Type-based Assessment of the Java Byte-level Language

Michael Anthony Smith
(M.A.Smith@eris.dera.gov.uk)
Systems Assurance Group
Defence Evaluation and Research Agency

14th September 1998

1 Introduction

The Systems Assurance Group (SAG) is involved in software assessment for security and safety related (or critical) systems. During, the last year the Java byte-level language has been under investigation. The aim has been to understand the subtleties of this language, to provide a basis for assessing software systems that use the Java technology. This requires an accurate and unambiguous interpretation of the Java Virtual Machine (JVM) specification [1]. Therefore, it was decided to start by providing a formal type-based description for each of Java's byte-level operations; as such rules (with an appropriate environmental model) can unambiguously capture many of the specified properties (e.g. memory integrity).

This document presents an overview of SAG's initial approach. It starts with an outline of the approach, followed by a brief discussion of the related work. Then it describes the constant aspects of the formal model that underpin the approach, which includes an overview of the type-system and class file inheritance structure. This is followed by a description of the concrete and abstract representations of a class-file's method, indicating why both are desirable (although not strictly necessary). Next, an outline of the structure of the type-based rules (including a few examples) is presented. Finally, a summary of what has been achieved and SAG's intentions for the future is given.

1.1 Overview of the approach

SAG's approach is aimed at software assessment rather than development. It is targeted at software products that the assessors have no control over, such as commercial off the shelf (COTS) products. Therefore, the only valid reasons for rejecting such a product are: that it can be shown not to satisfy a given safety (or security) property; or that it is not cost-effective to demonstrate such a property. This means that an assessment tool based on this approach should aim to cope with the full complexity of the byte-level language, rather than enforcing a particular style of programming (e.g. no overlapping exception handlers).

SAG's approach is based on applying a type-based theory to each of Java's byte-code operations, including the exception handling mechanism; as the exception handling mechanism is an integral part of the Java language, and thus cannot be avoided or ignored.

To date the work has focussed on investigating a single method within a class-file. This has produced both a concrete and an abstract model of the Java class-file's method format, along with a collection of type-based rules for most of the bytecode operations (i.e. a method's algorithm). In fact, over 190 of the 200 bytecode operators currently have associated type-based rules. It is believed that these rules guarantee the integrity of the virtual machine's memory, provided that its implementation is correct. In other words, these rules should guarantee that:

- the memory is only accessed in a type consistent manner;
- the bytecode stack does not over-flow;
- the bytecode local variables exist.

A proposal to extend SAG's initial investigation into Java's byte-level language is being considered by DERA. In particular, to specify and implement an automated assessment tool, based on an abstract execution of the type-based rules. This will provide an initial validation of these rules, as well as providing a good starting point for the assessment of Java within critical systems.

1.2 Related work

The type-based model presented has been influenced by the paper entitled "Type System for Java Bytecode Subroutines" by Raymie Stata and Martin Abadi [5]. In particular, it demonstrated a way of presenting intuitive type rules. In fact, several researchers are using formal methods to model various aspects of the Java programming and byte-level languages. Each of these models is aimed at solving a particular type of problem. The approaches (that SAG are aware of) can be broadly classified in two categories, type-theoretic and semantic.

The related type-theoretic approaches (e.g. [5, 6]) have focussed on validating a small sub-set of the Java byte-level language. They specifically ignore exceptions and the differences between the various numeric data types. Thus, this category of approaches is unsuitable for a basis of an automatic assessment tool, because they are too restricted.

The second category of approaches has been investigating various methods for formally defining the semantics of the Java byte-level language (e.g. [7]). These approaches are inappropriate for forming the basis of an automatic assessment tool, as they do not abstract the complexity of the Java byte-level language to an acceptable level for efficient automatic evaluation.

In summary, the two main differences between the approach presented and the other type-theoretic approaches are that this approach:

1. will model all 200 bytecode operations, whereas the other approaches only model small subsets, typically 10-20, of the bytecode operations;
2. already models exceptions, whereas the other approaches ignore exceptions.

2 Constant Aspects of the Formal Model

Several aspects of the formal system are considered to be constant, these include: the syntax of a class-file; the allowable types and ranges of the numeric values; and the allowable bytecode operations.

