The SSP: An Example of High-Assurance Systems Engineering[*]

Gregory L. Wickstrom
Sandia National Laboratories

Department of Surety Electronics and Software

glwicks@sandia.gov

Steven E. Morrison
Sandia National Laboratories

Department of Surety Electronics and Software

semorri@sandia.gov

Jared Davis
UT Austin

Department of Computer Science

jared@cs.utexas.edu

Steve Roach
University of Texas at El Paso

Department of Computer Science

sroach@cs.utep.edu

Victor L. Winter
University of Nebraska at Omaha

Department of Computer Science

vwinter@mail.unomaha.edu

Abstract

The SSP is a high assurance systems engineering effort spanning both hardware and software. Extensive design review, first principle design, n-version programming, program transformation, verification, and consistency checking are the techniques used to provide assurance in the correctness of the resulting system.

1 Introduction

At Sandia National Laboratories, an effort is underway to develop a computational infrastructure for high-consequence embedded real-time systems. The resulting computational system must satisfy numerous constraints including those listed below.

1. It must be possible to subject the hardware to the full spectrum of verification and validation techniques ranging from design-level analysis down to gate level inspections.

2. Application software must be developed using a strongly typed high-level language.

3. Software developers must be highly proficient in the application language.

4. High assurance must be provided in the correctness of the translation between the high-level application programs being developed and the corresponding low-level programs being executed.

5. After translation, resulting low-level programs must have a small memory footprint.

6. To the extent possible, software concepts should be traceable across the abstraction spectrum.

Requirement (1) provides the rationale for developing a processor in-house. Numerous benefits resulted from this decision. For example, a processor design team was assembled that included both hardware developers as well as people with backgrounds in software development and formal methods. Overall, the team recognized the importance of developing a design that was analyzable. On numerous occassions during development tradeoffs were made favoring analyzability over efficiency. In a similar spirit, functionality was frequently shifted between hardware and software in order to simplify the overall hardware design.

Requirements (2) and (3) provide the rationale for targeting a modern and popular high-level language. Requirement (2) calls into question languages such as C and C++ which support automatic coercions of data, elevate pointers to first-class citizenship,

and burden the user with the details of memory management. Requirement (3) rules out languages such as Ada for which highly proficient programmers are difficult to find. This leaves Java as a prime candidate for the development language of choice. This sentiment is beginning to be echoed throughout the military complex:

> "Among the key motivations for the military's interest in Java is a drive to transition away from Ada. The feeling is that Java represents a modern and more commercially available technology than alternatives"[4].

Java is strongly typed, provides users with very controlled access to and manipulation of references, and shields the developer from memory management concerns. From the perspective of security, one of the biggest improvements of Java over C and C++ is a memory model that eliminates the possibility of over-writing memory and corrupting data[5]. Java does not have pointers and neither does compiled Java code which only references memory in a symbolic fashion. Java also encourages software quality and analyzability by removing entire classes of errors that a C or C++ programmer could make. For example, Java does not support *goto*-statements, operator overloading, or automatic coercions. Another useful feature of Java is that its semantic model has been defined in an abstract *architecture neutral* manner. From the perspective of in-house processor development an architecture neutral model is attractive because it gives freedom to the hardware design. From the perspective of software development, the robustness of portability provided by Java's semantic model provides confidence that software developed and tested on given computing platform such as a PC will have an equivalent behavior when run on another platform such as the in-house processor we are developing. For languages with portability concerns such a conclusion cannot be safely drawn.

On commercial platforms, it is common to find Java programs being interpreted by a *virtual machine* (that is itself written in software). Requirement (5) made such an approach unattractive since the memory footprint of a virtual machine would most likely consume more memory than allowed for the entire application. Furthermore, requirement (6) would be jeopardized since it would be difficult to trace software concepts (e.g., classes, fields, methods) beyond the virtual machine boundary. As a result, the decision was made to implement a suitable subset of the Java Virtual Machine (JVM) directly in hardware. The

processor developed was christened the Sandia Secure Processor (SSP). Figure 1 shows an embedded system containing the SSP.
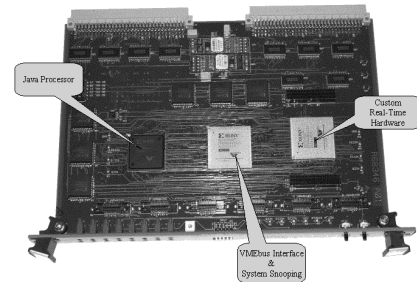

Figure 1: An Embedded System Containing the SSP

Though there are plans for future extensions, the current version of the SSP does not implement the following features of Java:

- **Garbage collection**. Although important to the execution of Java in many applications, real-time embedded systems generally avoid the need for dynamic memory allocation and de-allocation due to lack of timing determinism.

