# TinyVM: A Fault-Tolerant Virtual Machine

June 4, 2014

**D101F14**

Emil Čustić

Martin Sørensen

Rune Hansen

Aalborg University
Department of Computer Science
Software Engineer and Master of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

**Title:**
TinyVM: A Fault-Tolerant Virtual Machine

**Project Period:**
Spring Semester 2014, Master's Thesis

**Project Group:**
D101F14

**Authors:**
Emil Čustić
Martin Sørensen
Rune Hansen

**Supervisors:**
René Rydhof Hansen and Mads Christian Olesen

**Page Count:**
81

**Finished:**
June 4, 2014

# Resume

This report documents the development of a fault-tolerant version of the programming language TinyBytecode, as well as a virtual machine which can execute it.

A fault-tolerance scheme is proposed for TinyBytecode which utilizes redundant components for fault-detection and a checkpoint system for fault-recovery. Three components are initially duplicated on each frame on the call stack to ensure fault-detection: the operand stack, the local memory and the program counter.

At certain critical *checked* instructions the three components are compared to their redundant counterparts and if a fault is found the virtual machine rolls back to the most recent checkpoint.

Each frame contains exactly one checkpoint which is updated every time one of the checked instructions passes the comparison with the redundant components. This ensures the checkpoint is only updated with valid data and that no fault has occurred up to the point when the checkpoint is updated.

Since changes to the heap cannot easily be undone in case of a rollback, a fourth component is added to the frames on the call stack: the local heap. The local heap and its redundant counterpart are written to instead of directly writing to the heap. At every successful checked instruction the local heap is committed to the actual heap and both local heaps are cleared. This ensures that no faulty data is written to the heap.

An operational semantics for TinyBytecode is formalized. The formalization includes every instruction in TinyBytecode, exceptions, faults and fault-recovery.

Execution of native code is supported by TinyBytecode, however, certain constraints are placed on native methods. Each call to a native method must return exactly one integer value and is not allowed to modify the memory of the virtual machine.

The virtual machine is implemented using Java. The implementation of the virtual machine enables testing of the memory and performance impact of the fault-tolerance scheme.

The fault-recovery tests are performed using an external tool to inject bitflips at various points during execution. Tests are performed on all components of a frame in the call stack except the checkpoint which is assumed to be ECC protected. Two main points of potential failure are tested for each component during the tests: bitflips which occur after an instruction has completed, and bitflips which occur right after a checked instructions passes its comparison test. A set of test runs has also been performed where faults are injected at random points. A test of 5000 runs is performed on each component for each point of failure and a test of 10000 runs was performed with random fault-injection points, totalling 90000 runs.

The results of the tests show that in the the case of injecting faults after instruction execution, the virtual machine is able to recover 76.6% of all faults, with an additional 10.79% being masked. The silent data corruption rate is 0.24%. Injecting faults after the comparison of components in a checked instruction yields a predictably lower percentage of recovery of 28.25% and additionally 40.79% of faults being masked. The silent data corruption rate is significantly higher as well at 12.50%.

The results of the test show that corruptions in the program counter are the most difficult to recover from and result in the lowest number of recoveries. Due to this, an extra step is suggested which checks the program counter after every instruction. Using this extra precaution the silent data corruption rate of program counter faults is reduced to 0%. Furthermore, the results show that corruption of the redundant components is masked or recovered from in the majority of cases.

Performance tests were also performed by comparing the running time and memory usage to a version of the virtual machine with the fault-tolerance components removed. The results of the tests show that the implementation of the fault-tolerance components causes the virtual machine to be 50.5% slower and use 32.7% more memory.

# CONTENTS

CHAPTER 1

# INTRODUCTION

In this project we will create a fault-tolerant virtual machine, called Tiny Virtual Machine(TVM), which executes TinyBytecode. An operational semantics was initially created in our previous project [16], which models all of the instructions of TinyBytecode. In this project the semantics for TinyBytecode is expanded to include rules for exceptions, single-event upsets, as well as error recovery. We will develop a fault-tolerance scheme, which is to be an integrated part of the language. To enable error recovery, every other rule is reworked and expanded to reflect the developed fault-tolerance scheme. The faults, which this scheme is supposed to handle, will be single-event upsets in the operand stack, program counter, or the local memory in the form of a bitflip. While part of the goal of this project is to implement a working virtual machine, it is not meant to be fit for use in production. Instead, the purpose of the TVM is that of a tool to be used to test the fault-tolerance scheme integrated into the language's semantics, as well as the performance loss caused by the fault-tolerance scheme. Our contribution in this project is to develop a fault-tolerance scheme for a single running virtual machine, as compared to other JVM's utilizing a primary-backup architecture to achieve fault-tolerance. Another contribution will be to include native methods for the TVM and examine how these can be handled in a fault-tolerant manner.

