2001

Formalizing Java bytecode verifier using Z.

Rui. Guo
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Formalizing Java Bytecode Verifier Using Z

By

Rui Guo

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Computer Science at the
University Of Windsor

Windsor, Ontario, Canada
2001

ABSTRACT

Java security is a problem that is raised sharply with its ubiquity through Internet. Sun provides two kinds of specifications of Java bytecode verifier, which is used to ensure the soundness of a program. One is a prose specification and the other is a reference to implementation. Neither of these two way is a good approach to solve the security problems.

This thesis proposes a new way to formalize Java bytecode verifier by using Z language. The formal method is usually a perfect choice to some novel, difficult and critical projects. As a model-based formal specification language, Z is a good language to model a well-structured system. Besides the well-structured way it can offer, Z can also support verification of the implementation based on its specification. This approach offers a formal basis for any operation that is crucial to security. This helps to avoid causing security holes. By giving formal description to each bytecode instruction, the verifier could check the typing correctness within a method at each step of runtime. On the other hand, a more flexible and clearer module is constructed, which is applied to the type checking system, not only for Java but also for other similar mechanism. This thesis focuses on two essential problems: subroutine and object initialization.

Since this thesis gives a core model to deal with the Java bytecode verifier, it is easy to be extended to include all the instructions of Java bytecode.

Keywords: Java security, bytecode verifier, Z specification, formalization, type system, data flow analysis.

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my advisor, Dr. Liwu Li. His patience and guidance help me complete this thesis and make it good.

Also, I appreciate Dr. S. Bandyopadhyay's being my internal reader. His invaluable advices and comments are the great helps to my thesis.

I would thank Dr. Xiang Chen for being the external reader from his busy schedules.

Specially, thank Dr. Xiaobu Yuan for serving as the chair of the committee.

Finally, I want to give my thanks to all my friends who give me a lot of support during my work.

TABLE OF CONTENT

INTRODUCTION

Java is one of the most popular programming languages in the world. The design of Java combines the experience from several object-oriented programming languages, like Smalltalk, C++, and CLOS, etc.

However, Java is distinguished from other object-oriented programming languages by compiling its source code into an intermediate code instead of a final executable code. Java Virtual Machine (JVM) is responsible for interpreting this intermediate code (it is also called bytecode) so that programs could be executed on various platforms, which have JVM installed on it.

In addition to the object-oriented characteristics, Java has success providing a network-oriented approach, which allows downloading code as needed. Based on the Virtual Machine mechanism, Java applets allow downloading executable code over a network on demand. This flexibility is one of the advantages of Java. But a potential hindrance for the widespread use of Java is a security problem. From the previous paragraph, it is obvious that the process of compiling and linking are separated. Compiled bytecode is stored in a file with extension "class", which is produced by the Java compiler. However, bytecode may be written manually or changed on hostile purpose. The bytecode downloaded from another machine via networks or even moved by disks all become the untrusted source of language. Such code may cause unexpected results, even some serious problems. It is very dangerous to run such untrusted codes without check. Sun's solution to such security problem is to provide Java bytecode verifier.

Java bytecode verifier does verification on all untrusted bytecodes during linking time. Only those that have passed the verification can be executed. Thus, any hostile code or code that seems dangerous is prevented from running on the machine, but the security problems are not solved completed.

Many flaws are found on Java security [CG00][Dea97]. For example, according to some security analysis, some protected information is exposed to the users with downloaded applets without being detected by the bytecode verifier. Another example is that the downloaded applets can even execute some arbitrary machine code. One of the reasons is there is no formal model of bytecode verifier. Those flaws are reported in many researchers' papers, for example, Coglio and Dean's.

There are two official specifications of JVM published by Sun. One is a prose description and the other is a reference implementation of verifier (they are available at the Sun's website). As a prose description, it is ambiguous, imprecise and hard to reason about. On the other hand, the implementation reference is also hard to understand and reason about. Those kinds of specification may lead to improper or incomplete understanding and implementation of bytecode verifier. Therefore, the demand for a formal method in specifying bytecode verifier is obvious.

This thesis proposes a novel way to formalize the Java bytecode verifier. We select Z as the main language. As a mature formal specification language, Z has advantages to specify structural system. Since Java bytecode verification is a stack-based process, it is a good way to formalize bytecode verifier using Z. This approach offers a formal basis for any operation that is crucial to security. It helps to avoid causing security holes. Secondly, a more flexible and clearer module is constructed, which is applied to the type checking system not only for Java, but also for other similar languages.

In the first chapter, an introduction of Java Virtual Machine is given including its architecture and the process how bytecode is verified. Some basic features of Z like the notations used in this thesis are introduced in chapter two.

The following part is the major work of verifier formalizations using Z. The formalization work is separated into three parts according to different difficult points. The first part of chapter three describes all those general instructions used for sequential and branch statements. Subroutine and object initialization are discussed in the second and the third part respectively.

At the end of this thesis, some related work is introduced. Also, a summary of this whole work is given.

1. Java Virtual Machine

Java Virtual Machine is at the core of the Java programming language. It is the name of the engine that actually executes a Java program and is the key to many of Java's features, including portability, efficiency and security, etc.

"Java is called byte-compiled language or semi-compiled language" [Yel99]. From this description, it is obvious to see that the compiling process of Java is a little different from other traditional compiling methods. For a Java program, the program text is first parsed, some analysis is performed, and then a class file is created containing the Java Virtual Machine instructions needed for this program. Second, the data is iterated over, executing the JVM instructions. The set of instructions executed on JVM is just like the set of instructions running on a real machine. JVM is the real machine that Java programs run on.

The term "Java Virtual Machine" was coined by Sun to refer to the abstract specification of a computing machine designed to run Java programs, which is defined by the Java Virtual Machine specification [Yel99]. This specification includes topics such as the format of Java class files and the constraints of each instruction. Concrete implementations of the JVM specification are required to support these semantics correctly — these implementations are known as Java runtime systems. This thesis is to specify those semantics using specification language Z.

1.1 ARCHITECTURE OF JVM

1.1.1. General Architecture

Figure 1 gives a clear view of the architecture of Java Virtual Machine. Execution engine is the major part to actually carry out the instructions contained in the bytecode of a Java method. A runtime system can have many execution engines, but only one for a thread.

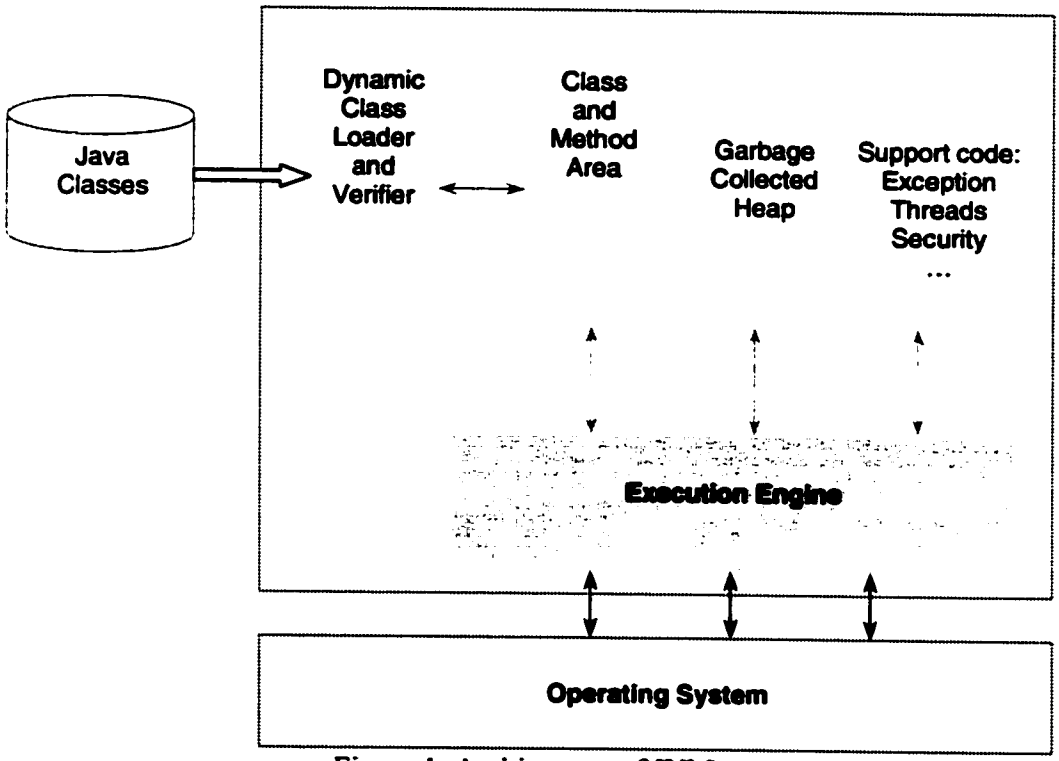


Figure 1. Architecture of JVM

The class loader loads the compiled classes and then the verifier does type checking on the instructions within a method. The method area stores structures of all those classes, such as runtime constant pool, field and method data, and the code for methods and constructors, etc. This area is shared among all Java Virtual Machine threads.

Every class or interface owns a runtime constant pool. Constant pool has the entry to each class that is contained in the class file and contains several kinds of constants. Each constant pool is allocated from the method area when the class or interface is created.

Memory management includes heap to store the data information for objects and arrays, which is also shared among all threads. The spaces allocated in heap can be released automatically by the technique of garbage collection.

1.1.2. Java Virtual Machine Stack

There are two essential structures at the run time system of Java program: Java stack and frame (see Figure 2). Each Java Virtual Machine thread has a unique private Java Virtual Machine stack, created at the same time as the thread. A Java Virtual Machine stack is mainly used to store frames. Because the Java Virtual Machine stack is never

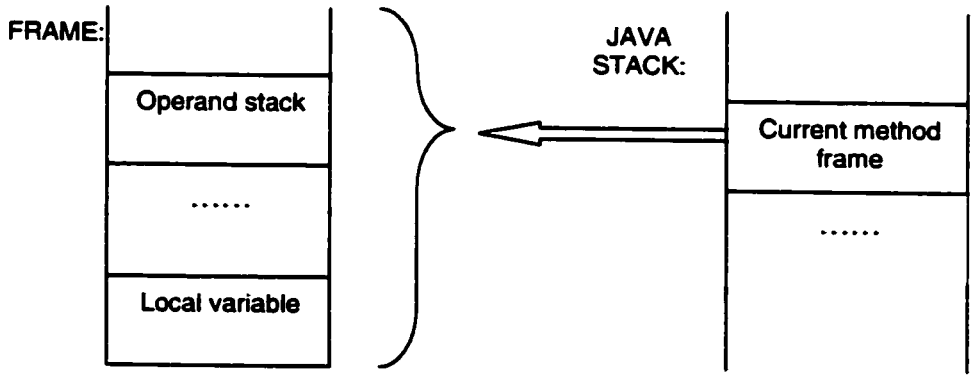


Figure 2. Stack and Frame

manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a JVM stack does not need to be contiguous. The Java Virtual Machine specification permits JVM stacks either to be of a fixed size or to dynamically resize as required by the computation. If the JVM stacks are of a fixed size, the size of each JVM stack may be chosen independently when that stack is

created. A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of Java Virtual Machine stacks, as well as, in the case of dynamically expanding or contracting Java Virtual Machine stacks, control over the maximum and minimum sizes [JT97].