The remainder of this section provides a brief description of both the type-system with its associated abstract values, and the class-file environment. Note that the abstract value space is described rather than the concrete value space, as the abstract values are used in the definition of the type based rules that follow in section 4.

2.1 Type System

Sun's JVM specification indicates that there are two types of value that can be assigned to a Java variable[1]: primitive for numeric values; and reference for class, object, interface and array values. The JVM specification does not have an explicit method type, as methods cannot be assigned to a Java variable. However, for the formal representation it is useful to introduce an explicit method type as well as a program counter (address) type. In fact the formal model partitions the types into six sets[2]:

1. primitive type set – represents the boolean and numeric values;
2. class type set – represents the class, object and interface identifiers;
3. array type set – represents an array and the type of its elements;
4. method type set – represents the method's formal parameters and result type;
5. program counter type set – represents an offset to a method's address;
6. void type set – represents an undefined local variable and a method with no return value.

This formal type system underpins the formal model. It is used by both the concrete and abstract representations of the Java class-file. However, the concrete and abstract value spaces associated with the various types differ.

2.2 Abstract Value Spaces

The abstract value space is constructed from pairs of types with their associated abstract values. For example, the whole number abstract value space has five types (i.e. byte, char, short, int & long) and five values (${}^{-}1_A, 0_A, {}^{+}1_A, \not{0}_A$ & $\mathcal{V}_A$) where:

${}^{-}1_A$	representing minus one and the exclusive or mask for not operation;
0_A	representing zero and false;
${}^{+}1_A$	representing one and true;
$\not{0}_A$	representing non-zero number;
$\mathcal{V}_A$	representing any number.

Thus operations such as subtract can be defined by:

sub_A	${}^{-}1_A$	0_A	${}^{+}1_A$	$\not{0}_A$	$\mathcal{V}_A$
${}^{-}1_A$	0_A	${}^{-}1_A$	$\not{0}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$
0_A	${}^{+}1_A$	0_A	${}^{-}1_A$	$\not{0}_A$	$\mathcal{V}_A$
${}^{+}1_A$	$\not{0}_A$	${}^{+}1_A$	0_A	$\mathcal{V}_A$	$\mathcal{V}_A$
$\not{0}_A$	$\mathcal{V}_A$	$\not{0}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$
$\mathcal{V}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$	$\mathcal{V}_A$

Note that: the decimal numbers (i.e. float & double) and void types have only one value $\mathcal{V}_A$; the reference types (i.e. class & array) have an integer value which refers to a list of objects for that method; the method type has its name as its value; and the program counter type has a number (address) as its value.

[1] A Java variable is either a class's field, an object's field or a method's local variable.

[2] The program counter and void types are represented by a singleton set.

2.3 Class-file environment

When analysing a method it is necessary to work out whether one object is an instance of another object. So, a class-file environment is created to model the relationship between the class-files of a Java program.

Some Java programs can be interactively directed by the user to load named class-files, thus there is no general mechanism for statically determining what class-files a program requires. However, any Java program that allows a user to load named class-files must insist that these class-files either implement a given interface or extend a given class. Therefore, it is possible to split a Java program into *parts* that communicate via common interfaces or classes; which can then be analysed in isolation. This separation of concerns allows the class-file environment, for a given *part* of a Java program, to be modelled by a function from a class-file's identifier (fully qualified name) to an appropriate abstraction of the class-file.

The basic structure of the class-file environment has further restrictions placed on it, such as the parent of every class-file must either be null (non-existent) or a member of the class-file environment.[3] These restrictions make sure that the modelled class-file structure is consistent and complete.

When statically analysing one part of a Java program it is reasonable to model the class-file environment as a constant entity, because the analysis is only valid for a given instance of the class-files. This implies that a new analysis has to be undertaken each time one of these class-files is changed. Further, modelling the class-file environment as a constant considerably simplifies its formal definition whilst increasing its clarity.

Note that there is no type-based configuration problem between the various parts of a Java program, as they rely on and communicate via at least one common class-file. Therefore, all that needs to be checked is that all the parts of a Java program's analysis are up-to-date.

3 Concrete and Abstract Representations

Before the type-based rules for each of the byte-code operations can be written, a model of a method and its run-time environment has to be constructed. This enables these rules to state under what environmental conditions they apply, and if they apply how they affect the environment. In other words, the type-based rules describe the method environment's state transition operations.