Throughout this report "bitflip" will be used to denote a fault causing a single bit in a single value to be flipped.

Chapter 2

# The Tiny Virtual Machine Structure

In this chapter the structure of the Tiny Virtual Machine will be presented followed by a description of the fault-tolerance scheme. Different aspects of this scheme will be discussed with respect to memory usage, performance and optimizations.

## 2.1 Fault Model

Since the focus of this report is fault-tolerance, defining what a fault means is necessary. We do not know the exact specifications of the platform that our system will run on, so we will have to make some assumptions about it. First, we assume that storage is non-volatile, and thus immune to single-event upsets. Second, we assume that main memory is protected using ECC, so that any single-event upsets will be automatically corrected. Thus, in our fault model single event upsets can only occur outside of main memory and storage. Furthermore, we restrict the amount of faults that can happen to only 1 SEU during execution.

In the following sections we will explain this fault model in the context of our TVM.

## 2.2 The Structure

The structure of the TinyVM is very similar to the structure of the JVM, which was used as inspiration [7]. The most significant difference is the duplication of the call stack, or frame stack. There are two ways of performing this modification: extend the frame with duplicate elements, or duplicate the entire frame stack. As these two approaches are equivalent for our purposes, we made the choice of duplicating the contents of individual frames, as can be seen in Table 2.1.

| **Frame** |
|---|
| 2 stacks |
| 2 local memories |
| 2 program counters |

Table 2.1: A frame containing two copies of the stack, the local memory, and the program counter.

In contrast to the duplication of frames, only a single heap is used. Because of this, special measures are needed to protect the heap in the presence of single-event upsets. This will be discussed in Section 2.3.

## 2.3 The Fault-Tolerance Scheme

In this section the fault-detection and recovery scheme will be explained in detail, while the semantic formalization of these will be presented in Chapter 3.

As explained earlier, each frame contains duplicate elements for the operand stack, local memory and program counter. When an SEU occurs, one value in one of these elements will be randomly changed. To screen for such faults a special check is performed on the two copies of each element. If this check uncovers a fault the frame will have to be restored to a valid configuration. The specifics of what this check should actually do is discussed in Section 2.4.

### 2.3.1 Checkpoints

Checkpoints are what makes the TinyVM able to restore a frame to a valid configuration when a fault is discovered. To do this, a checkpoint should contain a copy of the contents of a valid frame. As checkpoints are tied to frames each frame in the call stack has an associated checkpoint. When a new frame is created a checkpoint should be created containing the frame's initial configuration. For this chapter, we will assume that checkpoints are immune to SEU's - in practice this could for example be achieved by encoding the checkpoint with ECC. The checkpoint structure is shown in Table 2.2.

| **Checkpoint** |
|---|
| operand stack(encoded) |
| local memory(encoded) |
| program counter(encoded) |

Table 2.2: The checkpoint structure.

### 2.3.2 Heap

One problem with simply rolling back to a frame's initial configuration when a fault is detected is the heap. The heap is located in main memory, and as noted in Section 2.1 we assume that main memory is immune to SEUs. However, if a fault occurs in a frame it might propagate to the heap. To avoid this a check would have to be performed every time a field needed to be written to the heap. Another problem is that valid heap modifications might have been performed before a fault is discovered, and the frame is rolled back to its initial configuration. This would make the state of the heap inconsistent with that of the frame, and could also cause memory leaks.

To solve both of these problems we introduce another element to the frame, called the local heap. Whenever a field is supposed to be written to an object on the heap, it is instead written to the local heap. When a field is read from the heap it is first checked if the local heap has the relevant data, in which case that is used instead. Thus, whenever a check finds no faults it can commit the local heap to the actual heap, and update the checkpoint to the current configuration. If a fault is found, the checkpoint can be restored, and the local heap simply discarded. Since the local heap is in the frame, and not encoded like the checkpoint, it is vulnerable to SEUs. Therefore a redundant copy of the local heap need to be maintained as well, and these two copies should also be compared whenever a check occurs.

It should be noted that this approach is similar to transactional memory. Transactional memory allows committing multiple memory read/writes as one atomic action, and is often used in concurrent programs. TinyVM is not concurrent, but we do want to ensure that memory writes are only performed in the absence of faults. A more detailed discussion of transactional memory is included in Section 6.3.

The new frame format with two redundant local heaps and a checkpoint can be seen in Table 2.3.

| **Modified Frame** |
|---|
| 2 stacks |
| 2 local memories |
| 2 program counters |
| 2 local heaps |
| Checkpoint |

Table 2.3: A modified frame for recovery purposes.

