Formal Techniques for System-Level Verification

Moshe Y. Vardi

Rice University

This Talk

Partly a tutorial, partly a manifesto!

FV in HW Design: A Success Story

From an impossible dream to industrial reality in 30 years!

- Combinational equivalence checking
- Sequential equivalence checking
- Assertion checking
- Symbolic simulation
- Formal engines
- …

Still verification is often the bottleneck!

What is the problem? Design Methodology!

- ***HUGE*** gap between specification and implementation!
- Increased design complexity (pipelining, speculative execution, superscalar issue, vectorization, hyper-threading, out-of-order execution, …)
- Specifications – typically, English document, complete lack of formality
- Implementation – RTL (Verilog, VHDL)
- ***Missing in action***: high-level functional reference model

Sample Specification

“The commands flow one direction from AHB to I2C, but the data flows both ways. For the write direction data are written into a FIFO. When the FIFO is full, the AHB signal HREADYout is de-asserted until there is room in the FIFO again. Upon receiving the write data, if there is room in the FIFO, the AHB bus is free for other devices sharing the AHB to proceed to their transactions. A read-cycle, however, will hold up the bus until the data is ready (because a SPLIT transaction is not supported by the bridge). Therefore, the read transaction has priority over the write …”

High-Level Models

- It's like quitting smoking; it's been done a hundred times ☺
- Typically, semi-formal (PowerPC 604)
- Or focused on performance (Power4)
- What is the difficulty?
  - Model development is hard
  - Seems like additional work
  - Not clear how to relate HL model to RTL

Is this really a problem? Yes!

- Imagine coding a nontrivial algorithm
  - In C++
  - In C
  - In assembly language
  - In RTL
- RTLers are developing the micro-architecture and its implementation at the same time
- Verifiers are verifying the micro-architecture and its implementation at the same time
- The lack of a "golden" functional reference model is an obstacle to RTL development and validation; it cost $$$!

Dealing with complexity

- G. Singer, 2005: “RTL was established two decades ago. Since then, complexity has increased sevenfold”.
- Shift in focus from performance to functionality (virtualization, security, etc.).
- We handle complexity via ***abstraction.***
- RTL is too low level.
- ***Missing step***: high-level functional model (HLFM).

Isn’t this just more work?

- Formalization *always* enhances understanding and reveals bugs.
- HLFM serves as reference model for RTLers.
- Verification can start early and aim at system level - right level of abstraction
  - Both formal verification (FV) and dynamic verification (DV)
- HL verification IP can be re-used for RTL verification, e.g.,
  - coverage of micro-architectural features.
  - Co-simulation of HLFM and RTL

System-Level Validation

- Designers think in terms of CPU events.
  - E.g., which instruction is in which pipeline stage.
  - This is a system-level event.
  - It is hard to find it in the RTL code.
- Validators need to think in terms of system-level events.
  - E.g., if a certain buffer is full, then …
  - E.g., was an interrupted instruction aligned on a 32-bit boundary when interrupt occurred?
  - Coverage should be defined in terms of system-level events.

Two validations?

- So you are asking us to do two validations instead of one?
- Yes. ***Separation of concerns*** in an important principle in software engineering – total work should decrease.
  - Need to validate specification (HLFM) – algorithmics.
  - Need to validate implementation (RTL) – hardware.
- Algorithmic bugs should be corrected in the HLFM, hardware bugs should be corrected in the RTL.

System on a Chip

The CPU is just one component in today's systems! Systems consist of many blocks:

- Multi-core
- Power manager
- Memory blocks and memory manager
- DSP core
- GPU
- Analog
- ....

System-Level Model (SLM)

Design must start with an system-level model!

- Architectural validation
- Performance modeling
- Architectural trade-offs
- HW/SW co-design
- Functional reference model

The Language Question

- Desiderata: formality, high level, abstraction, executability
- Academic languages: ACL2, UCLID, Hawk
  - No industrial acceptance
- Industrial languages: Esterel, BlueSpec
  - Low popularity
- Industrial language: SystemVerilog
  - Not enough high-level, abstraction
- Industrial language: SystemC
  - Low formality
  - *High popularity* (www.systemc.org)

SystemC

- System-level modeling language
  - C++ based (C++ plus libraries)
  - OO used for abstraction, modularity, compositionality, and reuse
- Rich set of data types (C++ plus HW)
- Rich set of libraries for modeling at different levels
  - Signals
  - FIFOs
  - Transaction-level modeling (TLM)
- Simulation kernel - executability

Why Object Orientation?

