Formal Methods Need Not Be Black Magic

Joseph R. Kiniry and Daniel M. Zimmerman
1st RISC-V Summit, Santa Clara, California
4 December 2018
Formal Methods Through the Ages (1 of 2)

- computer science is the daughter of early 20th century mathematics/logic
- hardware/computer/electrical engineering is the daughter of CS and physics
- physics is the daughter of 18th and 19th century mathematics
- formal methods is the application of mathematics to the problems of computer science and systems engineering (hard-/firm-/software)
Formal Methods Through the Ages (2 of 2)

- formal methods in the 1950s–1990s was viewed as an esoteric, impossible to use discipline only fit for PhDs solving problems about toy systems and languages
- formal methods exploded into hardware verification in the late 80s and 90s due to enormously expensive failures in CPU design and testing
- even today, around twenty years later, formal methods is viewed in most hardware design houses as an esoteric, difficult to use discipline only fit for a few very expert ‘formal’ engineers solving problems for small IP blocks and antique languages like Verilog and VHDL
Advancements of ‘Formal’ in Systems Engineering

- 1950s–1990s: “esoteric”, “impossible to use”, “only fit for PhDs solving problems about toy systems and languages”
- Late 1990s: started to tackle mainstream programming languages and real systems, but it still took PhDs months of work for results
- 2000s: started to leverage modern GHz, GBs, and Gb/s, thus automation and scalability exploded and FM crept into mainstream tools
  - compiler semantic static checking, JVM class loading
- 2010s: started to focus on UX and mainstream integration
  - language, compiler, and IDE integration and UI advancements
- Now: pervasive (for software, at least), but completely hidden to engineers—almost as if someone has snuck FM into our workflows!
Secret Ninja Formal Methods (SNFM)

- ninjas are the sneakiest kids in town
- *Secret Ninja Formal Methods* is a peer-reviewed applied formal methods systems development methodology that we defined over a decade ago
- SNFM focuses on verification-centric system design and development
- we are now applying SNFM to hardware engineering and verification
- key idea: powerful tools hide the mathematics and expose all specification and reasoning using UIs and metaphors familiar to developers
Facets of (Secret Ninja Applied) Formal Methods

- **specification**: describe functional behavior or non-functional properties
- **modeling**: extract formal models from specifications, implementations, verification artifacts such as test benches, etc.
- **implementation**: build (parts of) the system according to a configuration
- **execution**: run (parts of) the system in simulation, on VMs, or hardware
- **validation**: execute (parts of) the system and evaluate properties using runtime assertion checking
- **verification**: check properties of a model for all inputs/environments
SNFM Tools and Technologies for Firmware/Software

- over the past 20 years we have developed instantiations of the SNFM methodology for various platforms and languages:
  - **Java**: BON for system specs; JML for model-based specs; Java for implementation; JMLUnitNG for validation; Eclipse/IDEA/Emacs for IDE; Beetlz+ESC/Java2+OpenJML+KeY for verification
  - **.NET**: BON for system specs; Code Contracts for model-based specs; F#/C#/Spec# for implementation; VisualStudio/VisualStudioCode/Emacs for IDE; PEX/Moles for validation; Code Contracts for verification
  - **Eiffel**: BON for system specs; Eiffel contracts for model-based specs; Eiffel for implementation; EiffelStudio/Eclipse/Emacs for IDE; EiffelStudio and Eve for validation and verification
SNFM Tools and Technologies for Firmware/Software

- over the past 20 years we have developed instantiations of the SNFM methodology for various platforms and languages:
  - **SPARK**: BON for system specs; SPARK contracts for model-based specs; SPARK/Ada/C for implementation; GPS for IDE; SPARK tools for validation and verification
  - **C/Rust**: BON for system specs; ACSL for model-based specs; C, Rust, and assembly for implementation; Eclipse/mbeddr/CLion/VisualStudio/Emacs for IDE; Frama-C and SAW for validation and verification
  - **Functional languages**: BON for system specs, Haskell types and modules for type-based specs; Haskell/ML/Idris/Coq for specification and implementation; Emacs for IDE; QuickCheck, behavioral types, and theorem proving for V&V
SNFM Tools and Technologies for Hardware

- What tools in the hardware world use formal methods but hide all logic?
  - Cadence’s JasperGold and Mentor Graphics’s Questa for (System)Verilog
  - Clifford Wolf/SymbioticEDA’s Yosys and SymbiYosys for (System)Verilog
  - Galois’s SAW for Bluespec SystemVerilog (BSV)

- The *BEESPIN Tool Suite* (in development at Galois) features:
  - support for five HDLs: Verilog, SystemVerilog, SystemC, BSV, and Chisel
  - extraction of architecture specification from an implementation
  - extraction of feature model from an implementation
  - configuration of a feature model into a product
  - measurement and evaluation of PPAS (S=Security) for a family of products
  - evaluation of correctness and security for a family of products
Peeking Behind the Curtain (State Space Complexity)

- state space complexity is the red herring of hardware verification!
  - if your only reasoning technique is model checking, then the size of your state space matters enormously (and reasoning effort is enormous!)
  - if you are using any one of a dozen other reasoning techniques rarely applied to hardware, it matters much less (and reasoning effort is remarkably small!)
- the kind of system you reason about and its complexity matters much more
  - sequential systems
  - (structured/reasonable) concurrent systems
  - (structured/reasonable) distributed systems
- the kind of specification and reasoning that you do matters enormously
  - compositional
  - non-compositional
models extracted from simple (100 LOC) programs and specifications vary widely in size:
  • complex programming/hardware description language: 10–100s of pages of logic
  • simple specification language (predicate logic or first-order logic): a few pages of logic
  • complex specification language (temporal logic or model-based spec): 100s of pages of logic
models derived from real world specifications and implementations are regularly 100–1000s of pages of logic per property
10+ Years of Case Studies in SNFM

- formally verified video games
- electronic voting technologies
- distributed systems frameworks
- mobile agent platforms
- hardware CAD technologies
- embedded systems products (IoT, smart sensor networks, etc.)
- tools like interpreters, type checkers, compilers, etc.
- secure boot for RISC-V CPUs
- RISC-V microcontroller-sized CPUs
SNFM and RISC-V

- formally verified secure boot for Bluespec’s Piccolo RV32I RISC-V
- ongoing work on rigorously validating and formally verifying arbitrary RISC-V CPUs (particularly the three CPU variants in SSITH, a 32b microcontroller, a 64b desktop CPU, and a 64b superscalar server CPU)
- the BESSPIN Tool Suite itself, which facilitates PPAS (power, performance, area, and secure) evaluation of SoCs
Be a Ninja? Be a Ninja!

- the core of being a ninja is accepting in your heart *thinking before doing*
  - *understand* what you want to create & *describe* it before you start coding
  - describe your system using artifacts that create *evidence*
  - descriptions for hardware are test, validation, and verification benches
  - design and build for verification

- **we want more ninjas in the world!** *dive in! get involved!*
  - *get a broader perspective* and learn more about what has happened in the world of applied formal methods and rigorous systems engineering
  - experiment with free rigorous validation and verification tools
  - contribute to open source tools for hardware design—don’t be resigned to the current state of the industry