### 2.3.3 Checked Instructions

Since checks now need to commit the local heap to the actual heap and update the checkpoint, checks could have quite a large impact on performance. To help alleviate this, checks should be performed as rarely as possible, while still protecting the heap from faults. We have chosen to do this by only performing checks when certain instructions are executed. These so-called checked instructions are shown in Table 2.4.

| **Checked instructions** |
|---|
| invoke |
| return |
| new |
| exception |

Table 2.4: The checked instructions for the fault-detection scheme.

*invoke* calls a new method, and as this method can modify the heap itself, invoke needs to be checked. It is also important to check that the correct method is called on the correct object, to ensure that the new frame is correct. *return* causes the current frame, including the local heap, to be destroyed. Therefore, return needs to be checked to ensure that the local heap is committed when a method returns, as well as to ensure that the correct value is returned. *new* would not need to be checked if the local heap could handle entire objects. However, to keep the local heap as light-weight as possible it is limited to containing object fields, but not actual objects. Therefore, new creates a new object on the actual heap, and since this cannot be undone, new needs to be checked.

Finally, *exceptions* cause the operand stack in the current frame to be destroyed, and might cause additional frames to be destroyed as well. Therefore, to ensure that these actions are not caused by a fault, and to keep the two frame stacks in sync, exceptions are also checked.

## 2.4 Example

In designing the structure of the TinyVM we were unsure of how inclusive the comparisons in checked instructions needed to be. Initially, we had planned only to check the parameters for the checked instructions, as it seemed like that would always catch any faults. As it turns out this approach does have a weakness, which will be illustrated in this section using an example program.

Listing 2.1 on the facing page shows a fragment of a TinyBytecode program, written in pseudo code for readability. The code first loads an int, and then a reference, which it then invokes a method with. Then it takes the modulo of the result of that method, and the int that was loaded, and returns that value. Table 2.5 on the next page shows the contents of the local memory when this code is run.

```
load int lv[1]
load ref lv[0]
invokevirtual Example.getX() int
comp mod
return int
```

Listing 2.1: TinyBytecode snippet.

| | |
|---|---|
| lv[0] | 64 |
| lv[1] | 5 |

Table 2.5: The values in the local memory.

Table 2.6 shows how both of the operand stacks will change during execution of the example code. First the integer 5 is loaded from local memory. Then a reference to address 64 is loaded from local memory. Since invoke is a checked instruction the two copies of the parameters are compared here, and, since no fault has occurred, the checkpoint is updated. After the check succeeds the method `getX` is invoked using the reference, which returns an integer 8. Then the modulo of these two values is taken, which leaves the stack with the integer 3. Return is also a checked instruction, so another check is performed here, after which 3 is returned.

| | | | |
|---|---|---|---|
| | 64 | 8 | |
| 5 | 5 | 5 | 3 |

Table 2.6: The state of the stack after each instruction.

Table 2.7 shows how the operand stack might change in the presence of a fault. The other instance of the operand stack would still look like the one in Table 2.6. In this run the integer 5 at the bottom of the stack experiences an SEU when the reference is loaded. However, when the check is performed for the invoke, only the parameter, the reference, is checked. Since the fault did not occur in the reference the check succeeds, and the checkpoint is updated to this faulty configuration. In the presence of the fault the modulo yields the value 8 instead of the expected 3. Once the return is reached, the parameters are checked again, but this time it fails, as the faulty value is now the parameter. Since the check fails the checkpoint is restored, however, since the checkpoint contains a faulty value this method will now produce an incorrect value.

| | | | |
|---|---|---|---|
| | 64 | 8 | |
| 5 | **13** | **13** | **8** |

Table 2.7: The state of the stack after each instruction in the presence of a fault.

Because of this type of scenario it is necessary to check the entirety of the two operand stacks for equality, to avoid saving faulty values in the checkpoint. It might also be possible to use a modular checksum for the elements on the operand stack. This modular checksum could then be updated as the operand stack was modified, and comparing these might be cheaper than comparing every element on the operand stacks. However, as performance is not the focus of this project, we leave this for future work.

## 2.5 Limitations

In this section limitations of the structure of the TinyVM will be described. Ways in which the structure is vulnerable to SEUs, as well as ways these vulnerabilities might be remedied will be discussed. Finally, the performance of the system as well as ways to increase the performance will be touched upon.