- **Multiple threads**. Again, important to Java in general, but not necessary for small embedded systems.

- **Interfaces.** A Java interface capability would have been nice for a number of reasons, but was not implemented due to schedule concerns.

- **Exception handling**. The SSP actually performs exception detection but not exception handling. Instead, on the detection of an exception the processor shuts down in an orderly fashion.

- **Floating point operations**. The application domains under consideration do not require floating point operations and therefore in order to avoid unnecessary complexity they are not implemented by the SSP.

- **Dynamic arrays**. Although arrays of multiple sizes and dimensions are supported, the size of those arrays must be statically determined.

- **Initialization of static fields**. The current version of the SSP does not support the "first use" semantics of Java relating to the initialization of static fields through the execution of <clinit> methods. However, the next release of the SSP will support initialization of statics.

- **Java Libraries**. The previously stated limitations also imply that the SSP does not support the traditional Java libraries, as many of them rely on these capabilities. However, even if the processor were to support all of the items listed, it would be unlikely that the standard Java libraries would be used in a security or safety critical system until their source code could be subjected to a complete analysis.

The embedded nature of the SSP prohibits the use of Java's dynamic binding capabilities. For example, loading a class file over a network during runtime is an operation that is physically not possible and would be strictly prohibited in any case. Thus the SSP is a *closed system* in the sense that all class files belonging to an application must be present prior to execution. This enables class loading to be performed statically (i.e., before execution). As a result, the functionality of the class loader can be factored out of the SSP hardware and implemented in software (see Figure 2). The advantages of this are: (1) the resulting hardware is greatly simplified yeilding a more analyzable system, (2) the resulting hardware is smaller taking up less space and consuming less power, (3) the ROM images produced by the class loader are on average one-third the size of their corresponding classfiles, and (4) the class loader software can be developed in a high-level language – in this case, a higher-order strategic programming language ameniable to formal verification.
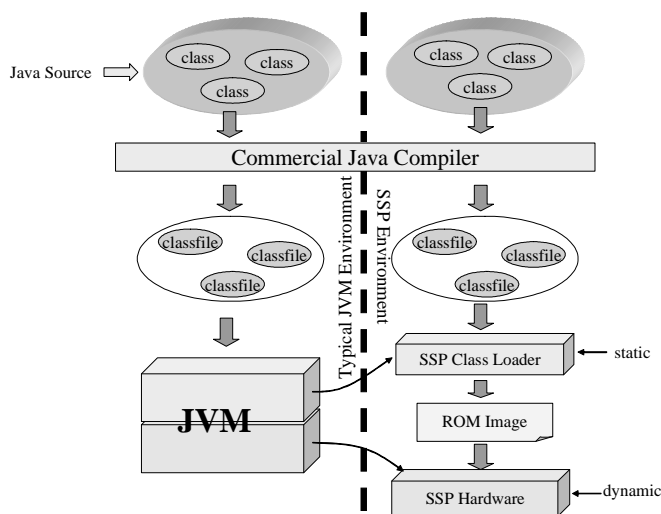

Figure 2: Factoring the Class Loader out of the SSP Hardware

2 Hardware: The SSP

Unlike commercial processors, the SSP was NOT designed for high throughput, but rather for safety and security. The design of the SSP was intended to be small, simple, and analyzable. In fact the core architecture was intended to not only to be conceptually simple, but yield a relatively simple hardware design as well. For those reasons, it has no pipelining or caching capabilities. The end result has been a hardware design having only 35K logic gates for the SSP implementation (not including embedded memories) and is capable of running at nearly 25Mhz.
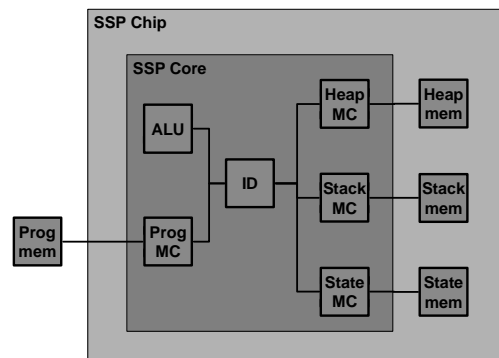