1.1.3. Frame

A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked and a frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception). As described above, frames are stored in the Java Virtual Machine stack of the thread that creates the frame. Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame. Thus the size of the frame data structure depends only on the implementation of the Java Virtual Machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the current frame, and its method is known as the current method. The class in which the current method is defined is the current class. Operations on local variables and the operand stack are typically with reference to the current frame.

A frame is no longer to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. When a method returns, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded, as the previous frame becomes the current one.

A frame created by a thread is local to that thread and cannot be referenced by any other thread.

1.2 Java Bytecode

Once a Java source code is compiled, a class file is produced, which is needed by the Java runtime system. For example, following is the Java program "Hello World":

```
public class HelloWorld{
        public static void main (String args[]){
                System.out.println("Hello World!");
        }
}
```

The above java file is compiled to Java Virtual Machine code:

```
Compiled from HelloWorld.java
public class HelloWorld extends java.lang.Object {
  public HelloWorld();
  public static void main(java.lang.String[]);
}
Method HelloWorld()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return
Method void main (java.lang.String[])
  0 getstatic #2 <Field java.io.PrintStream out>
```

```
3 ldc #3 <String "Hello World!">
5 invokevirtual #4 <Method void println(java.lang.String)>
8 return
```

The bytecode instructions exist on each line of the method. The Java Virtual Machine executes these instructions. According to Sun's specification, there are about 200 instructions of bytecode. In this thesis work, a more abstract subset of instructions is used instead of applying exactly the same instructions for convenience. Though a little different, it has no effect on the formalization work.

1.3 Verification

Verification is one of activities involved in the linking process. Linking is the process of taking the bytecode of a class or an interface and combining it into the runtime state of the Java Virtual Machine [Dea97], so that it can be executed. Since verification at linking time ensures the structural correctness of bytecode, it is an essential step to verify that the bytecode is executable.

A Java Virtual Machine verifier is responsible for the verification work. It ensures that each class file satisfies the necessary constraints at linking time. This linking-time verification enhances the performance of the interpreter. Otherwise, expensive checks would have to be performed to verify constraints at run time for each interpreted instruction. By doing those checks, the Java Virtual Machine can assume that these checks have already been performed. For example, the Java Virtual Machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java Virtual Machine instructions are of valid types.

Sun's class file verifier is independent of any compiler. It should certify all code generated by Sun's compiler or other compilers, as well as code that the current compiler could not possibly generate [O' Ca99]. The verifier will certify any class file that satisfies the structural criteria and static constraints. The class file verifier is also independent of the Java programming language. Programs written in other languages can be compiled into the class file format, but will pass verification only if all the same constraints are satisfied.

The specification work also implies this independence. It is not only for Java, but also for all other languages using similar mechanisms.

The class file verifier operates by four steps [LY96]:

1: When a prospective class file is loaded by JVM, it ensures that the file has the basic format of a class file.

2: When a class file is linked, the verifier performs all additional verification that can be done without looking at the code array of the Code attribute. The checks performed by this pass ensure that final classes are not subclassed and that final methods are not overridden, check that every class (except object) has a direct superclass, ensure that the constant pool satisfies the documented static constraints and check that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

3: During linking, the verifier checks the code array of the Code attribute for each method of the class file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the operand stack is always the same size and contains the

same types of values, no local variable is accessed unless it is known to contain a value of an appropriate type, methods are invoked with the appropriate arguments, Fields are assigned only using values of appropriate types, and all opcodes have appropriate type arguments on the operand stack and in the local variable array.

4: For efficiency reasons, certain tests that could in principle be performed in Pass 3 are delayed until the first time the code for the method is actually invoked. By doing so, pass 3 of the verifier avoids loading class files unless it has to.

1.4 Bytecode Verifier

It has been mentioned that performing data-flow analysis on each method checks the code array of the code attribute for each method of the class file. The code for each method is verified independently. In anther word, the domain of a certain data-flow analysis is within the current method being verified.

To perform the check, an array first is allocated to contain the index into the code array of the start of each instruction. The verifier then go through the code a second time and parses the instructions. During this process, a data structure is used to hold useful information about each instruction in the method. First, the contents of the operand stack and the contents of the local variable array have been checked. Then a data-flow analyzer is initialized and run to ensure the type correctness within a certain method. In one word, it ensures that at any given point in the method, no matter what code path is taken to reach that point, the types and size of operand stack and local variable are always the same or compatible.

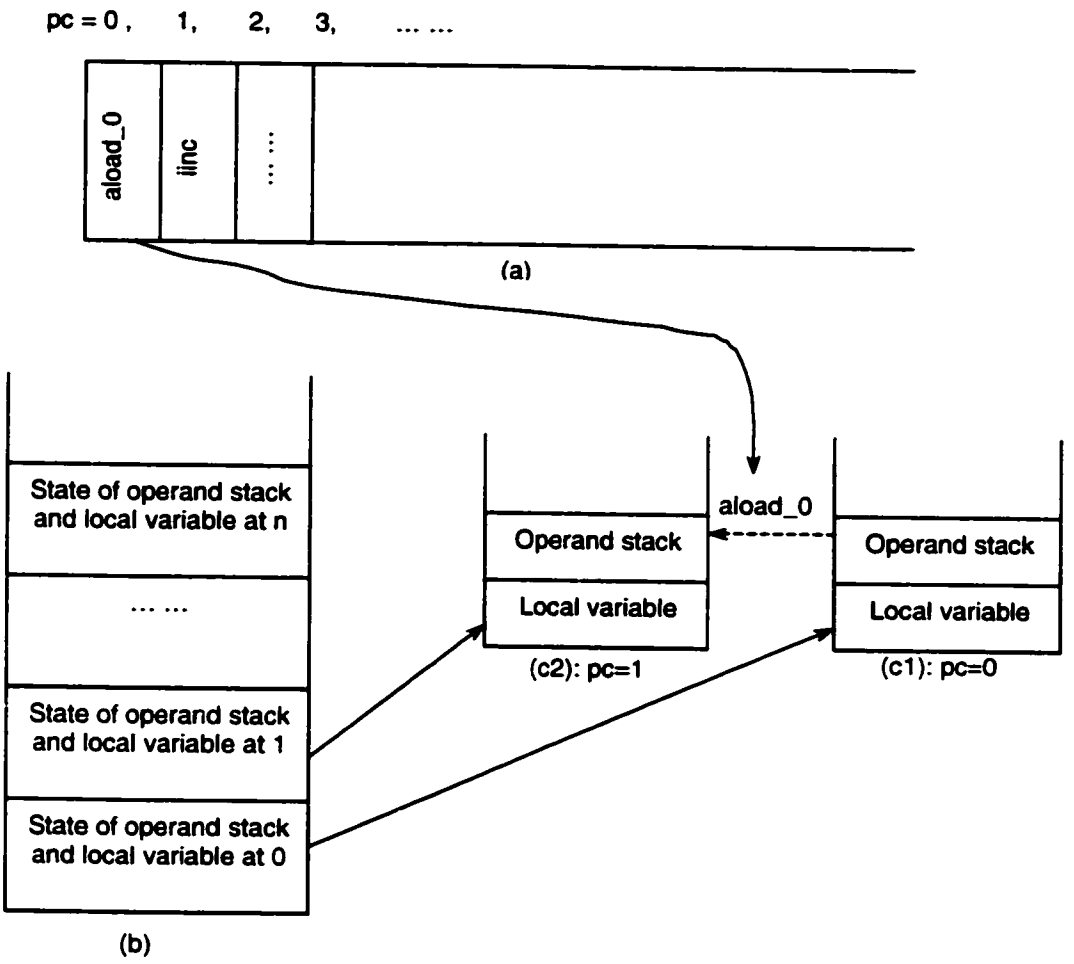


Figure 3. Structure of Data Flow Analysis

From the discussion above, it is clear that a structure to record the type information of the contents in the operand stack and local variable array is needed for each method. Figure 3 shows the three main structures needed by the data flow analysis. (a) is the structure to record program instructions. It is assumed to start at 0 and increased by one consecutively. (b) is the structure containing the type information of operand stack and local variable at every point of program. This structure has the same index of the program instructions. (c) has two parts. They belong to the same structure, the frame including the operand stack and local variable. What distinguishes them to each other is time, or in another word, different point of program. The dashed arrow in the middle represents the dynamic change from one state to another. The above example

is from point 0 to point 1. (c1) is the initial state of frame. When the program is starting, the next instruction is obtained and executed. After the operation, the state of frame varied and (c2) is the new state. The first element of (b) is corresponding to (c1), and the second is corresponding to (c2), and so on. These will be discussed further with the formalization of bytecode verifier.

2. Z Notation

Z (pronounced 'zed') is a formal specification notation based on set theory and first order predicate logic. Z is one of mature languages of formal specification. It has been developed by the Programming Research Group at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s. The first Z reference manual appeared in 1989 and developed through the 1980s in collaborative projects between Oxford University and industrial partners, including IBM and Inmos. Then, the American National Standards Institute (ANSI), the British Standards Institute (BSI) and the International Organization for Standardization (ISO) are considering writing a draft Z standard.

Z is a model-based notation. In Z, a system is usually modeled by representing its state — a collection of state variables and their values — and some operations that can change the state. This style matches well to imperative, procedural programming languages that provide a rich collection of data types, and also to some physical systems (such as digital electronics) that include storage elements. Z is also a natural fit to object-oriented programming. However, Z can also be used in a functional style. Z is just a notation. It is not a method. It is author who determines the meaning of a Z text. It can be understood to model only the behavior of a system: It is an abstract formal specification. The elements of a Z text can be understood to represent structures in code: modules, data types, procedures, functions, classes, and objects.

Z is very powerful. The notation is used in two rather different ways. First, there is a descriptive style where people use Z to model some particular systems they intend to implement, for example, a text editor. Second, there is an analytic style that experts

use to define and extend the Z notation itself. Several constructs in Z are put there to support the analytic style and are only rarely used for description.

Following is the introduction of the some definitions in Z concerned with this paper.

2.1 SCHEMA

A schema is a piece of mathematical text that specifies some aspect of software system. A schema is composed of a declaration part, and a predicate part. The declaration part declares some variables (local declaration). The predicate part expresses some requirements about the values of the variables.

A unique name is taken for each schema. It is used elsewhere in the document to refer to the mathematical text. Writing specifications often involves reusing the same notion many times. Using of notion names improves the readability of specification documents, and reduces the errors that arise when a specification is copied in other contexts.