The remainder of this section defines both a concrete and an abstract representation of a method's environment. It also discusses the significant differences between them, and why the abstract environment is a more suitable model for the intended analysis.

3.1 Concrete Representation

The concrete representation of a class-file faithfully represents the binary class-file information in a human readable form. In other words, there is a bijective map between the concrete and binary representations. In particular, the class-file's constant pool is not expanded into the method definition.

[3] The only class that does not have a parent is "java.lang.Object".

Each method that is defined in a class-file has an associated byte-code block that contains the executable instructions. Intuitively, the byte-code block is a sequence of byte-code operations. However as some of the byte-code operations have in-line static parameters, the number of bytes between operations in a byte-code block varies.

The concrete representation of a byte-code block uses the same program counter positions as the binary representation, in order to maintain a clear bijective mapping between them. Thus, the concrete representation of the byte-code block cannot be modelled by a sequence of byte-code operations. Instead, it is modelled by a function from a program-counter (unsigned 16-bit number) to a byte code operation; where the first operation starts at 0, the next operation at 0 plus the size of the 0^{th} operation, etc.

The concrete representation of a method's exception table mirrors the binary representation, which is a sequence of exception table entries. These entries are traversed from first to last when an exception is called to see if it is handled at this point in the byte-code block. Thus, it is a simple matter to write a function, called *ex_handled*, that indicates whether a particular exception is handled at a specific program-counter. This function is required for the type-rules to handle the exceptions, where it returns the program-counter of the handler (if it exists). When no-handler exists the *ex_handled* function returns either -2 or -3, depending on whether such an exception is allowed to be thrown by the method or not respectively.

3.2 Abstract Method Definition

The abstract method definition is similar to the concrete method definition, except that:

- references to constant pool values are expanded;
- abstract values replace concrete values;
- the byte-code block is converted into a sequence of abstract operations;
- the exception list is modified to reflect these changes.

The conversion of the byte-code block to a sequence of abstract operations is non-trivial, as it can change the value of the program counter for a given byte-code operation. This means that all the branching operations, such as if and goto, must suitably modify their destination program counters; i.e. the graph of the byte-code should remain unchanged.

The complications of converting the concrete representation to the abstract representation are worthwhile as they significantly simplify the formal model. For example, as there is no concept of the size of an abstract operation the next program counter can be obtained, for the successful execution of non-branching operations, by incrementing the previous program counter by one.

4 Method type-based rules

Each abstract method is defined by its name, type, byte-code operation list, exception handling list, stack size and local variable block size. When the method is invoked the run-time environment dynamically allocates at least enough memory for both the stack and local variable blocks (as defined by the method's stack and local variable block sizes). The method invocation routine then proceeds to copy both the implicit[4] and explicit parameters into the front of the

[4]Each object method implicitly has the "this" object as its first parameter.

variable block. Next it starts the run-time environment's method execution routine by setting the method's program counter to 0.

The analysis method presented here initialises each of the blocks as the run-time environment would, except that it uses abstract values instead of the decimal values. The analysis then continues to perform an abstract execution of the method, to determine if all paths through the code are valid. This execution process is based on a series of type-based rules which indicate when and how each operation affects a method's environment.

The method environment extends its definition by modelling the current state of its stack and local variable store, by two lists of typed abstract values; where the values associated with reference types, in these lists, point to a position in an auxiliary list, known as the reference list. This provides a simple mechanism for tracking known aliases. For example, when applying one of the duplication bytecode operations to a reference element on the top of a method's stack, the associated type-base rule simply duplicates the pointer to the reference list.

The method environment also contains a subroutine call stack. Whenever a subroutine is called its associated type-based rule pushes the subroutine's return address on to the call stack. Thus, allowing the type-based rules to check that there are no recursive subroutine calls, and that the subroutine return operation points to a valid address.

The remainder of this section is split into two parts. The first part describes what a type-base rule is and how it is presented. The second part provides a simple example, which demonstrates how the abstract value space and type rules interact.