Figure 3: Overview of the SSP Architecture

The design of the SSP is based on the philosophy that memory usages which are *conceptually distinct* should map to memory modules which are *physically distinct*. As a result, the design of the SSP closely follows the specification given for the Java virtual machine. For example, in accordance with the architecture neutral specification of the JVM, the SSP has divided what in common practice is implemented as a single physical memory into a number of independent ones. In the SSP what is traditionally is known as the *heap* has been implemented by a memory module that is physically independent from the memory module implementing the *stack*. This makes it physically impossible to crash the heap memory into the stack memory as can occur in traditional microprocessor memory layouts. Furthermore, it obsolesces all analysis related to processor behavior which could result in the case of such memory overruns, deferring the judgement to first principle design[23].

In accordance with the philosophy of physical separation, the design of the SSP includes a separate memory used to manage context switches resulting from method calls. The resulting memory module is called the *state memory* and represents an abstraction

that goes beyond the architecture neutral model of the JVM. Figure 3 below shows the basic architecture of the SSP. Note that all of the independent memories are on-chip and the application program memory is off-chip.

2.1 The Current System

The hardware implementation of the SSP ASIC ultimately yielded a 110K gate system with 110K embedded memory bits where the core SSP processor was responsible for roughly 35K gates of the total system. Transferring the digital design of the SSP into one capable of being printed on a custom ASIC was remarkably smooth. Each of the major internal SSP and I/O blocks were registered to simplify timing and synthesis efforts. The designers were rewarded with a system that was capable of being synthesized and laid out directly from the top level, in contrast to more traditional designs which require much of this work to be completed as sub-blocks and then integrated together.

The total synthesis time spent was just short of 3 weeks, where comparably sized ASICs at Sandia typically take from 6 to 18 months. The synthesis work resulted in a design where 100% of the 8000 registers were covered by the automatic insertion of 9 scan chains, and cover 98% of total logic. This work required roughly two days where typically it takes roughy a week to reach 70% logic coverage, and additional 6 weeks to even approach an end goal of 95% coverage. It also yielded a design with zero timing violations for a very aggressive target clock of 25MHz. Such results are very rare, as most synthesis outputs yield hundreds of timing violations, and each must be reasoned away.

The ASIC layout efforts were slightly more problematic. The difficulty came from a large number of internal 32 bit busses that led to high routing densities and difficulty meeting a useable die size.

2.2 Hardware Assurance

Validating the design of a new processor is not a trivial task. To minimize human-in-the-loop errors we sought to find the most automated way possible to verify that the low level digital design satisfied the Java level requirements. In short, an N-version programming solution was used. A copy of the SSP operational specification was distributed to three different development teams. Each implemented a solution in a completely different design paradigm described by:

- VHDL – a hardware description language that became the actual design for the physical hardware.

- Java model of the SSP Architecture – a high level object oriented Java implementation of the bytecode operation.

- ML model – a high level ML implementation of the bytecode operations. ML is a functional language with type inferencing.

Each of these implementations was executed on a large self-checking Java program designed to test all Java features supported by the SSP. The test program developed covers the general as well as special cases of each bytecode supported by the SSP. The tests are non-exhaustive in the sense that only representative elements of general and special cases are tested. Note that it would be virtually impossible to test every possible instance of every general or special case. For example, consider exhaustively testing every general case of adding two 32-bit integers or exhaustively testing every special case where the addition of two 32-bit integers causes an overflow in the ALU.

Some of the aspects of bytecode semantics tested include:

- all possible branch cases within the branching bytecodes (e.g., ifne, ifeq, etc.)

- multiple levels of inheritance (e.g., checkcast and instanceof)

- general cases for all the ALU operations (e.g., iadd, imul, ladd, ldiv, lshl, lushr, etc.)

- special cases for the ALU operations (overflow, underflow)

- access to all constant pool entries

- all supported exceptions as specified by the bytecodes (e.g., null pointer, divide by zero, array index out of bounds, etc.)

The resulting test suite resulted was named the *SSP-TestSuite* and resulted in the execution of 12,877 bytecodes. Initially the test suite was executed on each of the implementations of the SSP and final *states* compared. In this context a *state* consists of the state of each of the memories defined in the SSP design (e.g., *stack*, *heap*, and *state stack*) together with the value of all the registers accessing these structures.