A schema may be written in a vertical or a horizontal format. The former format is preferred if there are many declarations and predicates. Given that `Student` and `Professor` are defined sets, the schema following might be a part of a specification of school registering system.

```
COURSE
registered: P Students
instructor: Professor
attendant: P Students
---------------------
attendant ⊆ registered
```

The variables `registered`, `instructor` and `attendant` are called components of the schema `COURSE`.

If there is only one component in a schema, it is convenient to present its definition in horizontal format. For example:

```
Stack ≙[elements: seq N | #elements ≤ max_size]
```

The notation "≙" is used here for schema definition in the horizontal form. "[" and "]" are schema brackets with "|" to separate the declaration and predicate part of a schema. In fact, the schema definition is a declaration. It declares the schema name, and associates itself with a type characterized by names and types introduced in the declaration part.

2.2 TYPES

In Z, only one type is built in: integer, appropriately named `Z`, but types can be defined freely as needed. Most Z texts are largely populated with types that are invented by authors. These texts convey a lot of information choosing types. The free type definition is one of the biggest differences between Z and other traditional mathematics. In mathematics, there are not many types, and the type of each object is obvious from context. Mathematicians know that number theory deals with integers; analysis deals with real numbers, and so on. In Z, it typically works with many different types in the same text. It is important to be careful to make the type of each object explicit. In this respect, Z is more formal than traditional mathematics (more information is expressed in formulas).

There are two ways to introduce a new type. The first is free type definition. Free types are similar to enumerated types provided by many programming languages. A free type is defined when there are not too many elements in the set, and it is known

in advance what they all are. To define a free type, give its name and then, after the definition symbol ::=, list all of its elements. For example,

```
Color::= red | orange | yellow | green | black | blue
```

The second method is basic type definition. A basic type is defined when there is no need to tell in advance what all the elements are; this is used usually when sets are large. To define a basic type, just give its name, which is enclosed in brackets. For example,

```
[CHAR]
```

CHAR is character set. It is not necessary to say that characters are bytes; they might not be. It is not necessary to say what the encoding convention is; it might be ASCII, but it needn't be. It is not necessary to say which characters are in the set; they needn't include usual Roman alphabet. The omission of inessential details is one of the things that distinguish a model from an executable program. From the basic definition above, CHAR can be used in declarations, just like Z and all other predefined data types.

In addition to those two methods, an abbreviation definition can also be used to define a new type. The symbol "==" is used in abbreviation definition. For example,

```
TEXT==seq CHAR
```

"seq" here means a sequence of any type. A new type, TEXT, is introduced to define the type of a sequence of characters. Abbreviations are like the macros provided in some programming languages. They can decrease troubles of writing out long definitions again and again. They enable us to make intentions clear by giving meaningful names to constructions.

Types defined using these methods are global: they can be used anywhere in the text after their definition.

2.3 AXIOMATIC DESCRIPTIONS

In Z, an axiomatic description (also called axiomatic definition) is used to define constants of any type. There are also two parts for an axiomatic description: the declaration and predicate. For example,

```
max_size: ℕ
─────────────
max_size≤65535
```

The declaration part says that `max_size` is a natural integer, and the predicate below constrains the value of `max_size` not larger than 65535. It should have a constant value though it is not specified here.

Constants declared in axiomatic descriptions are global. This is suggested by the open appearance of the axiomatic description box.

2.4 STATE SCHEMAS

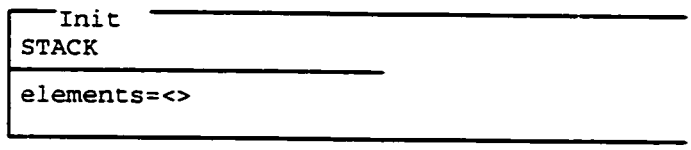
Schema adds something new when modeling computing systems: storage, or memory. The collection of contents of a system's memory is called *state*. Schemas model states as collections of state variables and their values. State variables are also called components. For example, if someone wants to model a system of stack operation, there must exist the memory to store values of stack elements.

```
STACK
elements: seq ℕ
──────────────────
#elements≤max_size
```

The schema tells that `elements` are the contents that a stack hold in memory and `elements` can be treated as a sequence of natural integer. The predicate part says that the size of the sequence must not be larger than `max_size` (defined above in axiomatic definition).

2.5 INITIALIZATION SCHEMAS

Every system has a special state from which it starts up. In Z this state is described by a schema conventionally names `Init`. For example, `STACK` is always starting with an empty sequence.



<> defines an empty sequence. The schema `Init` includes the `STACK` schema in its declaration section, which means that all the declarations and predicates in `STACK` apply to `Init` as well, so people can use local variables from `STACK`.

2.6 OPERATION SCHEMAS

System gets changed when an operation occurs. For example, the elements of a stack will be filled with natural integers starting from an empty sequence. To model this activity, Z gives the operation schema.

For a stack, there are two traditional operations: push and pop. The variables of a stack can be accessed only through the last entry. "push" is used to add an element,. "pop" is to delete an element.

```
PUSH
ΔSTACK
i?:N
─────────────────────
#elements<max_size
element'=element^i?
```

ΔSTACK tells that PUSH is an operation schema that changes the state of STACK. The symbol of question mark means that i? is an input variable of this operation. The prime mark is a decoration that represents the post state after the operation. Decorating a schema name defines a new schema that is the same as the named schema, except that all the variable names in the new schema are marked with the same decoration as the schema name itself. For example,

```
STACK
Elements': seq N
─────────────────────
#elements'≤ max_size
```

We can get the "Δ" naming convention being an abbreviation for the schema that includes both the unprimed "before" state and the primed "after" state. It is defined like:

```
ΔSTACK
STACK
STACK'
```

In addition to decoration "Δ", another symbol is used often in Z: Ξ. It means a constant state. There is no change of the state when Ξ is used. It is used only to read state variables without changing their values, or to consume inputs or produce outputs. For example, ΞSTACK can also be written as:

```
ΞSTACK
ΔSTACK
─────────────────────
element'=element
```

Back to the operation schema, the detail of how the stack changes is described at the predicate part. The first line of predicate indicates the condition that push is not allowed to happen when stack is not full. That operation adds a new element (the input) to the top of the stack.

2.7 SCHEMA CALCULUS

To make the system robust, all conditions that may happen should be considered. The operation schema PUSH defines what happens when the stack is not full. When the first condition is not met, the system will crash. To define a total version of PUSH, it can be defined in many parts, and then schema calculus is used to put all these parts together. For example, being a complement of PUSH, another schema is needed to describe the situation when the stack is full.

```
FULL
ΞSTACK
i!:seq CHAR
-----------
#elements=max_size
i!="the stack is full"
```

C_PUSH≙PUSH∨FULL

This formula means that PUSH behaves as its operation when the stack is not full, but gives a message when the stack is full.

3. Formalizing bytecode verifier

3.1 Types

We have talked about the verification of bytecode in the previous parts. It is clear that type is one of the most important thing of security. Some original types are defined at the beginning of the specificatioin:

```
[IP] [INT] [TOP] [TYPE] [BOOL]
NONE::=none
Instruction::= inc | push0 | pop | load | store |
               if | jsr | ret | new | init | use
```

IP is the set of indexes to access instructions, or in another word, it is the type for program counter. Because it is unnecessary to distinguish among the integral types (e.g., byte, short, char) for bytecode verifier, when determining the value types on the operand stack, INT is defined to represent these integral types. TOP is the type of all kinds of values permitted in Java language. NONE is the type for none, which means an invalid value. TYPE is the set of all kinds of types for variables; for example, INT is the element of TYPE. BOOL is the type of Boolean that has only two members, true and false. These types are defined for the global use throughout the specification. The definition of Instruction lists all members of bytecode instructions used in this thesis.

3.2 Main Structures

Since the class file verification is subject to the static feature like the format of the class file, it is relatively a simple task comparing with the bytecode verification. In 1.4, the data flow analyzer has been discussed in detail. There are three kinds of information should be recorded within a method. One is the bytecode array that

contains all the instructions of the method with index starting from 0. The second is the array to record the contents of the operand stack and local variable prior to execution of every instruction. The last one is to maintain the type information of the operand stack and local variable at each point of the method. There are two indexes to the structure. One is the index of the instruction and the other is the index of operand stack and local variable array.

3.2.1. Code Array

The first structure that should be defined is the program paragraph that would be verified. `P` represents the array that contains all the instructions of the method. The index to this array belongs to `IP`. Program has a pointer to point out current method, which is denoted as `pc`. `F` is an assistant array for the verifier. We have known the process of the data flow analysis. Because the verifier must know if the instruction has been met or not, `F` is used as a flag. If the instruction at a certain point has been met once, the value of `F` at the point is set to be true; otherwise, it keeps the value of false.

```
Program
P: IP   Instruction
F: IP   BOOL
pc:IP
-----------------------
pc ε  dom(P)
dom(F) = dom(P)
```

The instruction sets of the program keep unchanged all the time. Its value is assigned when it is initialized. The `pc` starts at point 0 and the initial value of all elements in `F` are all false. The initial state of `Program` is specified as following:

INITProg

Program
Instr?: IP Instructions

θ(P) = θ(Instr?)
pc=0
∀x ∈ dom(F)• F(x)= false

3.2.2. Operand Stack and Local Variable

When a method is invoked, a block of memory is allocated to store the useful data of method, the operand stack and local variable array. The operand stack and local variable array are defined to be composed of sequences of variables. The difference between them is depicted by their different way of accessing data. Operand stack applies the last-in-first-out rule. The only operand can be read or written is on the top of the operand stack. Stack has a dynamic size and there is a maximum size limit for it. Local variable array has a fixed length once the frame being initialized, and can be accessed by the index without any constraint.

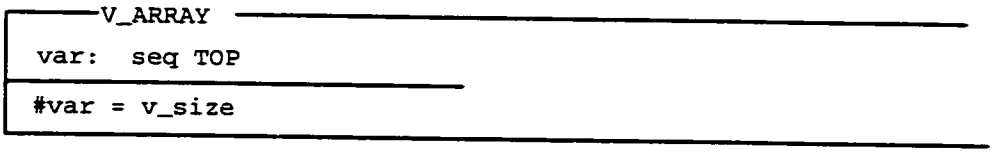
o_size: N
v_size: N

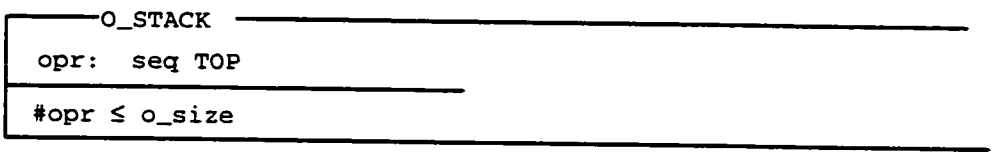
---

o_size ≤ 25
v_size = 20

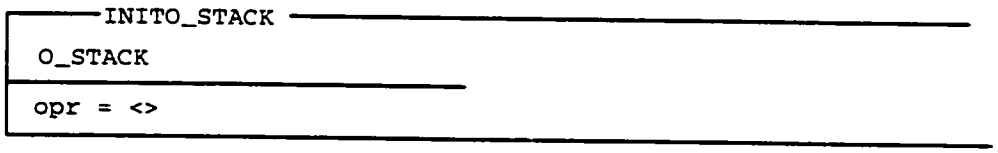
o_size and v_size are size limitations of operand stack and local variable array respectively. These two values can be decided once the method is invoked. It is not necessarily the value given here but depending on different situation.