- Capture core functionality in a base class.
- Capture specialized functionality via inheritance.
- Capture design refinement via inheritance.

- ***Advantage***: Small functionality changes need not entail major change of specifications.

- ***Point:*** Object orientation is useful for modeling HW, just as it is for SW.

Basic Elements of SystemC

- Modules: Basic building blocks, form hierarchy.
- Hardware data types, e.g., sc_logic (4 valued).
- Concurrent processes - methods and threads:
  - SC_METHOD: functions
  - SC_THREAD: processes that can suspend and resume
- Events and sensitivity: drive simulation.
- Channels and ports: modules communicate via channels; they connect to channels via ports.

Modules: Basic Building Blocks

- SC_MODULE:
  - Ports
  - Internal Variables
  - Constructor
  - Internal Methods.
- Ports:
  - direction : One of sc_in,sc_out,sc_inout
  - type : Data type
  - variable : Valid variable name
- SC_CTOR: Creates and initializes an instance of a module.

D-FF in SystemC

```
#include "systemc.h"
SC_MODULE(d_ff)
{ sc_in<bool> din;
sc_in<bool> clock;
sc_out<bool> dout;
void doit() { dout = din; };
SC_CTOR(d_ff)
{ SC_METHOD(doit);
sensitive_pos << clock; } };
```

SC_METHOD

- When called it gets started, executes, and returns execution back to calling mechanism.
- Gets called whenever the signal/event in the sensitivity list changes/occurs.

```
void incr_count () {
if (reset.read() == 1) { count = 0}
else if (enable.read() == 1) { count = count + 1} }
SC_METHOD(incr_count);
 sensitive << reset; sensitive << clock.pos();
```

SC_THREAD

- Process that suspends and resumes

```
void incr_count () {
while (true) {wait();
if (reset.read() == 1) { count = 0} else if
   (enable.read() == 1) { count = count + 1}} }


SC_THREAD(incr_count);
sensitive << clock.pos();
```

SC_EVENT

- Starts or resumes processes
  - Immediate notification – e.notify()
  - Queued notification – e.notify(SC-ZERO-TIME)
  - Timed notification – e.notify(1,SC_NS)
  - Cancellation – e.cancel()
  - Waiting for – wait(e)
  - Triggered by: next_trigger(event)

- SystemC simulation is event driven.

Built-In Channels

- sc_mutex
  - int lock() : Lock the mutex if it is free, else wait till mutex gets free.
  - int unlock() : Unlock the mutex
  - int trylock() : Check if mutex is free, if free then lock it else return -1.
  - char* kind() : Return string "sc_mutex"
- sc_fifo
- sc_semaphore
- …

Transaction-Level Modeling

- Motivation: system-level modeling
  - rise above the RTL
  - increased abstraction
  - early system exploration
  - significantly faster simulation
- ***Key***: separate communication and computation
  - Model communication via interfaces, e.g., tlm_transport_if
- Abstract interfaces can be later refined and concretized, all the way to RTL.
- Term “transaction” does not have precise meaning!

Memory Example - I

```
#include "systemc.h"
#define DATA_WIDTH 8
#define ADDR_WIDTH 8
#define RAM_DEPTH 1 << ADDR_WIDTH
SC_MODULE (ram_sp_ar_aw) {
sc_in <sc_uint<ADDR_WIDTH> > address ;
sc_in <bool> cs ;
sc_in <bool> we ;
sc_in <bool> oe ;
sc_in <sc_uint<DATA_WIDTH> > data_in ;
sc_out <sc_uint<DATA_WIDTH> > data_out;
```

Memory Example - II

```
sc_uint <DATA_WIDTH> mem [RAM_DEPTH];
void write_mem () { if (cs.read() && we.read()) {
mem[address.read()] = data_in.read(); } }
void read_mem ()
   { if (cs.read() && !we.read() && oe.read()) {
    data_out.write(mem[address.read()]); } }
 SC_CTOR(ram_sp_ar_aw) {
SC_METHOD (read_mem);
sensitive << address << cs << we << oe;
SC_METHOD (write_mem);
sensitive << address << cs << we << data_in; } };
```