Analysis discovered that a simple comparison of finals states allowed a number of errors to "slip through the cracks". For example, a computational sequence

in one implementation causing a bit to be flipped twice could not be distinguished from a computational sequence in another implementation having no bit flips.

The root cause of these problems was of course due to the fact that our test suite was only exhaustive with respect to representative cases but not exhaustive with respect to specific instances of those cases. Realizing this a decision was made to to perform an *n-way* comparison of all intermediate states encountered during the execution of the test suite. This comparison turned out to be too fine grained as it reported differences in implementation as errors. For example, for the "dup" bytecode, the VHDL implementation will read the top of the stack and copy it, thus performing one "read" and one "push". The Java and ML implementations will actually pop the top value off the stack and perform two pushes.
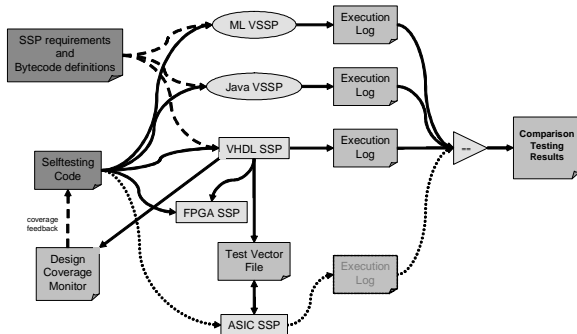

Figure 4: Diagram of N-version Testing

In general, the Java and ML implementations will execute a bytecode to completion before beginning the execution of the next bytecode in the computation sequence. However, the execution model in the VHDL implementation is more complex. The VHDL implementation fetches a bytecode, decodes it, and then sends commands to the various intelligent memory controllers. After a command has been sent to a memory controller, the VHDL implementation will immediately begin executing the next bytecode even though the particular memory controller might still be processing the previously sent command. The implementation will stall only if a command is made to a memory controller that is still executing a previous command. An example of this behavior can be seen in the execution of the "new" bytecode which will cause

the heap controller to allocate and zeroize heap memory. While the "new" bytecode is being executed by the memory controller, it is possible for the instruction decode to execute one or more bytecodes that only use the stack. Because of the parallelism involved, our test framework requires that the VHDL implementation record the bytecode associated with each memory controller command. This information is used in order to successfully distinguish "bytecode boundaries" so that a semantically accurate n-way comparison can be performed by our test oracle.

As a result of the differences in computational models, the choice was made to define an intermediate state precisely in terms of the before/after semantics used in the architecture neutral specification of Java bytecodes[10]. Each implementation is required to make this state sequence available to a test oracle for inspection. Conceptually, the implementations were designed to output their states before and after each bytecode execution. All state variables (all memory and register values) are compared before and after each bytecode execution, for each of the implementation. However, for efficiency reasons, state information is output in a compressed *relative* form. Specifically, only those memory locations *changed* by the currently executing bytecode are writen to an output file. The n-way comparison is automated by a test oracle which compares all of the output files for bytecode boundary equivalence. Figure 4 is a graphical representation of this process. Figure 4 also shows how this testing concept is carried into the hardware validation of the ASIC itself.

3 Software: The Class Loader

The specification of the JVM states that classes should be made available to a running program via a loading and linking sequence. The loading step consists of importing a class file into the JVM. A class file is the binary form of a class and is typically generated by a Java compiler. The linking phase can be broken down into three steps: (1) verification, (2) preparation, and (3) resolution. Verification ensures that a class is well formed. Preparation involves allocating the appropriate amounts of memory for classes and objects and assigning default values to these memory locations. Resolution is the act of transforming symbolic references into direct references with respect to a given hardware architecture.

The job of the SSP class loader is to correctly translate Java class files into a form suitable for execution by the SSP. This translation produces an inter-

mediate form that we call a *ROM image*, and concerns itself with many issues such as:

1. Resolving symbolic references to physical addresses or direct and indirect offsets. This resolution is intimately linked to the architecture of the hardware[1].

2. Correctly capturing the inheritance semantics of Java for applications consisting of classfile hierarchies.

3. Providing suitable information for method invocation and return.

4. Constructing method tables.

3.1 A Transformation-based Approach

The class loader for the SSP that is currently being used is written in Java. However, a research effort is underway to implement the class loader using a transformation-based approach. Analysis has revealed that a significant portion of the SSP class loader's functionality can be directly understood in transformational terms. Thus program transformation provides a paradigm well suited for expressing the class loader's functionality. Furthermore, program transformation admits the possibility of bringing the verification of the correctness of the class loader within reach of automated reasoning systems like ACL2. This is important, because *correctness* is the attribute of primary importance to the SSP class loader.