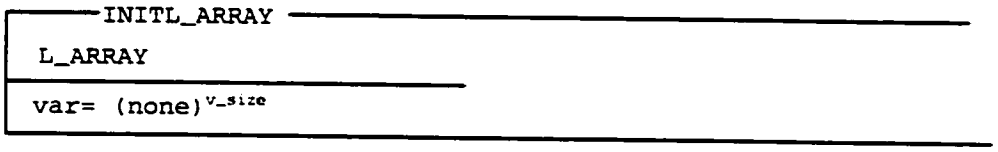
As described above, O_STACK is the schema to describe the operand stack, which is composed of a sequence opr. V_ARRAY, the local variable array, is composed of a sequence var. The predicate for each of them is the constraint on size.



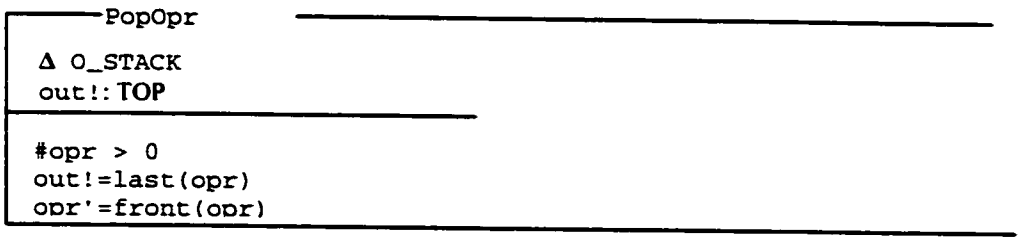


O_STACK and V_ARRAY are initialized once the method is invoked to record data information. At the beginning, the operand stack is an empty stack and local variable array contains a fixed number of invalid values, which is given when the method is called.





Though both of the operand stack and local variable array appear as a sequence, they are differed from each other by the operation on them. PushOpr and PopOpr are two operations defined to read and write the data in operand stack.



PopOpr is the operation to get the top element off from the stack. The precondition of this operation is #opr > 0, which means that to perform pop operation on an empty stack is illegal.

PushOpr

```
Δ O_STACK
in?: TOP
-----------------
#opr < o_size
opr'=opr ^ in?
```

PushOpr is the operation to put an input value onto the top of the stack. The precondition of this operation is #opr < o_size, which means that it is unacceptable to push anything to a full stack that is full.

It is clear that both of the two schemas guarantee the legal operations on the operand stack. Any attempt of overflows or underflows is avoided by defining those two operations. They are two and the only two operations of the operand stack used in the following part.

As for the local variable array, any element in it can be read or written with index. The operations GetVar and PutVar describe the actions of it.

GetVar

```
Ξ L_ARRAY
out!: TOP
x?: N
-----------------
x?∈ dom(var)
var(x)⌊ none
out!=var(x)
```

PutVar

```
Δ L_ARRAY
in?: TOP
x?: N
-----------------
x?∈ dom(var)
var(x?)=in?
```

3.2.3 Data Flow Analyzer

The other structure needed by the data flow analysis is the array to record type information of the operands and the local variables for each instruction. To describe this array, elements of them are specified as OPR_STATE and VAR_STATE:

```
OPR_STATE
O_STACK
OPR:  seq TYPE
#OPR ≤ o_size
dom(OPR)=dom(opr)
```

```
VAR_STATE
L_ARRAY
VAR:  seq TYPE
#VAR = v_size
dom(VAR)=dom(var)
```

OPR_STATE and VAR_STATE are similar to O_STACK and L_ARRAY. There is a sequence in each of them, corresponding to the operand stack and local variable array respectively. OPR is the array to contain the types of each element in opr. VAR contains the types of each element in var. OPR and opr have the same domain, as well as var and VAR. For each instruction that the analyzer passes, there is a state. The following schemas describe the structures of states at any point of program during run time.

```
VAR_TYPE
V_TYPE: IP   VAR_STATE
dom(V_TYPE) = dom(Program.P)
Ei:IP ƒ iϵdom(Program.P) ∞ V_STACK.var:V_TYPE(i)
```

```
OPR_TYPE
O_TYPE: IP   OPR_STATE
dom(O_TYPE) = dom(Program.P)
Ei:IP ƒ iϵ dom(Program.P) ∞ O_STACK.opr:O_TYPE(i)
```

OPR_TYPE and VAR_TYPE are the arrays with index to program. Their elements are the type states at each program point. Similarly, both of these structures are initialized

when the analyzer starts. The initial state of the operand stack is empty, while the state of local variable array is composed a fixed size of NONE, which is considered as the type of none.

INITOPR_TYPE

OPR_TYPE

$\forall i \in dom(O\_TYPE) \bullet O\_TYPE(i).OPR = <>$

INITVAR_TYPE

VAR_TYPE

$\forall i \in dom(V\_TYPE) \bullet V\_TYPE(i).VAR = NONE^{V\_size}$

The operations on OPR_TYPE and VAR_TYPE are SetOPR and SetVAR. They are used to set values to elements of O_TYPE and V_TYPE.

SetOPR

ΔOPR_TYPE
$i?: IP$
$in?: seq\ TYPE$

$O\_TYPE'(i?).OPR = in?$
$\forall x:IP \bullet x \in dom(Program.P) \bullet x \neq i? \Rightarrow O\_TYPE'(x)=O\_TYPE(x)$

SetVAR

ΔVAR_TYPE
$i?: IP$
$in?: seq\ TYPE$

$V\_TYPE'(i?).VAR = in?$
$\forall x:IP \bullet x \in dom(Program.P) \bullet x \neq i? \Rightarrow V\_TYPE'(x)=V\_TYPE(x)$

3.3 General operations

To simplify the formalization, a subset of JVM instructions is defined here. The basic subset of JVM instruction is the collection of instructions *inc, push0, pop, load, store, if, jsr, ret, halt, new, init, use.* Within this specification, we make an assumption that there is only one thread running at one time.

- *inc*: increases the top element on the operand stack by one, then goes to the next instruction.
- *push0*: pushes a constant value 0 onto the operands stack.
- *pop*: takes off the top element on the stack.
- *load*: puts the value of local variable x onto the top of the operand stack.
- *store*: pops off the top element on the operand stack and store it into variable x.
- *If*: pops off the top element from the operand stack, if this value equals to 0, goes to the instruction L; otherwise, goes to the next instruction.
- *jsr*: jumps to the instruction L, pushes the next instruction address on the top of the operand stack.
- *ret*: gets the return address from the variable x to return from the current subroutine, and then goes to the referred instruction.
- *new*: allocates a block of new memory and stores the reference to that memory on the top of operand stack, then goes to the next instruction. This new allocated object is uninitialized.
- *init*: pops the top element on the operand stack, initializes the object it refers to, pushes the new reference onto the operand stack.
- *use*: pops the top element on the operand stack, invoke the method of the object it refers to, after returning from the method, goes to the next instruction.

The first part of specification in this session only deals with of the situation without subroutine and object, so all instructions involved are *inc, push0, pop, load, store, if.*

```
Typeof: seq TOP   seq TYPE
-------------------------------------------
dom(seq TOP)=dom(seq TYPE)
Ai:N fi ∈dom(seq TOP)∞ TOP(i):TYPE(i)
```

```
ZERO: N
-------------------------------------------
ZERO=0
```

The major task of bytecode verifier is to go through instructions of the method, record the type information at each step and verify its soundness. The contents of the operand stack and local variable array are variable during verification. As a contrast, the type information of the operand stack and local variable are static and are easy to compare. According to the definition above, a program are composed of limited number of instructions. The verification work is to restrict possible action that can happen for each instruction by some constraints.

Following are the operation schemas for all the generated instructions defined:

```
INC
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
n:INT
-------------------------------------------
P(pc)=inc
last(opr)∈ INT
pc+1 ∈ dom(P)
pc'=pc+1
let n==PopOpr.out!+1 ∞ PushOpr[in?/n]
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

The operation INC occurs when all preconditions are matched. First, `P(pc)=inc` represents the current program instruction is "inc", because this operation schema

describe the situation when instruction "inc" is executed. According to the description of instructions, "inc" gets off the top element of the operand stack, increases its value by one and then pushes back the result to the operand stack. It is noticed that the only type that can be calculated is integer, `last(opr) ∈ INT`. Since this operation has an operand from the operand stack, it must be guaranteed that the operand stack contains at least one element. If the execution has been performed, the state of O_STACK must be changed. The predicate `let n==PopOpr.out! ∞ PushOpr[in/n+1]` gives the operation description on the operand stack and there's no action on the local variable array.

```
Type_Consist
ΔOPR_TYPE
ΔVAR_TYPE
ΞProgram
i?: IP
in_o?: seq TOP
in_v?: seq TOP
---
i?∈ dom(P)
if F(i?)=false
then {SetOPR.i?=i? ƒ SetOPR.in?=Typeof(in_o?) ƒ SetOPR
      SetVAR.i?=i? ƒ SetVAR.in?=Typeof(in_v?) ƒ SetVAR
      }
else {OPR_TYPE(i?)= Typeof(in_o?)
      VAR_TYPE(i?)= Typeof(in_v?)
      OPR_TYPE'=OPR_TYPE
      VAR_TYPE'=VAR_TYPE
      }
```

In order to keep the type consistent of over the operand stack and local variable, the type information of `opr'` and `var'` must be compared with the content recorded in V at the point pc+1. When the instruction at `pc+1` has not been executed, there is no need to compare. At that time, the only thing to do is to record the type information of `opr'` and `var'` into the corresponding location of V. Otherwise, the new state of the operand stack and the local variable must be compared with the type information that

is stored at the corresponding location in OPR_TYPE and VAR_TYPE. This is described as above, using a general operation schema Type_Consist.

PUSH0

```
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
---
P(pc)=push0
PushOpr[in?/ZERO?]
pc+1 ∈ dom(P)
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

"PUSH0" pushes constant 0 to the top of the operand stack. To avoid overflow, the current size of operand stack must not equal or be larger than the maximum size it could be. This can be guaranteed by pre PushOpr. The state of the operand stack will also have a new element on the top, which is INT, the type of integer 0. The local variable array and its state should keep invariant.

POP

```
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
---
P(pc)=push0
PopOpr
pc+1 ∈ dom(P)
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

The instruction "POP" is an operation on the operand stack. It gets rid of the top element on the stack, and so does the state of the operand stack. For this operation, there is no change on local variable array and its state. This operation is quite similar to PUSH0.