SystemC Semantics

- C++
- Event-driven simulation semantics
  - Interleaving semantics for concurrency
  - Informal standard, but precise description of event order and execution
  - Was formalized in terms of Distributed Abstract State Machines (Mueller et al., 01)
- Fully formal semantic is lacking

SystemC Verification Standard

- Transaction-based verification
  - Use "transactors" to connect test with design; bridge different levels of abstractions (e.g., TLM and RTL)
- Data introspection
  - Manipulation of high-level data types
- Transaction recording
  - Capturing transaction-level activities
- Constrained and weighted randomization
  - Constraint classes
- Sole focus: dynamic verification (DV) – assumes hand-written checkers

Formal Verification

- Maturing technology
  - Complete coverage of design state space
  - Highly effective at catching corner cases
  - Challenged by design size and complexity
- Case study: Intel's P4 verification (B. Bentley)
  - 60 person years (15% of verification effort)
  - 14,000 formal assertions proved
  - 5,000 bugs caught
  - 100 "high-quality" bugs
  - 20 "show stoppers"
- Today's challenge: FV for SystemC

Assertion-Based Verification

- Model checking:
  - Formal model ***M*** of system under verification
  - Formal assertion ***f*** describing a functional requirement (e.g., "every message is acknowledged within 10 cycles")
  - Algorithmically checking that ***f*** holds in ***M***
  - Counterexample trace when ***f*** fails in ***M***
- 25 years of model checking
  - Increasing acceptance by HW industry
  - Significant recent progress in applications to SW – significant push by MS
  - Main challenge: state-explosion problem

Assertion Languages

- Pnueli, 1977: focus on ongoing behavior, rather than input/output behavior – ***temporal logic***
- Standardization efforts of the early 2000s by Accellera
  - ***PSL***: temporal logic extended with regular events (based on ForSpec and Sugar)
  - ***SVA***: less temporal and more regular
- Focus: RTL
- Needed: Extension to SystemC

Temporal Resolution

Fundamental question: What is the trace of a SystemC execution

- ***Trace is the starting point of temporal logic!***

- Kroening&Sharygina'05: Hide kernel completely, expose user code
- Moy'05: Expose an abstract model of kernel
  - No canonical notion of *cycle* in SystemC
- Tabakov et al.'08: Expose semantics of kernel
  - No canonical notion of *cycle* in SystemC
  - Kernel has 11 states, no need to abstract

A Temporal Logic for SystemC

Q: How should a temporal logic be adapted to System?

A: Tabakov et al, 2009 – add lots of Booleans

- Booleans for C++ expressions
- Booleans for software (@label, procedure calls, etc.)
- Booleans for event notifications
- Booleans for kernel phases
- Clock mechanism of PSL/SVA can use Booleans
  - Adapt temporal resolution to abstraction level

Assertion-Based DV

- Traditional approach to DV:
  hand-crafted checkers, significant effort

- Abarbanel et al., 00: compile formal assertions into checkers
  - Allows for specification re-use
  - Consistency between DV and FV
  - Used in IBM (FoCs), Intel (Fedex)
  - Armoni et al., 06: applicable for full PSL and SVA, generates finite-state checkers
  - RTL checkers: fast simulation and emulation

ABDV for SystemC

- Initial efforts reported:
  - Grosse&Drechsler, 2004: very limited temporal assertions
  - Habibi et al., 2004: full PSL
    - Use Abstract State Machines as formal model
    - Details severely lacking
- Under work (Tabakov):
  - Modify kernel minimally to expose semantics
  - Compile assertions into SystemC checkers
- Overall: seems quite doable
  - Related: assertion-based test generation and coverage

Explicit-State Model Checking

- Prototype: SPIN (ACM SW System Award)
  - A specialized modeling language – Promela
  - Negated temporal assertion compiled into a nondeterministic checker ("Buechi automaton")
  - Search engine conducts DFS to find a counterexample trace – a trace of the design that is accepted by the checker
  - State caching using for liveness-error analysis
  - Can handle systems with millions of states
- Major weakness:
  - specialized modeling language (Contrast: HW model checkers use RTL)
  - State-explosion problem

Native Java Model Checkers

- Bandera:
  - Use slicing and abstraction to extract finite-state model from Java
  - Call model checkers such as SPIN or SMV
- Java Pathfinder:
  - Modified JVM to check all possible executions
  - Heavy use of abstraction to cope with state explosion

Explicit-State SystemC MC

Two possible approaches:

- Extract finite models from SystemC models and reduce to other model checkers
- Modify simulation kernel:
  - Resolve non-determinism exhaustively; all paths needs to be explored.
  - Add state caching to catch cycles – liveness errors analysis
- Which is more doable? More general? At any rate, quite non-trivial!