### 2.5.1 Vulnerabilities

We have attempted to design the structure of the Tiny Virtual Machine to be able to tolerate as many types of faults as possible. However, since SEUs can occur at any time there are still ways for them to corrupt the state of the TVM. For example, a fault could occur just after the check associated with an invoke succeeds. This would cause the invoke to be performed as normal, even though a fault is present. Depending on where the fault occurred, this could for example cause the method to be called with a corrupted reference, or parameters.

### 2.5.2 Additional Measures

Since our structure is not completely protected against SEUs additional fault-tolerance measures would probably be used in a real-world implementation of the system. One such measure which is often used in embedded systems is a watchdog. A watchdog will restart the system if it has not received a signal from it for a certain period of time. During normal operation the system will regularly signal the watchdog to prevent the system restart from triggering. Should a fault cause the system to enter an infinite loop, or simply hang, the watchdog should eventually clear the fault using the restart.

Faults can also cause the system to enter many different exceptional states:

- Referencing a heap object at an unused or invalid location.
- Referencing a method or field that does not exist on the target object.
- Attempting a division by zero.
- Attempting to access an instruction outside of the current method.

As noted in Section 2.3.3 exceptions should be checked instructions, so before handling of an exception is attempted, the TinyVM should ensure that an SEU has not occurred.

### 2.5.3 Performance

SEUs are expected to be quite rare in the real world, so ideally the performance overhead of a fault-tolerance scheme would be minimal in the absence of such faults. However, in our model, invokes carry quite a heavy performance overhead because of checks and checkpoint creation. Additionally a significant amount of space is required, as each frame has its contents duplicated, in addition to the inclusion of checkpoints, which could take up as much space as an entire frame.

While performance is not the aim of this project we have thought of some changes to the model which could increase its performance.

**Limiting checkpoints**

Currently every frame has a checkpoint, and every invoke and return is checked. However, it might be possible to modify the system, so that checks and checkpoint creation were only performed every few frames. To do this the local heap would have to be sent from frame to frame by invokes and returns whenever it was not committed. How often these checks should be done could then be configured by the programmer. This approach would greatly reduce the performance overhead tied to invokes and returns, since checks and checkpoint creation would not always be performed. Additionally the amount of space required for frames would be decreased, as they would not all contain a checkpoint. The drawback to this approach would be that the delay until a fault is discovered would increase, as would the amount of computation to be redone.

**Programmer involvement**

It might also be possible to increase performance with the assistance of programmers. Programmers could apply different tags to methods, which the TinyVM could then use to save checks or take other shortcuts. A method might be denoted as very fast, which would cause the TinyVM to skip checking it and creating a new checkpoint, similar to what was described above. It might also be possible to denote a method as inline, which would cause the compiler to simply include its code in calling locations, instead of actually invoking it.

A set of guidelines could be created for programmers, which would describe how to write their programs to achieve the best results. For example, programmers could be asked to limit method invocations, as well as include the different tags where relevant.

CHAPTER 3

# TINYBYTECODE

This chapter is based on a chapter of the same name from a previous report by Čustić *et al*. [16]. Parts of this chapter is taken directly from this report, but this is mainly the parts about the domains which are equivalent. Some of the domains have been extended and while others are completely new. All the semantic rules have been rewritten, partially or completely, to reflect the fault-tolerance scheme introduced in Section 2.3. In Section 3.1 the language domain is introduced, following the notation of Hansen [3], Section 3.2 introduces the instructions that form the TinyBytecode syntax, and in Section 3.3 the semantics of TinyBytecode are formalized.

## 3.1 Structure

In this section a formal description of the notation used and the program structure will be presented. Following the notation of Hansen [3], programs are represented as abstract data structures which are accessed by special access functions. This provides a convenient way to extract relevant information as shown in Section 3.1.1.

### 3.1.1 Notation

Hansen defines a domain to be a set. With this set corresponding access functions are given to access and modify different elements of the set. For convenience the record notation is used to specify a domain with all its access functions:

$$\text{Dom} = (f_1 : \text{Dom}_1) \times \ldots \times (f_n : \text{Dom}_n)$$

The above equation defines the domain Dom with access functions $f_i : \text{Dom} \rightarrow \text{Dom}_i$ for $1 \leq i \leq n$. The notation used to access an element $f_i$ of $d \in \text{Dom}$ is written $d.f_i$ and changing the value of an element is written $d[f_i \mapsto v]$.

### 3.1.2 Types

Like Java bytecode, TinyBytecode is a strongly typed language. In the version of TinyBytecode presented here the primitive types are treated as is, the semantics have no special support for varying lengths or bounds. The complete type-system of TinyBytecode is shown in Figure 3.1.

$$\begin{array}{lcl} \text{Type} & ::= & \text{ClassName} \mid \text{PrimType} \\ \text{PrimType} & ::= & \texttt{boolean} \mid \texttt{int} \mid \texttt{byte} \\ \text{ReturnType} & ::= & \text{Type} \mid \texttt{void} \end{array}$$

Figure 3.1: The type system TinyBytecode.