```
LOADX
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
x?:N
-----------------------------------
P[pc]= load
GetVar ∉ PushOpr[out?/in?]
pc+1 ∈ dom(P)
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

"LOADX" is the instruction that puts the x^{th} variable onto the top of the operand stack. A new element will be pushed on the operand state stack. The value is the type of the x^{th} variable in local variable, which is the x^{th} variable in the state of local variable. Neither the local variable nor its state will be changed.

```
STOREX
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΔL_ARRAY
x?:N
-----------------------------------
P[pc]=store
PopOpr ∉ SetVar
pc+1 ∈ dom(P)
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

The top element of operand stack is popped off and is put into the x^{th} variable when the instruction "STOREX" is met. The precondition of this operation is that the stack is not empty. It is also the precondition of PopOpr. The state of operand stack also pops the top element off, then puts it to the xth variable of the state of local variable.

"IFL" is a branch instruction. The element on the top of the operand stack is popped off as the branch condition. If it equals to 0 (true), the analyzer goes to instruction L, otherwise, goes to the next consecutive instruction.

```
IFL0
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
L?: IP
---
P[pc]=if
PopOpr ∫ PopOpr.out!=0
L? ∈ dom(P)
pc'=L?
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(L?)=true
```

```
IFL1
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
L?: IP
---
P[pc]=if
PopOpr ∫ PopOpr.out!⌊ 0
pc+1 ∈ dom(P)
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

Thus IFL0 and IFL1 are two and the only two possible actions "IFL" can do. For this instruction, whatever the next instruction is, the local variable and its state remain unchanged. The first element of OPR is popped off as well as opr. The instruction of "if L" is specified as: IFL | IFL0 ϖ IFL1

Now, instructions have been specified using schemas, which give some constraints on each operation to ensure the correctness of the program. From the specification above, it is easy to reason the soundness of the program if all constrains are applied.

The verification work is based on an assumption. When the instructions are passed within a method, if the state of each instruction matches all the constraints of the specification, the method is proved as a well-typed program. Given a program point that all the instructions before that point have got to a correct state, it can be proved that the next state is also correct if all the constraints of the next instruction have been applied. The assumption is:

Given OP is one operation schema among all the operations that have been described above, a tuple <O_STACK, L_ARRAY, OPR_TYPE, VAR_TYPE, Program> can be used to describe the state of analysis.

∃pc∈dom(P) • <O_STACK, L_ARRAY, OPR_TYPE, VAR_TYPE, Program> is a well-typed state ⇒

∀ OP • <O_STACK', L_ARRAY', OPR_TYPE', VAR_TYPE', Program'> is also a well-typed state

From the description of the predicates in each operation schema, for all operations defined above:

∀pc∈dom(P), i∈dom(var), j∈dom(opr) •

$\wedge$ var(i): VAR_TYPE(pc).VAR(i)

$\wedge$ opr(j): OPR_TYPE(pc).OPR(j)

$\wedge pc' \in dom(P)$

$\Rightarrow \forall i \in dom(var') \bullet$ var'(i): VAR_TYPE'(pc').VAR(i)

$\wedge \forall i \in dom(opr') \bullet$ opr'(j): OPR_TYPE'(pc').OPR(i)

3.4 Subroutine

Subroutines are subsequences of a method's larger sequence of instructions; they behave like miniature procedures within a method body [LY96]. Typically, Java's try-finally statement is compiled into a subroutine. For example,

```
int bar(int I){
        try{
            if(I==3) return 0;
        } finally{
            System.out.println("Hello World!");
        }
        return I;
}
```

This paragraph of Java program will be compiled into bytecode like:

```
    Method int bar(int)
      0 iload_1
      1 iconst_3
      2 if_icmpne 12
      5 iconst_0
      6 istore_2
      7 jsr 24
```

```
10 iload_2
11 ireturn
12 jsr 24
15 goto 36
18 astore_3
19 jsr 24
22 aload_3
23 athrow
24 astore 4
26 getstatic #2 <Field java.io.PrintStream out>
29 ldc #3 <String "Hello World!">
31 invokevirtual #4 <Method void println(java.lang.String)>
34 ret 4
36 iload_1
37 ireturn
```

The example above illustrates the use of subroutines to compile try-finally statements.

Subroutines have two challenges to the design of a type system for JVM:

(1) Polymorphism. Subroutines are polymorphic over the types of the locations they do not access.

(2) Last-in, first-out rule. When a return statement in procedure is executed, it is guaranteed that control will return to the point from which procedure was most recently called.

It has been known that there are two instructions concerned with subroutines, "jsr L" and "ret x". There is a sort of asymmetry between these two instructions, because "jsr

L" is to call the subroutine and push the address onto the operand stack but "ret x" is to return from subroutine getting its address from x^{th} variable instead of from operand stack. This kind of asymmetry is done by purpose. It is the verifier's responsibility to make sure that the variable stored in the x^{th} variable is just the return address from the current subroutine. Obviously, everything will be easy if we have a stack to track the way of subroutine, pushing the returning address onto that stack and popping off the returning address.

Here is the schema describing the subroutine call stack.

```
SUB-STACK
call: N→ IP×IP
---------------------
ran(call) ∈ dom(P) × dom(P)
#call ≤ sub_size
```

The SUB-STACK is initialized at the same time with FRAME and VERIFIER. When a method is invoked there will be a block of memory allocated to record the address information of subroutines. In SUB-STACK, call is the stack of addresses information of subroutine. The only element of call is a tuple like $<IP_0, IP_1>$. IP_0 represents the entry of the subroutine. IP_1 is the returning address of the subroutine. These two factors determine a unique subroutine. Because of the stack's last-in first-out action, the current subroutine information can be obtained by retrieving the top element of the stack call. The size limit for SUB_STACK is defined as sub_size.

The initial state of SUB-STACK contains an empty stack, which means current program point is not in any subroutine.

```
INIT_SUBSTACK
SUB-STACK
call=<>
```

Similar to the operand stack, the operations of SUB-STACK are restricted to only push and pop, which actualize the last-in first-out behavior. PushSub and PopSub are such two operations on SUB-STACK.

```
PushSub
SUB-STACK
in?:IP ξ IP
#call < sub_size
call'=call^in?
```

```
PushSub
SUB-STACK
out!:IP ξ IP
#call >0
out!=last(call)
call'=front(call)
```

The instruction "JSR-L" indicates the beginning of a subroutine. First, the address given by the instruction is checked to be within the domain of the current program, and then the next instruction address will be pushed onto the top of the stack. This return address for subroutine is the address that instruction "ret x" refers to. What is different to the traditional way is that a tuple <L, pc+1> is used here to compose the subroutine stack. Not only the return address, but also the subroutine address is pushed onto the subroutine stack. The state of operand stack of instruction L must equal to current state with additional IP on its top.

JSR-L

```
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
ΔSUB_STACK
L?,next: IP
element: IPξIP
```

```
P[pc]= jsr
pc+1∈ dom(P)
L? ∈ dom(P)
pc'=?L
¬(L? in call)
let next=pc+1 ¬ PushOpr[in?/next?]
let element=<L?, pc+1> ¬ PushSub[in?/element?]
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(L?)=true
```

RET-X

```
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΞO_STACK
ΞL_ARRAY
ΔSUB_STACK
L,next: IP
x?: N
```

```
P[pc]=ret
Let next=GetVar.out! ¬ next ∈ dom(P)
Let L=first(PopSub.out!) ¬ P[next-1]=jsr ∧ <L,next>=PopSub.out!
pc'= next
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc')=true
```

"RET-X" is the instruction to end subroutine. There is no change on either the operand stack or the local variable array. If the stack of subroutine is not empty, it pops the top element, compared its last member to the xth variable. These two must equal to each other, and the first member that is also an address, must be the address for an instruction "JSR-L". Furthermore, the address that the subroutine instruction refers must be the address saved in x^{th} variable. Though both O_TYPE and V_ARRAY

are constant this time, the state of them may also change under the condition when next instruction hasn't been executed.

Figure 4 gives an example of polymorphism over a subroutine.

pc	P[pc]
01	iconst_1
02	istore_1
03	jsr 13
04	aload_0
05	iconst_0
06	putfield_x
07	iconst_1
08	istore_0
09	iconst_0
10	istore_1
11	jsr 13
12	return
13	iload_1
14	iconst_0
15	if_icmpeq 18
16	astore_2
17	ret 2
18	pop
19	ret 2

Fig.4 example of polymorphism over subroutine

Now, let's analyze this method. The information at some certain points are shown following. The first subroutine entry is at 03. When the program goes to 15, the first time, since the instruction at 15 is a branch instruction, 16 is determined to be the next instruction by judging the condition.

var[0]	var[1]	Var[2]	opr	call
this	1	(04)	ε	<13, 04>.ε

pc=17

Up to the point 17, the first subroutine is returne. The 2nd local variable contains the address 04, which is just the same as the return address stored at the top of subroutine calling stack. The program goes back the point 04, and then comes the second subroutine, which is starting at 11. At the beginning of this second subroutine:

var[0]	var[1]	Var[2]	opr	call
1	0	(04)	(12).ε	<13, 12>.ε

pc=13

The branch instruction at 18 is passed again. At this time, the program jumps to the instruction at 18. This second subroutine ends at 19. Before it returns, the state of the program is:

var[0]	var[1]	var[2]	opr	call
1	0	(04)	ε	<13, 12>.ε

pc=19

A problem is found when program is executed to this point. Though the 2nd local variable contains a variable of return address, it is the return address for the first subroutine other than the second. It is obviously wrong because the return address stored in the top of subroutine calling stack is 12, while the actual return address obtained is 04. An exception occurs during the analysis at this point. As a result, the program is treated as an illegal one.

When a subroutine is concerned, a new structure is created to ensure the well-formed calling stack. The previous theorem in 3.1 will be changed to contain the subroutine. The assumption of the verification is if the program takes a step from a well-typed state with a well-formed subroutine call stack, then it reaches a well-typed state with a

well-formed subroutine call stack. Informally, a subroutine call stack `call` is well formed when:

- Given a return address `L` in `call`, the instruction just before `L` is a `jsr` instruction that calls the routine from which L returns;
- The current program counter pc and all return addresses in cal have the correct subroutine label associated with them;
- The return address at the bottom of the call stack is for the code that can access all local variables;
- Any variable that can be read and written by a callee can also be read and written by its caller.

From the operation schemas, we can get the theorem for all possible operations:

$$\forall pc \in P,\ i \in dom(var),\ j \in dom(opr) \bullet$$

$$\wedge\ var(i) : VAR\_TYPE(pc).VAR(i)$$

$$\wedge\ opr(j) : OPR\_TYPE(pc).OPR(j)$$

$$\wedge pc' \in dom(P)$$

$$\wedge \forall L,p \cdot \langle L,p \rangle \in (call) \wedge (P[p-1] = \langle jsr,\ L \rangle \vee \langle L,p \rangle = \varepsilon)$$

$$\Rightarrow \forall i \in dom(var') \bullet\ var'(i) : VAR\_TYPE'(pc').VAR(i)$$

$$\wedge \forall i \in dom(opr') \bullet\ opr'(j) : OPR\_TYPE'(pc').OPR(i)$$

$$\wedge \forall L,p \cdot \langle L,p \rangle \in (call') \wedge (P[p-1] = \langle jsr,\ L \rangle \vee \langle L,p \rangle = \varepsilon)$$

Proof: Assuming that all the hypotheses of the lemma are satisfied, there are three possible cases split on the instruction at `P[pc]`.