Symbolic Execution

Between DV and FV:

- Symbolic simulation – abstract interpretation
  - Explore concrete control paths of system
  - Use symbolic, rather than concrete, data
  - Extract logical conditions for path feasibility
  - Reason using decision procedures
- Recent successes with symbolic execution
  - Symbolic trajectory evaluation (STE)
  - Microcode verification (Intel)
  - Static analysis for dynamic errors, e.g., buffer overflow (PREfix and PREfast)
  - Verification of Java programs

Symbolic Model Checking

Beyond 10**6 states: 10**20 states and more

Symbolic Model Checking

- Describe state space and state transitions by means of logical constraints
- Symbolic algorithm for computing reachable state sets – BDD or SAT based.
- Scales to large state spaces: > 10**20 states.
- Bounded MC complements full MC by search for bounded-length error traces

- Key requirement: formal semantics!
  - Need to express transitions logically

Example: 3-bit counter

- Variables: v0,v1,v2
- Transition relation: R(v0,v1,v2,v0',v1',v2')
  - V0'=!v0
  - V1'= v0 xor v1
  - V2'= (v0 & v1) xor v2
- For 64 bits:
  - 64 logic equations
  - 10**20 states!

SMC for SystemC

- Initial progress: Grosse&Drechsler'03
  - Limited to RTL SystemC – semantics more easily formalizable
  - Moy'05: specialized library, scalability challenge
- Major challenge: formalizing full SystemC
  - No formal semantics for C++
  - Note: No symbolic model checker for Java
- Room for hope – recent reasoning tools for OO languages (Java and C#):
  - Bogor – assertion checking for JML
  - Spec# compiler

SMC for SystemC: KS'05

Kroening&Sharygina'05: The compiler is the semantics!

- Formal semantics: labeled Kripke structures
  - Both states and transitions are labeled
- Kernel is abstracted away (see earlier discussion)
- Front end of gcc is used to extract LKS from SystemC model
- HW/SW partition for increased abstraction
- SAT-based model checking

Equivalence checking

- Equivalence checking – most successful FV technique
  - Does not require formal assertions
  - Checks that one system is functionally equivalent to another system, e.g., circuit before and after timing optimizations
  - Widely used in semiconductor industry
- Combinational equivalence – solved problem
- Sequential equivalence – major progress
  - Analyze product machine (Coudert&Madre, Pixley, 1990-2)

  Key insight: two systems are closely related!

Eq. Checking: SystemC vs RTL

Challenging Goal (Calypto,Synopsys):

- Verify SLM
- Implement/Synthesize RTL
- Prove equivalence

Special challenge: SLM and RTL at different levels of abstraction

Eq. Checking: SystemC vs RTL

Relating SLM and RTL:

- Specify notion of equivalence (Calypto&Synopsis)
  - Reduce non-cycle-accurate problem to cycle-accurate problem
- Define memory mapping (Synopsys)
- Specify interface mappings and constraints
  - TLM vs signals

Challenging FV problem: bit level, word level rewrite engines (Synopsys)

Eq. Checking: SystemC vs RTL

Less than equivalence checking:

- Feasible now: “conformance” of SLM and RTL
  - Test suite developed for SLM and then applied to RTL
  - Co-simulation of SLM and RTL (interface with RTL simulator)
  - Refinement mappings from SLM to RTL
  - …..

In Summary

Formal techniques for SystemC:

- Assertion-based DV
- Explicit-state model checking
- Symbolic execution
- Symbolic model checking
- Equivalence checking

Call to Arms

A brief history of FV

- 1981 - : HW verification
- 1996 - : Protocol verification
- 2000 - : SW verification

- Today's challenge: System-level verification
  - HW+SW+protocols

Back to the language question

- What makes SystemC so popular?
  - Open source?
  - C++ is gradually fading out in SW.
- Is SystemC here to stay?
  - Or is it a fad?
- Esterel and BlueSpec have many technical advantages over SystemC
- At least, why not SystemC#?
  - Sigh. SystemC is it, at least for now.