Let A_{SSP} denote the Java source code of an application written in the subset of Java supported by the SSP. Let C be a function denoting a trusted Java compiler. Let $JVM(x)$ denote the execution of the application x using the virtual machine JVM. Similarly, let $SSP(x)$ denote the execution of the application x using the SSP. Assuming the implementation of the SSP faithfully implements the semantics of the appropriate subset of the JVM, we say a class loader CL for the SSP is correct if and only if:

$$\forall A_{SSP} : JVM(C(A_{SSP})) \equiv SSP(CL(C(A_{SSP})))$$

where $CL(C(A_{SSP}))$ is the ROM image resulting from applying the class loader CL to the class files produced by $C(A_{SSP})$.

---

[1] One of the unique aspects of this project has been the close relationship between the classloader development team and the SSP development team. Both teams have frequent contact and are willing to negotiate design complexity issues in favor of increasing the analyzability of the overall system.

3.1.1 Transformation

Program transformation is an approach to software development that, in spirit, is based upon the notion that equals may be replaced by equals. An example of such a replacement in the class loader would be that a *symbolic reference* may be replaced by an "equivalent" memory *address* or *offset*. In a transformational framework, such replacements can be specified through rewrite rules of the form: $lhs \rightarrow rhs$. Thus the symbolic reference replacement could be abstractly expressed by the following rewrite rule:

$$symbolic\ reference \rightarrow corresponding\ address$$

A *strategic programming language* is a language that supports a syntax for expressing rewrite rules and provides operators for controlling their application[1][3][8][9][19][20][21]. This paradigm provides an elegant framework for specifying how a collection of class files can be altered by the application of a set of rewrite rules which realize the class loader's functionality.

In the context of the class loader, control is necessary to restrict where rules should be applied. For example, rewrite rules for resolving the indexes found in constant pool entries should only be applied to the appropriate constant pool, and so on.

From the perspective of strategic programming, the class loader problem is distinguished by the following characteristics:

- Class files are complex structures.

- Data frequently needs to be distributed between unrelated portions within a class file. For example, information from the constant pool needs to be distributed over the field and method sections.

- Data also needs to be distributed between classes within the application. Sometimes this data should only be distributed within an inheritance hierarchy and sometimes the data should be distributed throughout the entire application.

In a general fashion, we have characterized the notion of data distribution with respect to strategic frameworks. We introduce the term *distributed data problem* to describe such activity.

3.1.2 Higher-Order Transformation

First-order strategic systems often encounter difficulties expressing the distribution of data between complex structurally unrelated terms. Thus, even though

a transformation may be conceptually straightforward, its realization within a first-order framework becomes complex. At the conceptual level, rewrite rules are oftentimes straightforward because one can envision a strategy consisting of rewrite rules containing problem specific constants (e.g., a field entry in the constant pool of a class and a corresponding offset for that field). Unfortuntely, such problem specific constants change from problem to problem (e.g., different class files will have correspondingly different constants). In a first-order setting, this type of change is most often captured through rule parameterization. Thus rule instances are not actually created, but rather their effect is simulated through a framework of accumulation, parameterization, and associated lookup functions. This has the undesirable side-effect of making rules more complex than conceptually they need to be. However, this problem can be avoided in a higher-order rewriting framework. In a nutshell, the advantage offered by higher-order rewriting is that collections of first-order rewrite rules (containing constants) can be generated dynamically. Thus the need for accumulation, parameterization, and associated lookup is subsumed by dynamic rule creation and rule application. Our experiences have shown[24] that in a higher-order framework, the distribution of data can oftentimes be elegantly expressed – mirroring the simplicity of the conceptual understanding. As a result, we have developed a higher-order strategic system called HATS[22] that implements the ideas described here.

3.2 Providing Assurance in the Class Loader

The class loader is a weak link in the assurance chain of the SSP. Commercial compilers take Java source code and produce class files. The assurance provided by a commercial compiler stems from several sources including the fact that the Java community at large performs an extensive stress test of the compiler. Over time, such a testing environment causes a software product to mature. Granted bugs may still exist in the compiler, but all things being equal, the likelihood of encountering a bug in the class loader is significantly greater. Thus we are devoting considerable effort towards providing assurance in the translation performed by the SSP class loader. Our assurance activities fall into two categories: (1) *verification*, which uses formal reasoning to prove properties about the class loader in general, and (2) *validation*, which discretely inspects individual outputs of the class loader

to demonstrate properties in particular.