(1) The first case is instruction that do not modify the subroutine call stack: inc, if, load x, pop, push0 and store x. For these instructions, since there is no operation on call, whatever the current call stack is, the next state of the subroutine call stack is still the same. Given the current subroutine call stack is well formed, another well-formed call stack state at next step is obtained directly.

(2) The other case is the instruction jsr L for some L. In this case, the JSR-L implies that call'=call^<L,pc+1>, so if at current state, the call stack is empty, call=<>, at point L, the only element in call stack is <L,pc+1>, and P[pc]=jsr. When the current call stack is not empty, following the assumption $\forall L,p \cdot <L,p> \in (call) \wedge P[p-1] = jsr$ and the predicate of schema JSR-L we can get $\forall L,p \cdot <L,p> \in (call^{\frown}<L,pc+1>) \wedge (P[p-1]=jsr$

(3) The third case is the instruction ret x for some x. Following the constraints defined in the schema RET-X, the subroutine call stack must not be empty. Since all the elements in the stack satisfy the conditions of being well formed, the next state of call stack is still well formed.

Compared with the method that other papers have used, we can obviously see that it is easier and clearer to specify in Z.

$$\frac{\begin{array}{c} P[i]=ret\ x \\ R_{p,i}=\{L\} \\ x \in Dom(F_i) \\ F_i[x] \in (IP\ L) \\ \forall j \cdot P[j]=jsr\ L \Rightarrow (\wedge \forall y \in \\ Dom(F_i) \cdot F_{j+1}[y]=F_i[y] \wedge S_{j+1}=S_i) \end{array}}{F,\ S,\ i \vdash p}$$

$$\frac{\begin{array}{c} P[i]=\text{jsr } L \\ Dom(F_{i+1})=Dom(F_i) \\ Dom(F_L)\subseteq Dom(F_i) \\ \forall y\in Dom(F_i)\setminus Dom(F_L)\cdot F_{i+1}[y]=F_i[y] \\ \forall y\in Dom(F_L)\cdot F_L[y]=F_i[y] \\ S_L=(IP\ L)\cdot S_i \\ (IP\ L)\notin S_i \\ \forall y\in Dom(F_L)\cdot F_L[y]\neq (IP\ L) \\ i+1\in Dom(P) \\ L\in Dom(P) \end{array}}{F,\ S,\ i \vdash P}$$

These are rules for *ret x* and *jsr L*. For instructions at point i, R_i is a subroutine label that identifies the subroutine to which the instruction belongs. For some programs, more than one R may satisfy both $R_1=\{\}$ and the constraints because the labeling of unreachable code is not unique. It is convenient to assume a canonical R for each program P, when one exists. R_P is written for this canonical R, and $R_{P,i}$ is the value of R_P at the address i. The third and forth premise of the rule for ret will check if the information stored in the location referred by the instruction is the type of IP L, and this type is to ensure that it is a return address for L not for other subroutine. Only making sure that the type matches is not enough because this doesn't imply last-in, first-out behavior. To make sure that the only return address for L available inside L is the most recent return address, not one tucked away during a previous invocation, two rules are provided to constrain the instruction.

Dynamic semantics and static semantics are separately specified for this method above. What are showed above are just two of static rules in the typing system. By checking static information, the verification of type safety is done. Unlike the specification in this paper, dynamic and static rules are integrated to one schema using Z. For this reason, the structure of JVM verifier is implicit not explicit in the specification of operations. Though, it is reasonable to study checking apart from

inference, using Z specification, the structure is described clearly before any operation schema or the state of each operation shows what the operation does on those structures.

In addition, a domain of subroutine is defined like Dom (F), which denotes the domain of the local variable and operand stack that a subroutine can access. But the domain idea cannot be found in general JVM specification. In this specification, subroutine is treated as a special block of instructions inside the method, which has the same domain with the method. The instructions in subroutine may access all the contents of local variable array instead of a subset of local variable array. What should be mainly noticed is to keep the state continuous with consecutive instructions.

According to the informal specification of JVM, there doesn't exist a structure like the subroutine call stack. This method can easily eliminate the explicit existence of a call stack. Instead, it uses labels to ensure the correct relation between subroutine return addresses and instructions.

Both of these two ways make the method more complex than the way Z specification uses. The specification using Z describe the calling stack explicitly. The proof is also more difficult because a lot of assistant lemmas are needed to prove the soundness.

3.5 Object Initialization

As an object-oriented programming language, the most important features of Java are the concepts of class and object. Following are the security problems concerned with object.

The main problem is the initialization of objects before use. Though it is clear from the Java source language statement that the class constructor will be called before any

methods can be invoked through the object reference, it is not obvious from a simple scan of the resulting JVM program [FM98]. There are mainly two reasons for that. First, there may be many bytecode instructions to evaluate the parameters for the call to the constructor. The other reason is that the structure of the JVM requires copying pointers to uninitialized objects. Therefore, some kinds of analysis are needed to make sure that an object is initialized before it is used.

There are several attacks on various implementations that are published. Some of them are related to the object initialization. For example, a bug in an early version of Sun's bytecode verifier allowed applets to create certain system objects which they should not be able to create, such as class loaders [DE97]. The problem was caused by an error of how constructors were verified and resulted in the ability to potentially compromise the security of the entire system. It is important to have a formal specification of the verification. This part of analysis is also done during the data flow analysis.

Creating a new class instance is a multi-step process. The statement

```
...
new myClass(i, j, k);
...
```

can be implemented by the following:

```
...
new #1   // Allocate uninitialized space for myClass
dup      // Duplicate object on the operand stack
iload_1  // Push i
iload_2  // Push j
iload_3  // Push k
invokespecial #5 // Invoke myClass.<init>
```

. . .

This instruction paragraph leaves the newly created and initialized object on top of the operand stack. The instance initialization method for class myClass treats the new uninitialized object as its argument in local variable 0. Before the method invokes another instance initialization method of myClass or its direct super class on this, the only operation the method can perform on this is to assign values to the fields declared within myClass.

It is very difficult to recognize initialization-before-use in bytecode. It is clear that the first instruction `new #1` is only to allocate memory for the object. It is separated from the constructor invocation instruction `invokespecial #5` by four lines. The first intervening instruction `dup`, duplicates the pointer to the uninitialized object, because a pointer to the object must be passed to the constructor. As a convention of the stack-based virtual machine architecture, parameters to a function are popped off from the stack before the function returns. The other three instructions are to put the parameters onto the stack for the use of constructor.

Depending on the number and types of constructor arguments, many different instruction sequences may appear between object allocation and initialization. For example, several new objects are passed as arguments to a constructor. In this case, it is necessary to create objects and initialize them before passing them to the constructor. Generally, the code fragment between allocation and initialization may involve substantial computation, including allocation of new objects, duplication of object pointers, and jumping from other locations in the code.

A few of pointers to the object that is initialized may exist as a result when a constructor is called. So do the pointers to other uninitialized objects. In order to

verify code that uses pointers to initialized objects, it is therefore necessary to keep track of which pointer points to initialized objects, and it is necessary to keep track of which pointer is aliases (name the same object).

```
...
new #1
new #1
dup
iload_1
iload_2
iload_3
invokespecial #5
...
```

For the example above, there are two instances of the same class. When the instruction `invokespecial #5` is reached during execution, there will be two references to two different uninitialized objects. If the bytecode verifier checks object initialization statically, it must be able to determine which reference points to the object that is initialized at this line and which point to the remaining uninitialized object. Otherwise, the verifier should either prevent use of an initialized object or allow use of an uninitialized one.

The verifier initializes the local variable 0 to contain an object of the current class when doing dataflow analysis on instance initialization methods. For an instance initialization method, local variable 0 contains a special type indicating an uninitialized object. After an appropriate instance initialization method is invoked on this object, the current class type replaces all occurrences of this special type on the verifier's model of the operand stack and the local variable array. The verifier rejects the code that uses the new object before it has been initialized or that initializes the

object more than once. In addition, it ensures that every normal return of the method has invoked an instance initialization method either in the class of this method or in the direct super class [LY96]. For each line of the bytecode program, some status information is recorded for every local variable and stack location. When a location points to an object that is known to be uninitialized in all executions reaching this statement, the status will include not only the property uninitialized, but also the line number on which the uninitialized object would have been created.

A special type is created and pushed on the operand state stack as the result of the Java Virtual Machine instruction new. This special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an instance initialization method is invoked for that class instance, the intended type of the class instance replaces all occurrences of the special type. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds. This is a very simple and highly conservative form of aliasing analysis.

A valid instruction sequence must not have an uninitialized object on the operand stack or in the local variable array during a backwards branch, or in the local variable array in code protected by an exception handler or a finally clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through a loop.

A new special type should be defined first before other descriptions on the operations related to objects,

$$\tau ::= \texttt{INT} \mid \sigma \mid \sigma_i^{\cdot} \mid \texttt{TOP}$$

The type σ_i^* is just the special type for an object of type σ allocated on line i of a program, which has not been initialized. Every object type or uninitialized object type τ has a corresponding infinite set of values, $A(\tau)$, different from values of any other type.

NEW-σ

ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
$\sigma^*?: \tau$
$\alpha? : \sigma^*?$

P[pc]= new
$\alpha? \in A(\sigma_{pc}?^*)$
$\neg$ (α? in opr $\vee$ α? in var)
PushOpr[in?/α?]
$\neg$ ($?\sigma_{pc}^*$ in O_TYPE[pc+1] $\vee$ $?\sigma_{pc}^*$ in V_TYPE[pc+1])
pc+1 $\in$ IP
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true

To allocate and initialize objects, it is needed to generate object values not in use by the program. The values that appear on the operand stack or in the local variable array are the only values in use by the system. Therefore, when a new object is created, a currently unused value of an uninitialized object type is placed on the operand stack. $\neg$ ($?\alpha$ in opr $\vee$ $?\alpha$ in var) is just the predicate to ensure that the newly created is not in the memory of system, in another word, it is not being used currently. Both the object type and the line number of the instruction together determine the special type of that value. Similarly, this special type should not pre-exist in the state of the operand stack or the local variable array at that point. This is crucial to avoid

the situation in which a running program may have two different values mapping to the same statically computed uninitailized object type.

The instruction new creates a new object of a certain and put it onto the operand stack. Correspondingly, the state of the operand stack adds a new element of this new object type. It has no effect on the local variable array.

INIT-σ

ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΔL_ARRAY
α?:σ
α!:σ!