### 3.1.3 Program

In the JVM specification [7] packages are used to distinguish class implementation and support portability. For TinyBytecode the idea of packages was left out because the complexity of the program structure was to be as low as possible. Instead the program only contains a set of classes. Furthermore, the *exceptionRef* component has been introduced for the purpose of runtime exceptions. This component contains references to pre-allocated runtime exception objects, which can be accessed by indices. Thus, when a runtime exception occurs these objects can simply be used, instead of creating a new object. The notation $\mathbb{N}_0$ is used to denote positive integers including zero.

$$\begin{array}{lcl} \text{Program} & = & (\textit{classes} : \mathcal{P}(\text{Class})) \times \\ & & (\textit{exceptionRef} : \mathbb{N}_0 \rightarrow \text{Location}) \end{array}$$

For TinyBytecode a subset of runtime exceptions is included. The exceptions can be seen in Table 3.1 on the next page with their corresponding index for *exceptionRef*.

These runtime exceptions are introduced for the completeness of the language, and later only a few examples will be given for clarification as these are not in the main focus of this report.

| | |
|---|---|
| 0 | OutOfMemoryException |
| 1 | NullReferenceException |
| 2 | DivisionByZeroException |

Table 3.1: The runtime exception with their respective indices.

### 3.1.4 Class

Classes in Java are the main components constituting a program. Classes are instantiated as objects and provide ways of accessing the encapsulated information. A class is defined by a class name, and the methods and fields defined by the class are accessed by their respective components. Furthermore, a class keeps reference to the class hierarchy by a *super* component. The class hierarchy is encoded by a function $\mathit{super} : \text{Class} \rightarrow \text{Class}_\perp$ which returns the superclass of a given class. In Java a special class exists, namely `Object`, which is implicitly the superclass of all classes except itself. Therefore, by convention, using the *super* function defined before on the `Object` class will return the bottom value $\perp$. A class also contains a constant pool. This constant pool holds statically known information about methods to be called, fields accessed and class dependencies.

$$\begin{aligned}\text{Class} \quad = \quad & (\mathit{name} : \text{ClassName}) \times \\ & (\mathit{methods} : \mathcal{P}(\text{Method})) \times \\ & (\mathit{fields} : \mathcal{P}(\text{Field})) \times \\ & (\mathit{super} : \text{Class}_\perp) \times \\ & (\mathit{constantPool} : \text{ConstantPool})\end{aligned}$$

A class inherits fields and methods of its superclass. In the Java specification [7] methods and fields can be static as well. For TinyBytecode this option has been removed for simplicity as well as abstract classes and interfaces. A class can provide its own implementation of an inherited method, and when such a virtual method is invoked, a lookup is performed at runtime to determine the class in the class hierarchy from which the implementation is to be found. This is also called *dynamic method dispatch* [7].

### 3.1.5 Field

A field is identified by the class where it was defined, the name of the field and the type.

$$\begin{aligned}\text{Field} \;=\; & (\mathit{class} : \text{Class}) \times \\ & (\mathit{name} : \text{FieldName}) \times \\ & (\mathit{type} : \text{Type})\end{aligned}$$

### 3.1.6 Method

Methods are identified by a class, a method name and its argument types. In Tiny-Bytecode the methods provide the functionality of a program by containing instructions needed to execute the method. These instructions are accessible through the function $\mathit{instruction} : \text{PC} \to \text{Instruction}_\perp$ which takes a program counter as parameter and returns the instruction at that program counter or the bottom value $\perp$, if no instruction is present. As the last component, a method contains a list of handlers for exceptions.

$$\begin{aligned}\text{Method} \;=\; & (\mathit{class} : \text{Class}) \times \\ & (\mathit{name} : \text{MethodName}) \times \\ & (\mathit{argType} : \text{Type}^*) \times \\ & (\mathit{retType} : \text{ReturnType}) \times \\ & (\mathit{instruction} : \text{PC} \to \text{Instruction}_\perp) \times \\ & (\mathit{handlers} : \mathbb{N}_0 \to \text{ExceptionHandler}_\perp)\end{aligned}$$

The first instruction of a method is defined to be at program counter 0. To advance a program counter $pc$, the program counter is incremented as $pc + 1$.