3.2.1 Verification

Formal verification provides a framework where it is possible to demonstrate that a system such as the class loader possesses general properties. In this context, assurance comes in the form of a mathematical proof and typically involves a model of the system under analysis (rather than the actual system itself). General properties are stated in terms of theorems involving the model. The proof of theorems provides strong assurance that the model behaves as required. Assurance of the correctness of the system under analysis relies on (1) confidence that the proofs themselves are sound, and (2) confidence that the model faithfully describes the system (e.g., theorems that hold for the model hold for the system). While it is theoretically possible to automate the construction of proofs, in practice it is extremely difficult and requires sophisticated tools and approaches.

We are using the modeling and verification framework provided by ACL2 to formally prove theorems about the class loader. ACL2 [6][7] is a programming language based on the applicative subset of Common Lisp. In this language, users can build executable models of software systems. ACL2 is also a tool that assists users in proving theorems about their ACL2 programs. It has been used to prove the correctness of hardware implementations of microprocessors and floating point algorithms [2][16]. More recently it has been used to verify parts of implementations of the JVM [13][14][15][17].

Our approach to modeling the class loader is based on a heavily-researched model[2][13][14][15][17] in which a system is described in terms of states and state transitions. In the past, this type of model has been extensively applied to reasoning about the behavior of low-level assembly-based computations[12]. More recently, this type of model has been used to reason about the behavior of computations described in terms of Java bytecodes. In our research, we are adapting this model in a novel way to the computational paradigm offered by strategic programming. In the context of the class loader, we model the class files of the Java application as our *state* and the transformations on this application as our *state transitions*. The JVM and the SSP provide the basis for formally understanding equivalence between states.

A model of HATS is constructed by defining an *abstract machine* that controls the application of transformation rules. Each transformation rule modifies the state of the system. The abstract machine op-

erates according to the following sequence: fetch the next transformation rule and node from the current state, apply the transformation, and return a new machine state.

Though we are exploring the verification of a number of properties of the classloader, our ultimate goal is to verify that the transformation rules preserve the meaning of the term to which they are applied (i.e., the class loader is correct). In the context of the SSP, the initial term is a set of class files, C_0, generated by a Java compiler. The semantics of this term is defined by the JVM specification. We can think of the JVM as defining a mapping from (classfiles $\times$ inputs) to outputs. Let $Eval_{JVM} : classfiles * inputs \rightarrow outputs$ denote this mapping function. $Eval_{JVM}$ defines the behavior of the program encoded in the classfiles. The final term is a *ROM image*, which we denote C_{ROM}. The semantics of this term is defined by the SSP hardware, $Eval_{SSP}$. HATS accomplishes the conversion of C_0 to C_{ROM}, as indicated by the notation $C_{ROM} = T^*(C_0)$. In this notational framework, what must be shown for inputs I is:

$$\forall(C_0, I) Eval_{JVM}(C_0, I) = Eval_{SSP}(T^*(C_0), I)$$

The problem above can be decomposed by defining a sequence of normal forms, $C_0, C_1, C_2, ...$ in the transformation of C_0 to C_{ROM}. These normal forms are formally specified and are theorems within our verification framework. *Constant pool normalization* and *field distribution* are two examples of normal forms. Informally stated, in constant pool normalization all indirection is removed from the constant pool entries of the classfiles in C_0. Let T^1 denote the transformation that accomplishes this task. Similarly, let T^2 denote the normal form resulting from field distribution. At present, a sequence of five intermediate normal forms have been defined. For each normal form, there is an evaluation function, $Eval_n$. Thus, the original correctness conjecture can be restated as a sequence of conjectures:

$\forall(C_0, I) Eval_{JVM}(C_0, I) = Eval_1(T^1(C_0), I)$
$= Eval_2(T^2(T^1(C_0)), I)$
$...$
$= Eval_{SSP}(T^*(C_0), I)$

where T^* is the composition of the individual transformations. This allows the proof to be constructed incrementally, and therefore, reduces the complexity of the proof.