P[pc]= init
α!= PopOpr
∃j:IP • α! ∈ A(σ_j!) ∧ P[j]=new
α? ∈ A(σ)
¬ (α? in opr ∨ α? in var)
ReplaceOpr[α!/α?]
ReplaceVar[α!/α?]
pc+1 ∈ IP
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true

The constraints for the instruction init are the keys to the analysis method. An initialized type replaces all occurrences of the type in the stack. This will change the types of all references to the object that is being initialized, because all those references will be in the same static equivalence class and, therefore, have the same type. ReplaceOpr and ReplaceVar are two assistant operations on the operand stack and the local variable array.

```
ReplaceOpr
Δ O_STACK
pre?, post?: TOP
---
∀i∈dom(opr)fopr(i)=pre? ¬ opr'(i)=post?
```

```
ReplaceVar
Δ L_ARRAY
pre?, post?: TOP
---
∀i∈dom(var)fvar(i)=pre? ¬ var'(i)=post?
```

For operation INIT-σ, it is assumed that the input of the schema, α?, is just the correct initialized object for type σ.

The instruction use requires an initialized object reference on the top of the operand stack. Whatever the instruction would do TO the system, the state after this operation is always the same. At next step, the top element of the operand stack, which should be the object reference, is popped off. The local variable array remains unchanged.

```
USE-σ
ΔProgram
ΔOPR_TYPE
ΔVAR_TYPE
ΔO_STACK
ΞL_ARRAY
---
P[pc]= use
last(opr)∈ σ
PopOpr
pc+1 ∈ IP
pc'=pc+1
Type_Consist[i?/pc',in_o?/opr',in_v?/var']
Program'.P=Program.P
F'(pc+1)=true
```

Figure 5 gives an example of verification procession of object initialization.

pc	P[pc]	VAR[0]	OPR
1	new C	TOP	ε
2	new C	TOP	$C_1^{\cdot}.\varepsilon$
3	store 0	TOP	$C_2^{\cdot}.C_1^{\cdot}.\varepsilon$
4	load 0	$C_2^{\cdot}$	$C_1^{\cdot}.\varepsilon$
5	load 0	$C_2^{\cdot}$	$C_2^{\cdot}.C_1^{\cdot}.\varepsilon$
6	init C	$C_2^{\cdot}$	$C_2^{\cdot}.C_2^{\cdot}.\varepsilon$
7	use C	C	$C.C_1^{\cdot}.\varepsilon$
8	halt	C	$C_1^{\cdot}.\varepsilon$

Fig. 5 example of object initialization

There are two objects with the same class are created consecutive. One is at point 1, and the other is at 2. These two uninitialized objects have two different types, which are distinguished by the line number. When the constructor is invoked, it is easy to find out which is the exact object that needs to be initialized.

One limitation of aliasing analysis based on line numbers is that no verifiable program can ever be able to reference two objects allocated on the same line, without first initializing at least one of them. If this situation were to occur, references to two different objects belonging to the same equivalence class would exist. Unfortunately, there was an oversight in this regard in the development of the Sun verifier, which allowed such a case to exist.

It is a key point to ensure that the machine cannot access two different uninitialized objects created on the same line of code. This will require a unique correspondence between uninitialized object types and run-time values.

Therefore, the one step soundness would be that from the operation schemas, we can get the theorem, for all possible operations:

$\forall pc \in P,\ i \in dom\ (var),\ j \in dom(opr \bullet$

$\wedge$ var(i): VAR_TYPE(pc).VAR(i)

$\wedge$ opr(j): OPR_TYPE(pc).OPR(j)

$\wedge\forall L,p \cdot <L,p> \in (\ call) \wedge (P[p-1]=<jsr,\ L> \vee <L,p>=\varepsilon)$

$\wedge(\forall \sigma^{\cdot} \in T^{\cdot},\ \exists o^{\cdot}:\sigma^{\cdot} \cdot$ var(i)= $o^{\cdot} \wedge$ VAR_TYPE(pc).VAR(i)= $\sigma^{\cdot}$ $\wedge$

opr(i)= $o^{\cdot} \wedge$ OPR_TYPE(pc).OPR(i)= $\sigma^{\cdot}$)

$\wedge$ opr(j):OPR_TYPE(pc).OPR(i)

$\wedge pc' \in P$

$\forall i \in dom(var') \bullet$ var'(i): VAR_TYPE'(pc').VAR(i)

$\wedge \forall i \in dom(opr') \bullet$ opr'(j): OPR_TYPE'(pc').OPR(i)

$\wedge\forall L,p \cdot <L,p> \in (call') \wedge (P[p-1]=<jsr,\ L> \vee <L,p>=\varepsilon)$

$\wedge\forall \sigma^{\cdot} \in T^{\cdot}, \exists o^{\cdot}:\sigma^{\cdot} \cdot$ var'(i):$o^{\cdot} \wedge$V_TYPE'(pc').VAR(i)= $\sigma^{\cdot}$

$\wedge$ opr'(j): $o^{\cdot} \wedge$ O_TYPE'(i).OPR(j)= $\sigma^{\cdot}$

3.6 Main Soundness Theorem

There are mainly six structures in this specification, Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, and SUB_STACK. A tuple < Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, SUB_STACK > is used to represent a well-typed state of the program.

If a program executing in the machine model attempts to perform an operation leading to a type error, such as using an uninitialized object, it will get stuck, since those operations are not defined by this specification. By proving that well-typed programs only get stuck when a halt instruction is reached, it is known that well-typed programs will not attempt to perform any illegal operation. Thus, this theorem implies that the analysis is correct by showing that no erroneous program is accepted.

From the one-step theorem, we can get

$\forall pc \in P \bullet$

<Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, SUB_STACK > $\rightarrow$

<Program', OPR_TYPE', VAR_TYPE', O_STACK', L_ARRAY', SUB_STACK'>

"$\rightarrow$" is defined as a map from a well-typed state to another well-typed state through any operation defined above.

A well-typed program stops only when it has reached a halt instruction, not because the program counter has gone out of bounds or a precondition of an instruction does not hold. It can be proved that if for all the possible operations, a new well-typed state can be obtained through multiple operations starting at a well-typed state when all the constraints of each operation are satisfied.

$\forall pc \in dom(P) \bullet$

$(\wedge$<INIT_Program, INITOPR_TYPE, INIT_VAR_TYPE, INITO_STACK, INITL_ARRAY, INIT_SUBSTACK> $\rightarrow^{*}$

<Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, SUB_STACK>

$\wedge \neg \exists pc', OP \cdot$

< Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, SUB_STACK > $\rightarrow$

< Program', OPR_TYPE', VAR_TYPE', O_STACK', L_ARRAY', SUB_STACK'>)

$\Rightarrow$ P[pc]=halt $\wedge$ $\forall i \in dom(opr) \cdot$

opr(i):OPR_TYPE(pc).OPR(i)

It is clear that the initial state of a program must be a well-typed state. From the one-step theorem, the multi-step theorem can be drawn. Since <Program, OPR_TYPE, VAR_TYPE, O_STACK, L_ARRAY, SUB_STACK> is the final state of the program, there must not exist a situation to continue the program. The last condition stands because the returning result of a method is stored in the operand stack when the method returns. Thus a formal specification of Java Virtual Machine bytecode verifier has been built, which can be proved to verify a program's soundness.

4. CONCLUSION

Security is a very important feature of Java. Java security relies on the type-safety of Java Virtual Machine [TH99a]. The bytecode verifier is an essential part of the Java Virtual Machine. Through type checking, it helps reconcile safety with efficiency. More bugs [Dea97] are found in JVM. It is insufficient to tell if the Java is type-safe or not, because only one prose specification is applied, though it is supposed to be safe [DE97] [NV98].

This thesis first introduces the significance of formalization of the Java bytecode verifier, which is essential to solve the security problems of mobile Java code. Secondly, the fundamentals if Java Virtual Machine and Z language are introduced briefly before the formalization work. The major contents of this thesis include the formal specification of the main structure of JVM and bytecode verifier based on a subset of Java Virtual Machine instructions. From this work, it is obviously seen that Z is a good way to explore the security problems in Java Virtual Machine.

4.1 Main features

- This work employs Z as the specification language to formalize the process of bytecode verification. The main structures of JVM are constructed using Z schemas, which clarify all the components of each structure. All the operations are also associated with operation schemas. By doing so, every step of the verification became reasonable and the clear specification style of Z makes it much easier to be understood.

- Differed from other research works, this thesis combines both the static and dynamic features of Java to resolve the security problems. Most of current works

on verification extract only the static semantics to constrain the behavior of program. Though that is feasible, we believe that it is better to use both static and dynamic semantics.

- This specification uses a subset of JVM instructions that is also an essential set to other similar VM instructions. A general model is provided to security problems of mobile languages such as Java. This formalization can also be applied to other similar systems.

4.2 Other related works

Except the formal method mentioned before, there are many more other ways. Following are some of the examples.

Coglio and Goldberg unconverted subtle bugs in Sun JDK 1.2.2 that lead to type safety violations and gave fixes of all the bugs [CG00]. Dean [Dea97] and Gary McGraw [GE97] both have talked about the Java security problems in detail and pointed out some flaws of verification. All those works have discussed the origin of security problems, given some example of Java security holes and also proposed some key ideas to solve these problems. All the works above did analysis on the Java bytecode verifier but not the specification.

Based on the research work on Java security holes, one of effective ways to avoid the problems is to give a formal specification of the verification work. Tozawa presented a new formalization of the JVM [TH99a], which is a formal basis of security problems.

Coglio treated the data flow analysis problem as a constraint-solving problem on lattices and used Specware to formalize the verifier [CGQ98]. Drossopoulou

considered s type safe subset of Java and gave a proof of the soundness of the Java type system [DE97]. Another subset was defined as $Java_{light}$ to be formalized and proved type-safe by Nipkow [NV98]. Qian, comparatively, gave a larger subset of JVM specification [Qia97] and also JVM instructions specification using typing rules [Qia98].

The verification activities occur dynamically at linking time. Dean discussed the interaction between static typing and dynamic linking process [Dea97]. Based on the concept of proof linking, Fong proposed a standard architecture to craft mobile code verifiers [FC98]. Class verification is relatively a easy process. The most difficult point is class loading because of latest loading policy provided by Java. Many researchers have contribution on this. Goldberg provided a mathematical specification for Java loading and bytecode verification [Gol98]. Jensen gave a formal specification of the class loading and the visibility of members of the loaded classes [JDT98]. Those specifications didn't take subroutine and object initialization into account; instead, most of them provided a general system of verification.

Stata and Abadi [SA99] proposed a type system for subroutines, provided lengthy proofs for the soundness and clarified several key semantic issues about subroutines. Hagiya and Tozawa [HT98] presented a new type system for subroutines, where the soundness proof is extremely simple. O'Callanhan provided another simple and comprehensive type system for Java bytecode subroutines [O' Ca99]. Qian [Qia98] gave a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness. Freund and Mitchell [FM99] made a significant extension of Stata and Abadi's type system by considering object initialization.