### 3.1.7 Exception Handler

An exception handler is identified by the supported interval of the program counter where the exception can be handled and the type, or class, of the exception. Furthermore, an exception handler contains a target where the program counter is supposed to jump to, if the exception can be handled.

$$\begin{aligned}\text{ExceptionHandler} \;=\;& (\mathit{from} : \text{PC}) \times \\ & (\mathit{to} : \text{PC}) \times \\ & (\mathit{target} : \text{PC}) \times \\ & (\mathit{type} : \text{Class})\end{aligned}$$

### 3.1.8 Constant Pool

A constant pool is contained in a class. The constant pool contains elements of different types: method descriptors, native method descriptors, field descriptors and class references. The contents of the constant pool is accessed via indices.

$$\begin{aligned}\text{ConstantPool} \;=\;& (\mathit{methodDesc} : \mathbb{N}_0 \rightarrow \text{MethodDescriptor}) \times \\ & (\mathit{nativeMethodDesc} : \mathbb{N}_0 \rightarrow \text{NativeMethodDescriptor}) \times \\ & (\mathit{fieldDesc} : \mathbb{N}_0 \rightarrow \text{FieldDescriptor}) \times \\ & (\mathit{classRef} : \mathbb{N}_0 \rightarrow \text{ClassName})\end{aligned}$$

A global index is assumed, which can refer to any type of element.

### 3.1.9 Method Descriptor

A method descriptor is a lightweight representation used for the constant pool. A method descriptor holds the relevant values needed to identify a method and find its full representation.

$$\begin{aligned}\text{MethodDescriptor} \;=\;& (\mathit{class} : \text{ClassName}) \times \\ & (\mathit{name} : \text{MethodName}) \times \\ & (\mathit{argType} : \text{Type}^*) \times \\ & (\mathit{retType} : \text{ReturnType})\end{aligned}$$

### 3.1.10 Native Method Descriptor

In TinyBytecode there are two different ways of calling methods: virtual or native. Therefore, another type of method descriptor called a native method descriptor is introduced. The reason for this is that the normal method descriptor contains a class name of a class in the program which contains the method. For a native method this is not the case. Instead, a path to a compiled library of native code is used. This is represented as a string in the constant pool.

$$\begin{aligned}\text{NativeMethodDescriptor} &= (\mathit{library} : \text{LibraryPath}) \times \\ &\quad (\mathit{name} : \text{MethodName}) \times \\ &\quad (\mathit{argType} : \text{Type}^{*}) \times \\ &\quad (\mathit{retType} : \text{ReturnType})\end{aligned}$$

### 3.1.11 Field Descriptor

A field descriptor is a lightweight representation used for the constant pool. A field descriptor holds the relevant values needed to identify a field and find its full representation.

$$\begin{aligned}\text{FieldDescriptor} &= (\mathit{class} : \text{ClassName}) \times \\ &\quad (\mathit{name} : \text{FieldName}) \times \\ &\quad (\mathit{type} : \text{Type})\end{aligned}$$

## 3.2 Syntax

The original Java bytecode has 149 instructions [7]. The simplified version TinyBytecode, contains 16 instructions as shown in Figure 3.2 on the facing page. Compared to the syntax of TinyBytecode in the previous version of this report [16], most of the redundant instructions, which can be achieved by combining other instructions, have been removed. Furthermore, arrays were removed in order to be able to to focus solely on the core parts of the language.

| | | | |
|---|---|---|---|
| | | *Imperative Core* | |
| TinyBytecode | ::= | nop | No operation. |
| | \| | push *t v* | Push *v* onto the stack, type *t*. |
| | \| | pop n | Pops the top n elements off the stack. |
| | \| | dup | Duplicates the top element of the stack |
| | \| | load *t i* | Load local variable index *i*, type *t*. |
| | \| | store *t i* | Store in local variable index *i*, type *t*. |
| | \| | goto *pc* | Jump to program counter *pc*. |
| | \| | if *op pc* | Conditional jump. |
| | \| | comp *op* | Number operation with operator *op*. |
| | | *Objects* | |
| | \| | new *classId* | Instantiate new object of class *classId*. |
| | \| | getfield *fieldId* | Get value of object field. |
| | \| | putfield *fieldId* | Set object field. |
| | | *Methods* | |
| | \| | invokenative *methodId* | Invoke native method *methodId*. |
| | \| | invokevirtual *methodId* | Invoke virtual method *methodId*. |
| | \| | return *t* | Return from a method with type *t*. |
| | | *Exceptions* | |
| | \| | throw | Throw exception |

Figure 3.2: TinyBytecode instructions.

