod maxPhil\}
Dining Philosophers ASM: State Space

dynamic function **status** : FORK $\rightarrow$ STATUS

initially \{\text{status}(f) \mapsto released \mid f \in \text{FORK}\}

dynamic function **eating** : PHILOSOPHER $\rightarrow$ BOOLEAN

initially \{\text{eating}(p) \mapsto false \mid p \in \text{PHILOSOPHER}\}

external function **hungry** : PHILOSOPHER $\rightarrow$ BOOLEAN

dependent function **forks_free**(Self) : BOOLEAN

$= \text{status(left\_fork}(\text{Self})) = \text{released}$
$\land \text{status(right\_fork}(\text{Self})) = \text{released}$
Dining Philosophers ASM: Transition Rules

if hungry(Self)
    then if forks_free(Self)
        then status(left_fork(Self)) := taken
           status(right_fork(Self)) := taken
           eating(Self) := true

if ¬hungry(Self)
    then if eating(Self)
        then eating(Self) := false
           status(left_fork(Self)) := released
           status(right_fork(Self)) := released
Different levels of abstraction

Problem

Validation

ASM0

Verification

ASM1

Verification

ASM2

Verification

Implementation

several refinement steps
Dining Philosophers ASM: Revisited

- Function **status** of forks has to be *global*
- All other functions (parametrized by Self) are *local*
- Contradictory to the notion of distributed processes
- Model distributed processes without shared functions by *refining* the ASM.
ASM WorkBench

- Formal ASM language semantics
- ASM2SMV
- Specware (graphical system for composing algebraic specifications)
- More recently – ASM2PVS

- Plentiful examples applied to different kinds of algorithms and systems
Example: Correctness of pipelined $DLX$

- Define a hierarchy of refinement steps
- Successive refinement steps mimic incremental features in hardware design
- Proofs are smaller and more elementary
- Refinement hierarchy determined by important instruction pipelining principles
- Successive refinements eliminate structural, data and control hazards in the $DLX$ design
$DLX^p$

- Parallel execution model
- Free of structural hazards (resource conflicts)
- Abstracted from data hazards, control hazards and stalls
- Compiler has an assumption to organize the sequence of instructions to be sufficiently independent of each other
$DLX^p$ correctly refines $DLX$

Let $C$ be a computation in $DLX$ and $C^p$ a computation in $DLX^p$.

Computation is decomposed into subcomputations during which a given instruction is completely executed.

**Correctness Lemma:**
For every pair of corresponding computations $(C, C^p)$, the corresponding instruction cycles compute the same result.

Correspondence between instruction cycles in $C$ and $C^p$ is defined by the order in which they occur.
$DLX^{data}$

- $DLX^p$ with data hazards for non jump/branch instructions

- Prove computational equivalence with respect to $DLX^p$

- Proof proceeds by case analysis

- Compiler assumption is free to the extent that there are no jump/branch instructions in the pipe which can have a data hazard
$DLX^{ctrl}$

- $DLX^{data}$ eliminating the stalls following jump/branch instructions

- $DLX^{ctrl}$ correctness theorem is the same as that for $DLX^{data}$ except that the computations now involve jump instructions

- Proof is by reducing this theorem to the $DLX^{data}$ theorem
$DLX^{pipe}$

- $DLX^{ctrl}$ with data hazards for jump/branch instructions

- Proof is by extensive case analysis and induction over a given $DLX$ computation

- Finishes the proof that the pipelined $DLX^{pipe}$ is computationally equivalent to the sequential $DLX$
Automation?

- Method generates smaller inductive lemmas as a by-product (incremental refinement)

- Can be amenable to automation

- ASM-WorkBench provides a framework to write executable ASM specifications

- Proof by model checking – ASM2SMV

- Proof by theorem proving – ASM2PVS
Problems

1. Translating Hardware Descriptions to ASMs

- ASMs can model the hardware structure as states and the dynamic execution flow as transitions

- Ideally, model a Verilog description as an ASM $A_v$, and its corresponding C-model as another ASM $A_c$ and refine $A_c$ to $A_v$

- Can use refinement steps for optimization

- Can visualize an ASM as a kripke structure and verify properties on it
2. Automating the ASM refinement process

   - Given an ASM $A_1$ and a set of definitions and lemmas describing a small incremental change in hardware structure, can we automatically generate the ASM $A_2$ which will is $A_1$ refined to reflect this change?

   - This has been attempted with PVS, with minimal success
Problems

3. Defining a logic for ASMs which model a class of circuits

   – Can we restrict operations on ASMs based on a class of circuits being modeled, such that it helps in automation?

   – There have been previous attempts at defining a logic for ASMs (dynamic logic, subset of modal logic)

Note: Hierarchical ASMs are ideal to represent modularity of hardware designs