3.2.2 Validation

From the perspective of validation, the class loader is viewed as a class file to ROM image compiler whose trustworthiness is in question. In an approach similar to the one taken for Java class files, information is embedded in (binary) ROM images that enable their semantic structure to be recovered using simple parsing techinques. This enables ROM images to be subjected to tests similar in spirit to those performed on Java class files by byte code verfiers. A ROM Integrity Checker (RIC) has been developed to perform such tests. The embedded nature of the SSP enables additional checks to be performed in which ROM images are compared to the class files which generated them. This type of checking goes beyond what is possible for byte code verifiers.

The ROM Integrity Checker   The Rom Integrity Checker (RIC) operates entirely within the bounds of a single ROM image – that is to say, it does not examine source code or class files, only the image itself. Since SSP ROM images include parsing information, RIC can parse the image into its various components (method areas, method tables, methods, constant pools, instance field elements, and so forth). At this point, several checks are run on each component. An informal description of some of the checks is given below.

- The parent address must point to a class (or 0 for Object).

- The class must have at least as many methods as its parent.

- For each method:

  1. The program counter and constant pool addresses of an inherited method must be consistent among all classes that inherit the method.

  2. Methods defined within a given class must have a program counter that falls within the class and a constant pool address that points to the constant pool of the current class.

- Only one class in the ROM image can be designated as Object.

- The branches in byte codes must not leave their method.

- The byte code *new* must take a class element as its argument.

All in all, over 80 distinct checks are provided covering broad structures such as method areas and constant pools and going all the way down to the bytecodes themselves.

The Class Loader Integrity Checker The consistency properties checked by RIC are all properties that are necessary for ROM Image to be well formed. However, since RIC does not consider any context beyond the image itself, RIC is limited in the classes of errors that it can detect. For that reason, we have defined an additional set of checks between ROM Images and the class files from which they are generated.

This resulted in the development of a new tool: the Class Loader Integrity Checker (CLIC). The first stage of CLIC's execution is to run RIC against the ROM image. If RIC is claims that this is a valid ROM image, then CLIC parses a set of Java class files (specified by the user) from which this ROM image is alleged to have been created. After the class files have been parsed, comparisons are made between the class files and the ROM image. A mapping is establishes between classes, allowing us to compare their contents one by one. (The ROM image format was actually extended slightly to allow this mapping to occur more easily). CLIC performs a structural comparison of the class heirarchy defined by the Java class files with the class hierarchy in the ROM image. Similarly, methods and instance fields are verified to correlate between the two structures.

Notably, CLIC performs numerous, very detailed checks of the individual bytecodes and their arguments. For example, the targets of invoke instructions are resolved, and checked to ensure that they are pointing at the correct classes. Bytecodes in each method are checked "one by one" and must correspond exactly with the class files. All in all, CLIC expands the number of distinct checks performed to over 190.

Testing the Checkers The SSP-TestSuite (see Section 2.2) was converted into a ROM image by the class loader. We then took this image and modified it by hand to create new images. Each of these images was wrong in some way, with respect to the original class files. Many of these images were designed with malicious intent to trick RIC/CLIC, but the effort did not go beyond modifying more than a few bytes in each file.

Hand crafted images are reasonable for many purposes, but we would also like to be able to test more than just what we come up with by hand. CLIC was not particularly slow (each test run took roughly 5-7 seconds on a 1GHz Pentium 3), and so we devised a program to sequentially flip every bit of a ROM image (one at a time) thereby producing a new ROM image. Let R_0 denote an initial $32K$-bit ROM image produced by applying the SSP class loader to the SSP Test Suite. The set of ROM images produced was:

$$ROM_{set} = \{R_0, R_1, R_2..., R_n\} \text{ where } n = 32K$$

Note that for all $i > 0$, R_i will differ from R_0 by exactly one bit. Each of the ROM images produced was checked by CLIC to see if it was able to catch the error resulting from the bit-flip. Split across a few computers, the testing of this ROM image sequence was completed in a few days. RIC itself was able to detect around 70% of the errors, and CLIC was able to detect 100%. Note that these figures are the results for these particular images – a different set of images will surely yield different results. Of course, this is not a proof that CLIC catches 100% of errors, and it may well be the case that there is some error which CLIC does not detect.

We would like to mention that catching any single bit flip is not a proof of correctness. The implications of CLIC's checks with respect to correctness is something that we are currently investigating. We believe that it is definitely possible, even likely, that there are some problems that CLIC will not be able to detect. Nevertheless, these tests seem to be fairly strong evidence that CLIC is capable of detecting a large class of errors.

