Effective Validation of Firmware

Enabling firmware development and validation to keep pace with hardware innovations.

Tom Melham

University of Oxford

Problem

Firmware today

- facing greater complexity and shorter schedules
- coded at low level, including inline assembly
- gated by HW development

Current testing-based approaches inadequate

- need *both* HW and SW models
- debugging difficult

The problem is growing – need some new ideas.

Attack

Objectives

- enable much earlier development and validation
- better, faster debugging - through automated analysis
- higher productivity - by raising abstraction level

Critical-mass effort by a top-class team, balancing

- near-term, immediately applicable tools and results
- transformative, ambitious, revolutionary research

Five-year effort funded by Intel ARO + public funds.

Key Ideas

Specifically target *low-level* firmware

Joint HW/SW modelling in *SystemC*

- for maximum near-term impact
- transaction-level approach

Modern *automated analysis*

- proven ideas from OS software level of firmware
- in parallel pursue: static analysis, dynamic testing, hybrid

Raise *abstraction* level

- type-based resource analysis and address safety

Daniel Kroening
Oxford

Tom Melham
Oxford

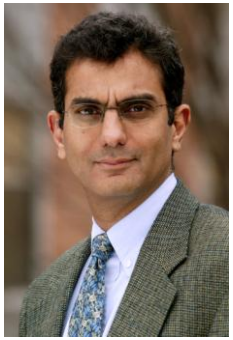
Moshe Vardi
Rice

A world-class team

- with full spectrum of HW, SW, and validation expertise
- at four top universities
- working closely together

and a proven track record of delivering innovation to industry.

Luke Ong
Oxford



Sharad Malik
Princeton

Alan Hu
UBC

SystemC
C Bounded Model Checking
Decision Procedures

Hardware FV
High level modelling
Symbolic simulation

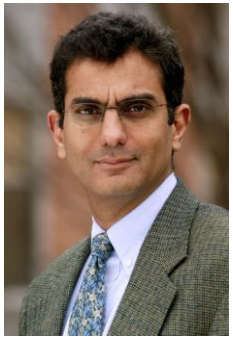
SystemC
Assertion-based FV
High level modelling

Intel Mentor

Jim Grundy

Firmware validation
Domain knowledge

Program verification
Types
Semantics



SAT Solvers & Extensions
Transaction-Level Models
Embedded SW Timing Analysis

Symbolic execution
Low-level SW analysis
Concurrent SW

◆ Environment Modelling

SystemC *bridging model* of HW/SW interface

- **early abstract model of HW, to validate SW**
- **model of SW to check design of HW**
- **breaks sequential dependency**

A *transaction-level* model

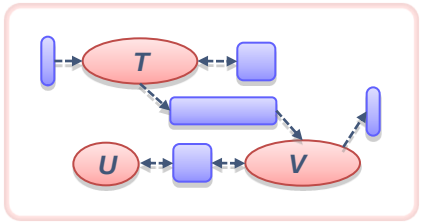
- **capture higher-level *meaning* with coherent ‘units of work’**
- **enable specifications in terms of this meaning**

How obtained?

- **legacy designs, data-mining techniques, …**

TLM: Princeton Model and Language

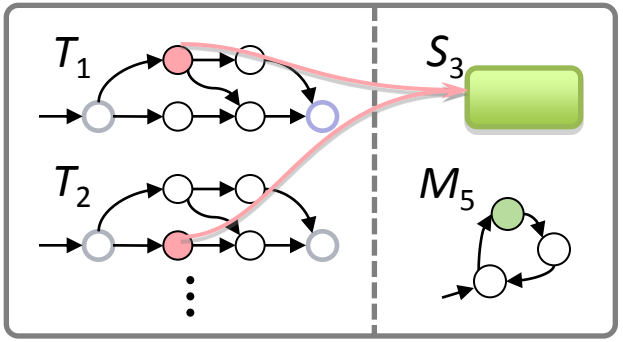
PriM Architecture



Architecture Model

Specification with concurrent "units of work"

PriM Microarchitecture



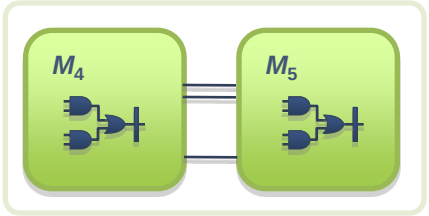
Microarchitecture Model

Implementation of "units of work"

•in space (physical resources) and

•time (clock cycles)

⬇ *Synthesized RTL*



Transactions enable:

- **refinement checks**
  does the microarchitecture implement the architecture?
- **test generation**
  analysis of high-level cases
  analysis of potential resource conflicts
- **equivalence checking**
  controlled synthesis enables simpler equivalence checking between microarchitecture and RTL

◆ Automated Firmware Analysis

Static checkers – analyze code properties without running it

- conformance to HW/SW interface
- safety properties – e.g. memory safety
- quantitative properties – timing, power

Technology

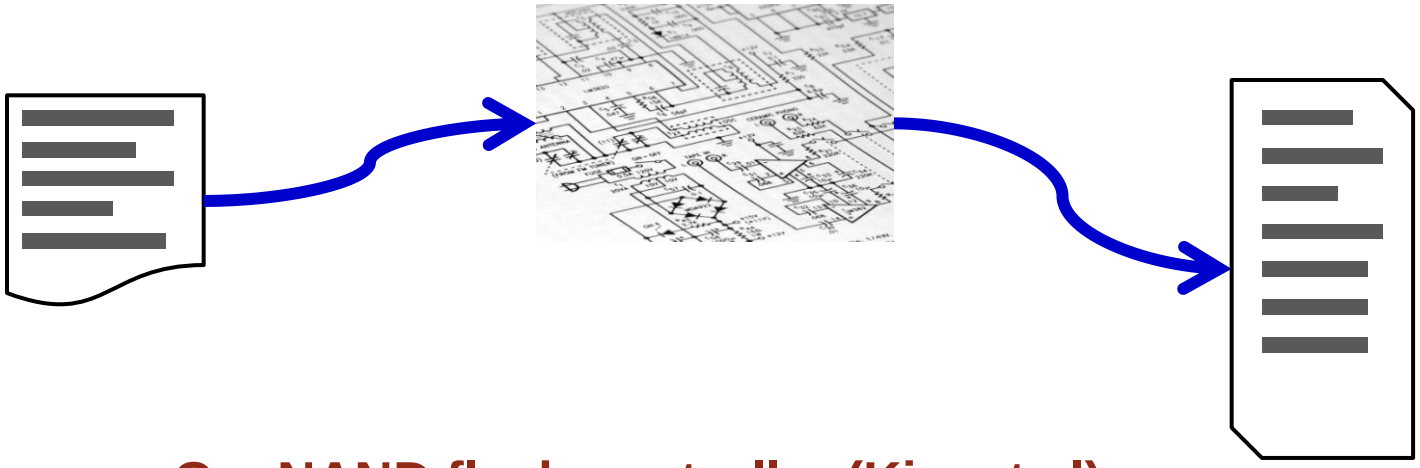
- *symbolic* code execution, backed up with SMT
- bit-precise semantics for tricky low-level features
- inline assembly, interrupts, typecasting

Dynamic testing & hybrid methods

- leverage TLM for test generation, coverage
- derive monitors the HW/SW interface model

Symbolic Simulation

CBMC – bounded model checking for C code



Samsung OneNAND flash controller (Kim et al)

- sector translation layer, multi-sector read
- deeply nested loops iterating though complex data structure
- exhaustive validation that data correctly read

Progress is Rapid

CBMC (Clarke, Kroening, Yorav - 2003)

- pioneered using bit-accurate symbolic execution
- completely automatically, for full ANSI C.
- scales to a few thousand lines of code.

Calysto (Babic, Hu - 2008)

- also based on fully automatic, bit-accurate symbolic execution
- but with improvements on all levels:
  - preliminary, lightweight static analysis
  - symbolic execution algorithm
  - abstraction/refinement algorithm
  - decision procedure

◆ Languages and Types

Raise coding abstraction level of low-level firmware

- type checking to establish specific properties
- more scalable than e.g. model checking

Main target: resource usage analysis, investigating

- assembly language with explicit heap operations – size types
- stack overflow in interrupt driven systems – types + MC
- synchronous cooperative concurrency – resource bounds

Address safety and access control

- type-enforced freedom from memory races
- infer data-flow properties, e.g. memory ordering

Formal Analysis of Interrupt-Driven Programs

Simple example problem

- **interrupt handling is governed by a stack discipline.**
- **interrupts can be interrupted - programmer error can allow the stack to grow unchecked.**

Our approach

- **typing discipline for a family of generic assembly languages with interrupts (interrupt calculus of Palsberg et al. 2002).**
- **type soundness: well-typed code does not overflow the stack.**
- **model checking + type inference: use pushdown automata model checking to help derive types.**

Other properties

- **liveness properties; termination and recurrence.**
- **performance analysis – e.g. avoidance of interrupt storm**

We Would Value Your Input

Insight – characterizing the *real* issues

Industrial challenge problems

A steer towards relevant public-domain examples

Joint research