## 3.3 Semantics

In Section 3.2 the syntax of TinyBytecode was introduced and in this section a formal semantics of that syntax will be provided. The constructed semantics is a small step structural operational semantics [6]. In the following sections the formal notation and domains will be introduced followed by the major parts of the semantics: imperative core, objects, methods, exceptions, virtual machine errors and runtime exception, single-event upsets and rollback.

### 3.3.1 Domains

TinyBytecode is modelled according to the specification of the JVM [7] with some simplifications. A value is defined as either a number or a reference.

$$\text{Value} = \text{Ref} + \text{Number}$$

A number in TinyBytecode is simply expressed as an integer.

$$\text{Number} = \mathbb{Z}$$

A reference is a location on the heap or a null reference.

$$\text{Ref} = \text{Location} \cup \{\texttt{null}\}$$

Unlike the JVM, heap locations in TinyBytecode contain only objects.

$$\text{Heap} \quad = \quad \text{Ref} \rightarrow \text{Object}$$

In Section 2.3.2 the local heap was introduced, which is a simplified version of the heap. The local heap can only hold fields, instead of complete objects. Trying to access a non-existent field in the local heap will return $\perp$.

$$\text{LocalHeap} \quad = \quad \text{Ref} \times \text{FieldName} \rightarrow \text{Value}_{\perp}$$

An object is the instantiation of a class and thereby defined by it. Furthermore, an object contains the fields of its class which can be accessed by the function *fieldVal* : FieldName $\rightarrow$Value.

$$\begin{aligned} \text{Object} \quad = \quad & (\mathit{class} : \text{Class}) \times \\ & (\mathit{fieldVal} : \text{FieldName} \rightarrow \text{Value}) \end{aligned}$$

While the heap contains objects, a separate area is needed for local variables in methods. Local variables can be primitive types, or references to heap elements, and they are stored in an area called local memory. Elements in local memory are not accessed by name, but simply by a number.

$$\text{LocalMemory} \quad = \quad \mathbb{N}_0 \rightarrow \text{Value}$$

TinyBytecode is modelled with two stacks: one local stack, the operand stack, for each method and a global stack, the call stack, which holds the frame under execution, as well as frames in queue. The operand stack contain values used for calculations.

$$\begin{aligned}\text{CallStack} &= ((\text{Frame} + \text{ExcFrame}) \times \text{Frame}^*) + \{\epsilon\} \\ \text{OperandStack} &= \text{Value}^*\end{aligned}$$

We use the notation $os_i : os_n$ to indicate elements on a stack. Additionally the notation $os_1 : \cdots : os_n$ is used to denote a range of elements.

A frame is written $\langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), chkp\rangle$. Its components are:

| | |
|---|---|
| $\mathbf{m}$ | The current method under execution. |
| $(\mathbf{lh}, \mathbf{lh_r})$ | The local heap, and the redundant local heap. |
| $(\mathbf{pc}, \mathbf{pc_r})$ | The program counter, and the redundant program counter. |
| $(\mathbf{lv}, \mathbf{lv_r})$ | The local memory area, and the redundant local memory area. |
| $(\mathbf{os}, \mathbf{os_r})$ | The operand stack, and the redundant operand stack. |
| $\mathbf{chkp}$ | The checkpoint. |

The program counter is a positive integer that points to next instruction of the method under execution.

$$\text{PC} = \mathbb{N}_0$$

From the fault-tolerance scheme, a checkpoint is defined as:

$$\text{Checkpoint} = \text{PC} \times \text{LocalMemory} \times \text{OperandStack}$$

A frame with all the necessary components can now be defined as:

$$\begin{aligned}\text{Frame} = {} & \text{Method} \times (\text{LocalHeap} \times \text{LocalHeap}) \times (\text{PC} \times \text{PC}) \times (\text{LocalMemory} \times \\ & \text{LocalMemory}) \times (\text{OperandStack} \times \text{OperandStack}) \times \text{Checkpoint}\end{aligned}$$

Whenever an exception is thrown an exception frame is created on top of the call stack, to signify that finding a handler for the exception is required. An exception frame is defined as:

$$\text{ExcFrame} = \text{Location}$$

No redundancy is used here, since the reference on the exception frame is to be encoded in such a way, that errors are visible and can be corrected.

A TinyBytecode program will continue to execute as long as there are frames on the call stack. A running configuration can be defined as:

$$\text{RunConf} = \text{Heap} \times \text{CallStack}$$

A terminating configuration can be defined as:

$$\text{TermConf} = \text{Heap} \times (\{\epsilon\} + \text{Value})$$

The symbol $\epsilon$ denotes an empty stack or empty memory. These two sets of configurations together give the definition of all valid configurations:

$$\text{Conf} = \text{RunConf} \cup \text{TermConf}$$

Finally, we can define the general form of the semantic reduction rules for TinyBytecode. For a program P ∈ Program, the reduction rules are of the form:

$$P \vdash C \Rightarrow C'$$

In this rule $C, C' \in \text{Conf}$.

