The SSP: An Example of High-Assurance Systems Engineering

Gregory L. Wickstrom
Sandia National Laboratories

Steven E. Morrison
Sandia National Laboratories

Victor L. Winter
University of Nebraska at Omaha

Jared Davis
UT Austin

Steve Roach
University of Texas at El Paso
Recent focus: correct behavioral implementations
- Targeted Java for its safety and security features
- Environmental requirements drove custom processor design
- Balance validation/verification work with budget and schedule
The Sandia Secure Processor

- **Supported Java Features**
  - Primitive types
    - boolean
    - byte
    - char
    - short
    - int
    - long
  - Language
    - object orientation
    - inheritance
    - run-time type identification
    - arrays

- **Unsupported Java Features**
  - Primitive types
    - float
    - double
  - Language
    - exceptions
    - interfaces
    - threads
    - garbage collection
    - standard libraries

- **Implementation Details**
  - Direct hardware bytecode support
  - Object oriented hardware support
  - 32 bit internal architecture
  - Processor core is \( \approx 40,000 \) gates
  - Operates up to 25Mhz
  - Radiation hardened

- **Validation/Verification Work**
  - Hardware/Software co-simulation tools
  - VHDL validation
  - ASIC validation
  - Class loader validation
  - Class loader verification
• Design synthesis and layout verification was unusually smooth
  ➢ Synthesis was only 3 weeks (usually 6 to 18 months)
  ➢ Zero timing violations despite aggressive clock rate (usually 100’s)

• Standard IC fabrication testing was highly successful
  ➢ Automatic scan insertion
    Reached 100% of 8000 registers (usually 90%)
    Covered 98% of total logic (usually 65%)
    Largely effortless

• The bullets above provide evidence of clean digital design and correct IC fabrication
  ➢ Must also provide evidence of correct architecture w.r.t. Java
  ➢ Must show that design meets functional requirements
  ➢ Must show that ASIC meets functional requirements
  ➢ Used N-Version programming and automatic checking and cross-checking for validation evidence
ROM Generation & Validation Tools

- **Error Injection Info**
  - Induced single bit errors in every location of 18Kbyte application ROM
  - ROM contained 20 classes
  - All bit flips were detected by RIC/CLIC

- **RIC Info**
  - Ensures $\approx 60$ properties are always true
  - Found 73% of bit flip errors

- **CLIC Info**
  - Ensures $\approx 20$ application tree properties are always equivalent
  - Found 27% of bit flip errors not detected by RIC

---

**Diagram:**
- Source Code
- Commercial Java Compiler
- Class Files
- Custom Java-based SSP Class Loader
- Custom HATS-based SSP Class Loader
- ROM Image
- Class Loader Integrity Checker
- Equivalence Validation
- Internal Consistency Validation
- Bit Errors
A Transformational Perspective

- In the current system the class loader is written in Java

- However, many of the activities of a class loader can be understood in transformational terms

- Transformation-based manipulations can have characteristics that make them amenable to formal verification

- As a result, we have also developed a transformation-based class loader for the SSP and are in the process of formally verifying its correctness
An Example

Constant pool index:
Information local to a specific class file

Symbolic reference:
Global information

Absolute address or offset address

27

Fido.color int

0x0016

This transformation is based on information found in the local constant pool of a given class file.

This transformation is based on global information distributed across multiple class files. In this case, Fido is a dog class that inherits some properties from ancestor classes, each of which in a different class file.
Higher-Order Transformations

- Class files and ROM images are complex structures.

- Information distributed throughout a class file or distributed across multiple class files are difficult to manage through first-order transformation.

- However, in a higher-order transformation framework, one is oftentimes able to elegantly express the distribution of data throughout a term structure.

- The transformational-based class loader for the SSP has been developed using the higher-order transformation system called HATS.
Verification Efforts

- Produce convincing evidence that the SSP and associated class loader is correct
  \[ \forall (C_0, I) \text{Eval}_{JVM}(C_0, I) = \text{Eval}_{SSP}(T^*(C_0), I) \]
  where:
  - \( C_0 \) is a set of class files generated by a Java compiler
  - \( I \) is a set of inputs
  - \( \text{Eval}_{XXX} \) behavior of the program encoded in the class files or ROM image
  - \( T^* \) is the set of transformations on \( C_0 \) to produce a ROM image

- Plan to use ACL2
  - Automated theorem prover written by Moore, Kauffman, and Boyer
  - Uses induction over a logic written in Common Lisp
  - Can be thought of as a “proof checker”

- SSP architecture verification
  - A model of the SSP architecture has been specified in Lisp
  - Formal specifications must be developed and verified

- HATS-based class loader verification
  - Verify each transformation
Future Work

- Continue SSP system verification
  - Of architecture
  - Of class loader
  - Verify at low level (i.e. gates)

- Microcode assembler
  - Improves conceptual understanding of operation
  - Formalizes specification for automated reasoning

- Reduce SSP’s hardware footprint
  - Minimizes effort for low level verification
  - Centralize more control into microcode
  - Enables smaller devices

- Extend Java capabilities
  - clinit, interfaces, exceptions, garbage collection, threads, etc.