4.3 Future Work

In the future, more work will be done on the formalization of Java verifier to get a completely safe system. More should be studied to find out the feasibility of using formal method of Z to cover all these problems and extensions, for example, the class loading problem and including heap into the specification work, etc.

BIBLIOGRAPHY

[AC94] Abadi, M. and Cardelli, L. A semantics of object types. In LICS'94 Proceedings. 1994.

[ALL96] Abadi, M., Levy, J.-J, and Lampson B. Analysis and caching of dependencies. Proc. 1996 ACM International Conference on Functional Programming, 83-91. 1996.

[Ber97] Bertelsen, P. Semantics of Java byte code. Technical report, department of Information Technology, Technical University of Denmark, March 1997

[BS99] Borger E. and Schulte W. Modular design for the Java Virtual Machine architecture. ftp://ftp.di.unipi.it/pub/papers/boerger/jvmarch.ps, 1999.

[Bra99] Bracha, G. A critique of 'Security and dynamic loading in Java: a formalization. http://java.sun.com/people/gbracha/critique-jmt.html, 1999.

[Bue99] Buechi, M. Type soundness Issues in Java, May 1999. http://www.cis.upenn.edu/bcpierce/tpes/archives and then /current/msg00140.html.

[Car97] Cardelli, L. Program fragments, linking, and modularization. In Proc. 24th ACM Symp. Principles of Programming Languages, 266-277, 1997

[CG00] Coglio, A. and Goldberg A. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In Proc. ECOOP Workshop on Formal techniques for Java Programs, 2000. Long version available at http://www.kestrel.edu/java.

[CGQ98] Coglio, A., Goldberg, A., and Qian, Z. Towards a probably-correct implementation of the JVM bytecode verifier. In Proc. OOPSLA'98 Workshoop Formal Underpinnings of Java, 1998.

[Coh96] Cohen, R.M. The Defensive Virtual Machine Specification 0.5. 1996. http://www.cli.com/software/djvm/index.html

[Coh97] Cohen, R.M. Defensive Java Virtual Machine version 0.5 alpha release. 1997.

[Dea97] Dean, D. The security of static typing with dynamic linking. In Proc. 4th ACM Conf. on Computer and Communicatuions Security. ACM Press, 1997.

[Dea99] Dean, D. Formal Aspects of Mobile Code Security, PhD thesis, Princeton University, 1999.

[DFWB97] Dean, D., Felten, D. W., Wallach, D. S., and Balfanz, D. Java security: Web browsers and beyond. In Internet Beieged: Countering Cyberspace scofflaws, D. E. Denning and P.J. Denning, Eds. ACM Press, New York. 1997.

[DEW96] Dean, D., Edward W. F., and Wallach D. S. Java security: From HotJava to Netscape and beyond. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 190-200, May 1996.

[Dro96] Drossopoulou, S. Towards an abstract model of java dynamic linking and verification. Department of Computing, Imperial College, London, UK. 1996.

[DE97] Drossopoulou, S. and Eisenbach, S. Java is type safe? probably. In Proceeding of ECOOP'97. Springer-Verlag, 389-418. 1997.

[DEK97] Drossopoulou, S., Eisenbach, S. and Khurshid, S. Is the Java type system sound? Theory Pract. Obj. Syst. 5, 1, 3-24. 1999.

[DEW99] Drossopoulou, S. Eisenbach, S. and Wragg, D. A Fragment calculus – towards a model of Seprate Compilation, Linking and Binary Compatibility. In LICS Proceedings, 1999.

[FC98] Fong, P. and Cameron, R. Proof linking: an architecture for modular verfication of dynamically-linked mobile code. In Proc. 6th ACM SIGSOFT Int. symp. on the foundations of software engineering (FSE'98), 1998.

[FM98] Freund, S. N. and Mitchel, J. C. A type system for object initialization in the Java bytecode language. In OOPSLA'98 Conference Proceedings: Object-Oriented Programming, Systems, Languages, and Applications. ACM, New York, 310-327. 1998.

[FM99] Freund, S. N. and Mitchell, J. C. A type system for object initialization in the Java bytecode language. ACM Trans. Program. Lang. Syst., Vol. 21, No.6, 1196-1250. 1999.

[GE97] Gary M. and Edward W. F. Java Security: Hostile Apptets, Holes, and Antidotes. John Wiley & Sons, Inc, 1997.

[GM99] Glew, N. and Morrisett, G. Type-safe linking and modular assembly language. In Proc. 26th ACM Symp. principles of Programming Languages, 1999.

[Gon97a] Gong, L. Going beyond the sandbox: an overview of the new security architecture in the Java development kit 1.2. In Proc.

Of USENIX Symposium on Internet Technologies and Systems, December 1997.

[Gon97b] Gong, L. Java security: Present and near future. IEEE Micro, May-June: 14-19, 1997

[Gon97c] Gong, L. New security architectural directions for Java. In Proc. Of IEEE COMPCON, pages 97-102, 1997

[Gon99] Gong, L. Inside Java 2 Platform Security: architecture, API Design, and Implementaion. Addison-Wesley, 1999.

[Gol98] Goldberg, A. A specification of java loading and bytecode verification. In 5th ACM Conference on Computer and Communications Security. ACM, New York, 49-58. 1998.

[GJS96] Gosling, J., Joy, B., and Steele, G. The JavaTM Language specification. Addison-Wesley. 1996.

[HT98] Hagiya, M. and Tozawa, A. On a new method for data flow analysis of Java Virtual Machine subroutines. In Static Analysis, 5th International Symposium, SAS'9. Springer-Verlag, 17-32. 1998.

[JDT98a] Jensen, T., Daniel L. M., and Thorn, T. A Formalization of Visibility and Dynamic Loading in Java. In IEEE ICCL, 1998.

[JDT98b] Jensen, T., Daniel L. M., and Thorn, T. Security and dynamic class loading in Java: a formalization. In Proc. IEEE Int. conference on Computer Languages, 1998.

[JT97] Jon, M. and Troy, D. Java Virtual Machine. 1997. O'REILLY

[Jon98] Jones, M. The functions of Java bytecode. In Proceedings of the workshop on the formal underpinnings of the Java Paradigm. 1998.

[KMS00] Krary, K., Michael, H., and Stephanie, W. Safe and Flexible dynamic Linking of Native Code, May 2000. Internal Report, University of Pennsylvania.

[LZV99] League, C., Zhong, S., and Valery, T. Representing Java Classes in a Typed Intermediate language. In ICFP Proceedings, September 1999.

[LB97] Liang, S. and Bracha, G. Dynamic class loading in the Java™ virtual machine. In Proc. Conf. on Object-Oriented programming, systems, Languages, and applications, 36-44. ACM Press, 1998. April 1997, pp. 607-621

[LY96] Linholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison-Wesley. 1996.

[MF97] Mcgraw, G. and Felten, E. Java Security: hostel applets, holes and antidotes. J. Wiley& Sons, 1997

[MTC+96] Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R., and Lee, P. The TIL/ML compiler: Performance and safety through types. In Workshop On Compiler Support for Systems Software. 1996.

[NV98] Nipkow, T. and Von, O. D. Javalight is type-safe?definitely. In Proceedings of the 25th ACM Symposium on Principles of Programming Languages. ACM, New York, 161-170. 1998.

[O'Ca99] O'Callanhan, R. 1999. A simple, comprehensive type system for Java bytecode subroutines. In Proceedings of the 26th ACM symposium on Principles of Programming Languages. ACM, New York.

[OW97] Odersky, M. and Wadler, P. Pizza into Java: Translating theory into practice. In POPL'97 Proceedings, January 1997.

[PV98] Posegga, J. and Vogt, H. Bytecode verification for Java smart cards based on model checking. In Proceedings of the 5th European symposium on Research in Computer Security(ESORICS). Lecture Notes in Computer science. Springer-Verlag, Berlin. 1998.

[Pus99] Pusch, C. Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL, http://in.tum.de/~pusch, 1999.

[Qia97] Qian, Z. A Formal Specification of Java Virtual Machine Instruction, 1997. http://www.infomatik.uni-bremen.de/~qian/abs-fsjvm.html

[Qia98] Qian, Z. 1998. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In Formal Syntax and Semantics of JavaTM, J. Alves-Foss, Ed. Springer-Verlag.

[Qia00a] Qian, Z., Goldberg, A., and Coglio A. A formal specification of JavaTM class loading. Long version. http://www.kestrel.edu/java, 2000

[Qia00b] Qian, Z. Standard fixpoint iteration for Java bytecode verification. ACM TOPLAS. July, 2000.

[SaV97] Saraswat, V. Java is not type-safe. Technical report, AT&T Research, 1997. http://www.research.att.com/~vj/bug.html

[SaB97] Saraswart, B. The Java bytecode verification problem. 1997. Web page at www.research.att.com/~vj/bcv.html.

[SMB97] Sirer, E. G., Mcdirmid, S., and Bershad, B. Kimera: A Java system security architecture. 1997. Web pages at kimera.cs.Washington.edu/.

[SA99] Stata, R. and Abadi, M. A type system for Java bytecode subroutines. ACM Trans. Program. Lang. Syst. 21, 1(Jan.), 90-137. 1999.

[SM99] Stephen N. F. and Mitchell, J. C. A Formal Framework for the Java Bytecode Language and Verifier. In OOPSLA Proceedings, November 1999.

[Sym97] Syme, D. 1997. Proving Java type soundness. Tech. Rep. 427, University of Cambridge Computer Laboratory. June.

[Kii97] The Kiiera Team. Security Flaws in Java Implementations, 1997. Available at http://kimera.cs.washington.edu/flaus/.

[Tho97] Thorn, T. Programming Languages for Mobile Code. ACM Computing Surveys, September 1997

[TH99a] Tozawa, A. and Hagiya, M. Careful analysis of type spoofing. In JIT'99 Java -Informations-Tage 1999, Clemens H. Cap, Hrsg., Informatik aktuell, 290-296. Springer Verlag,1999.

[TH99b] Tozawa, A. and Hagiya, M. New formalization of the JVM. http://nicosis.is.s.u-tokyo.ac.jp/members/miles/papers/cl-99.ps, 1999.

[Vol97] Volpano, D. A Type-Based approach to Program Security. Int'l Joint Conference on the Theory and Practice of software Development, LNCS 1214, Lille framce, 1997.

[Yel99] Yelland, P. A compositional account of the Java Virtual Machine. In Proceedings of the 26th ACM Symposium on Principles of Programming Languages. ACM, New York. 1999.

[Spi96] Spivey, J. M. The Z Notation: A Reference Manual. Second edition. 1996.

VITA AUCTORIS

NAME: Rui Guo

Place of Birth: Shenyang, Liaoning, China

Year of Birth: 1977

EDUCATION: Shenyan 2nd high school, Shenyang, China
1992-1995

Northeastern University, Shenyang, China
1995-1999 B.Sc.

University of Windsor, Windsor, Ontario, Canada
1999-2001 M.Sc.