4.1 Type-based rule format

A type-based rule takes a method environment that satisfies its pre-condition and returns a method environment. The format for a type-based rule is shown below, where: md stands for the method definition; s for the current state of the stack; l for the current state of the local variables; r for the current state of the reference list; j for the list of (jump to) subroutine return addresses; and pc for the current program counter. Next, the rule indicates which byte-code operation must be at the program counter pc for it to apply. Then, given that the *Side Conditions* are true the type rule states that the new method environment is constructed from the same method definition md, but with a new stack s', local variable store l', reference list r', jump return list j', and program counter pc'.

$$\frac{md(s,l,r,j)[pc] = op(arg_1,\ldots,arg_n)}{md(s',l',r',j')[pc']}name\_mr_A \quad \boxed{Side\ conditions}$$

Note that some_mr is the name given to the above type-based rule.

The simplest type rule is for the nop operation, i.e. it does nothing except increment the program counter for the next instruction. Thus the stack, local variable store and reference list remain unchanged.

$$\frac{md(s,l,r,j)[pc] = nop\_op_A}{md(s,l,r,j)[pc+1]}nop\_mr_A$$

Note that the nop operation has no side conditions as it is always allowed (i.e. the side condition is implicitly true). The only interesting feature of this operation is that it allows the program counter to be incremented past the end of the method's byte-code block. In such a situation there would be no rule associated with the new program counter, so no further progress could be made. Successful termination of a method is achieved if either a *return_op* operation is successfully completed or unhandled exception occurs, in which case the program counter is set to -1 or -2 respectively.

4.2 If Integer Comparison Branching Operation

The *if_zero_op* and *if_int_op* operations compare two integer values to determine whether one is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to the other. The if_zero_A function takes a token that represents one of the six styles of comparison and an integer abstract value. It returns either false ($\mathsf{0}_A$), true ($^+\mathsf{1}_A$) or unknown ($^-\mathsf{1}_A$), as recorded by the table below.

if_zero_A	$^-\mathsf{1}_A$	$\mathsf{0}_A$	$^+\mathsf{1}_A$	$\cancel{\mathsf{0}}_A$	$\mathcal{V}_A$
$equalTk$	$\mathsf{0}_A$	$^+\mathsf{1}_A$	$\mathsf{0}_A$	$\mathsf{0}_A$	$^-\mathsf{1}_A$
$noteqTK$	$^+\mathsf{1}_A$	$\mathsf{0}_A$	$^+\mathsf{1}_A$	$^+\mathsf{1}_A$	$^-\mathsf{1}_A$
$lessTk$	$^+\mathsf{1}_A$	$\mathsf{0}_A$	$\mathsf{0}_A$	$^-\mathsf{1}_A$	$^-\mathsf{1}_A$
$lesseqTk$	$^+\mathsf{1}_A$	$^+\mathsf{1}_A$	$\mathsf{0}_A$	$^-\mathsf{1}_A$	$^-\mathsf{1}_A$
$greateqTk$	$\mathsf{0}_A$	$^+\mathsf{1}_A$	$^+\mathsf{1}_A$	$^-\mathsf{1}_A$	$^-\mathsf{1}_A$
$greatTk$	$\mathsf{0}_A$	$\mathsf{0}_A$	$^+\mathsf{1}_A$	$^-\mathsf{1}_A$	$^-\mathsf{1}_A$

The *if_zero_op* operation takes two static parameters (k – comparison token, pc' – the successful branch program counter) and one dynamic parameter (tv – the integer for comparison). The following type rules show when the comparison can be deduced to be true, and when the comparison can be deduced to be false; where the condition $tv.1 \in tkType(intTk)$ indicates that the type part of the (type, value) pair tv is an integer.

True case:

$$\frac{md(s^\frown\langle tv\rangle, l, r, j)[pc] = if\_zero\_op_A(k, pc')}{md(s, l, r, j)[pc']}\,if\_zero\_mr_A \quad \boxed{\begin{array}{l} tv.1 \in tkType(intTk) \\ if\_zero_A(k, tv.2) \neq \mathsf{0}_A \end{array}}$$

False case:

$$\frac{md(s^\frown\langle tv\rangle, l, r, j)[pc] = if\_zero\_op_A(k, pc')}{md(s, l, r, j)[pc+1]}\,if\_zero\_mr_A \quad \boxed{\begin{array}{l} tv.1 \in tkType(intTk) \\ if\_zero_A(k, tv.2) \neq {}^+\mathsf{1}_A \end{array}}$$

The if_int_op operation is similar to the if_zero_op operation except that it takes two dynamic parameters (tv_1 & tv_2 – the integers for comparison) and compares one against the other. The following type rules define this operation.