Summary and Limitations Images that pass through CLIC verification have passed a great number of tests. Taken as a whole, these tests can be considered "necessary but not sufficient" for correct translation of the class files. These tools appear to be quite useful – it would seem to be unlikely that an unsophisticated bug in the class loader could output a ROM image that would pass CLIC verification. RIC and CLIC are relatively simple, together comprising only about 6700 lines of liberally commented, straightforward Java code.

Yet RIC/CLIC are a far cry from formal proof of correctness. We do not yet have the tools to formally prove properties of complex Java programs such as the class loader. Furthermore, the testing we have done of RIC and CLIC, perhaps extensive by human standards, is nowhere near exhaustive. Indeed, the problem space is so large as to be virtually infinite (with 32Kb per image, there are over 2^260000 possible images).

4 Conclusion

The SSP offers a design space in which significant tradeoffs can be made between hardware and software. Much of the success of the SSP can be attributed to the diverse nature of its design team whose members have expertise in (1) system design, (2) hardware design and development, (2) software design and development, and (3) formal methods. Additionally, the members of this team share the perspective that the primary design requirement is correctness and analyzability.

Assurance efforts for the SSP are ongoing. We see the correctness argument as consisting of a chain having numerous links (e.g., the correctness of the Java compiler, the correctness of tools used to develop various portions of the system, etc.). Our overall plan is to focus our efforts on the weakest links of this chain.

References

[1] P. Borovansk, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. *An Overview of ELAN*. In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, France, 1998. Elsevier Science.

[2] Robert S. Boyer and Yuan Yu. *Automated proofs of object code for a widely used microprocessor*. Journal of the ACM, 43(1):166-192, January 1996.

[3] M. van den Brand, P. Klint, and J. Vinju. *Term rewriting with traversal functions*. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.

[4] J. Child. Java Proving Itself Worthy for Defense Apps. COTS Journal, pp25-28, July 2003

[5] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper.* http://java.sun.com/docs/white/langenv/index.html

[6] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[7] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: Case Studies.* Kluwer Academic Publishers, June 2000.

[8] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies.* Journal of Logic and Algebraic Programming, Vol 54, pp 1–64, 2003.

[9] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming.* Draft.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2nd Edition.* Addison-Wesley, Reading, Massachusetts, 1999.

[11] J. A. McCoy. *An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software.* Proceedings of the 5th IEEE International High-Assurance Systems Engineering Symposium, Nov. 2000.

[12] J S. Moore. *Piton – A Mechanically Verified Assemply-Language.* Kluwer Academic Publishers, 1996.

[13] J S. Moore. Proving Theorem about Java-like byte code. In E.-R. Olderog and B. Steffen, editor, correct system design-Recent Insight and Advances, pages 139-162, Heidelberg, 1999. LNCS 1710

[14] J S. Moore. Proving Theorems about Java and the JVM with ACL2. Models, Algebras and Logic of Engineering Software, M. Broy and M. Pizka (eds), IOS Press, Amsterdam, pp 227-290, 2003.

[15] J S. Moore and Robert S. Boyer. Mechanized Formal Reasoning about Programs and Computing Machines, with, in R. Veroff (ed.), Automated Reasoning and Its Applications: Essays in Honor of Larry Wos , MIT Press, 1996.

[16] J S. Moore, Tom Lynch, and Matt Kaufmann. *A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm.* IEEE Transactions on Computers, 47(9), pp. 913-926, Sep., 1998.

[17] J S. Moore. *Proving theorems about Javalike byte code.* In E.-R. Olderog and B. Steffen, Eds., Correct System Design-Recent Insights and Advances, LNCS 1710, pp. 139-162. Springer-Verlag, Berlin 1999.

[18] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag 1990.

[19] E. Visser. *Scoped dynamic rewrite rules.* In M. van den Brand and R. Verma, editors, Rule Based Programming (RULE'01), volume 59/4 of Electronic

Notes in Theoretical Computer Science. Elsevier Science Publishers, September 2001.

[20] E. Visser. *Language Independent Traversals for Program Transformation.* In Johan Jeuring, editor, Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, July 2000.

[21] E. Visser. *Strategic Pattern Matching.* In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).

[22] HATS – http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html

[23] V. L. Winter, J. M. Covan, and L. J. Dalton. *Assuring Passive Safety in High Consequence Systems.* IEEE Computer Vol 31, No. 4, April, 1998, pp 35-36.

[24] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software.* Advances in Computers vol 58, to appear.