True case:

$$\frac{md(s^\frown\langle tv_1, tv_2\rangle, l, r, j)[pc] = if\_int\_op_A(k, pc')}{md(s, l, r, j)[pc']}\,if\_int\_mr_A \quad \boxed{\begin{array}{l} \{tv_1.1, tv_2.1\} \subseteq tkType(intTk) \\ j = sub_A(tv_1.2, tv_2.2) \\ if\_zero_A(k, j) \neq \mathsf{0}_A \end{array}}$$

False case:

$$\frac{md(s^\frown\langle tv_1, tv_2\rangle, l, r, j)[pc] = if\_int\_op_A(k, pc')}{md(s, l, r, j)[pc+1]}\, if\_int\_mr_A \quad \boxed{\begin{array}{l} \{tv_1.1, tv_2.1\} \subseteq tkType(intTk) \\ j = sub_A(tv_1.2, tv_2.2) \\ if\_zero_A(k, j) \neq {}^{+}1_A \end{array}}$$

5 Conclusions

5.1 Summary

The type-based rules appear to contain enough information to establish the integrity properties listed in the introduction, for the whole of the Java byte-level language. The remaining rules are being documented in a part-time Master's degree dissertation[5].

The analysis method that uses these rules to perform a type-based abstract execution, has yet to be written. Although, this should be straight forward, as it will use a tree-traversal algorithm that applies to all the type-rules whose pre-conditions are satisfied by the current method's environment to produce a sub-tree of new method environments. For the analysis to succeed each leaf in the tree must have a program counter that is either -1 (for a successful return) or -2 (for a successful unhandled exception).

5.2 Related work within DERA

DERA currently has several research projects involved with investigating the various Java (and Java based) technologies. For example, one project is looking at the vulnerabilities of using the Java Button technology for storing medical records on the battle-field, which is a part of a bigger project examining the vulnerabilities of the Java technologies as a whole. And another project is investigating the security implications of emerging technologies, such as Java, for future security architectures and requirements.

5.3 Future work

There is an internal proposal to extend the work being presented in this paper:

1. to cover the whole of a class-file rather than just the method definitions (and selected auxiliary information);
2. to provide a specification for the type-based analysis algorithm;
3. to implement the specified algorithm using Java - so that the algorithm can be validated against existing byte-code verifiers;
4. to extend the type-based type rules to check for other properties as well, such as sub-routine calling order and volatile variable analysis;
5. to adjust the algorithm specification and implementation to the changes in the type rules, and then re-validate it.

There is also the potential that this work will be used in the assessment of safety related (or critical) software development within the air-systems (and other) industries.

[5]The initial draft of the M.Sc. dissertation should be available by the end of September 1998.

Acknowledgements

The work presented is being examined as part of my M.Sc. in Software Engineering at Oxford University. I would like to thank my M.Sc. supervisor Dr. Jim Woodcock for his encouragement and guidance over the last six months. I would also like to thank my colleagues, in particular Dr. Colin O'Halloran who provided me with the opportunity of investigating Java's byte-level language.

References

[1] Tim Lindholm and Frank Yellin, *Java Virtual Machine Specification*, Addison Wesley, September 1996.

[2] James Gosling, Bill Joy and Guy Steel, *Java Language Specification*, Addison Wesley, August 1996.

[3] Ken Arnold and James Gosling, *The Java Programming Language*, Addison Wesley, May 1996.

[4] Jon Meyer and Troy Downing, *Java Virtual Machine*, O'Reilly & Associates, March 1997.

[5] Ramie Stata and Martin Abadi, *A Type System for Java Bytecode Subroutines*, SRC Research Report 158, Digital Equipment Corporation, June 1998.

[6] Stephn N. Freund and John C. Mitchell, *A Type System for Object Initialization in the Java Bytecode Language*, Stanford Computer Science Technical Note: STAN-CS-TN-98-62, April 1998.

[7] Zhenyu Qian, *A Formal Specification of Java^{TM} Virtual Machine Instructions for Objects, Methods and Subroutines*, To appear in 'Formal Syntax and Semantics of Java^{TM}', ed. Jim Alves-Foss, Springer Verlag LNCS, 1998.

[8] Dexter Kozen, *Efficient Code Certification*, Cornell University, January 1998.