## 3.3.2 Imperative Core

This section presents the imperative core of TinyBytecode. It has been split into four different parts with respect to usage. Figure 3.3 on the next page shows the semantic rules for stack manipulation, Figure 3.4 on the facing page shows the semantic rules for local memory manipulation, Figure 3.5 on page 22 shows the semantic rules for program counter manipulation, and lastly, Figure 3.6 on page 22 shows the semantic rules for logical and arithmetic operations. In this section and in the remainder of this chapter, $ChkP$ is used as shorthand notation for $\langle pc_{cp}, lv_{cp}, os_{cp}\rangle$ when the checkpoint is not read or modified for simplicity.

$$\text{push}\ \frac{m.instruction(pc) = \text{push } t\ v}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (v : os, v : os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{pop}\ \frac{m.instruction(pc) = \text{pop } n}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v_1 : \cdots : v_n : os, \\ v_{1_r} : \cdots : v_{n_r} : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{dup}\ \frac{m.instruction(pc) = \text{dup}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v : os, v_r : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (v : v : os, v_r : v_r : os_r), ChkP\rangle : CS\rangle \end{array}}$$

Figure 3.3: Semantic rules for stack manipulation.

$$\text{load}\ \frac{m.instruction(pc) = \text{load } t\ i}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (lv[i] : os, lv_r[i] : os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{store}\ \frac{m.instruction(pc) = \text{store } t\ i}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v : os, v_r : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv[i \mapsto v], lv_r[i \mapsto v_r]), 8os, os_r), \\ ChkP\rangle : CS\rangle \end{array}}$$

Figure 3.4: Semantic rules for local memory manipulation.

$$\text{nop}\ \frac{m.instruction(pc) = \text{nop}}{\begin{array}{c}P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle\end{array}}$$

$$\text{goto}\ \frac{m.instruction(pc) = \text{goto}\ pc'}{\begin{array}{c}P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc', pc'), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle\end{array}}$$

$$\text{if}\ \frac{\begin{array}{c}m.instruction(pc) = \text{if}\ op\ pc_n \\ exp = v_1\ op\ v_2 \quad exp_r = v_{1_r}\ op\ v_{2_r} \\ op \in \{equals, nequals, gt, lt\} \\ pc' = \begin{cases} pc_n & exp = true \\ pc+1 & exp = false \end{cases} \quad pc'_r = \begin{cases} pc_n & exp_r = true \\ pc_r+1 & exp_r = false \end{cases}\end{array}}{\begin{array}{c}P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v_1 : v_2 : os, v_{1_r} : v_{2_r} : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc', pc'_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle\end{array}}$$

Figure 3.5: Semantic rules for program counter manipulation.

$$\text{comp}_{un}\ \frac{\begin{array}{c}m.instruction(pc) = \text{comp}\ op \\ op \in \{neg\} \\ res = op\ v \quad res_r = op\ v_r\end{array}}{\begin{array}{c}P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v : os, v_r : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (res : os, res_r : os_r), ChkP\rangle : CS\rangle\end{array}}$$

$$\text{comp}_{bin}\ \frac{\begin{array}{c}m.instruction(pc) = \text{comp}\ op \\ op \in \{add, sub, mul, div, mod, shl, shr, ushr, and, or, xor\} \\ op \in \{div, mod\} \Rightarrow (v_2 \neq 0 \wedge v_{2_r} \neq 0) \\ res = v_1\ op\ v_2 \quad res_r = v_{1_r}\ op\ v_{2_r}\end{array}}{\begin{array}{c}P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v_1 : v_2 : os, v_{1_r} : v_{2_r} : os_r), \\ ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (res : os, res_r : os_r), ChkP\rangle : CS\rangle\end{array}}$$

Figure 3.6: Semantic rules for logical and arithmetic operations.

### 3.3.3 Objects

In this section the semantic rules concerning objects will be presented. For the allocation of new objects, a function called alloc is used. This function takes a class name and a heap as parameters, and is defined as follows:

$$\text{alloc}(cName, H) = \begin{cases} (ref, H') & \text{if } ref \notin dom(H) \wedge \exists c \in P.classes : c.name = cName \\ & \wedge\, H' = H[ref \mapsto obj] \wedge obj.class = c \\ \perp & \text{otherwise} \end{cases}$$

Alloc returns the reference to the newly allocated object as well as the modified heap, if there exists a class in the program with the provided name, and if the object was allocated correctly.

As discussed earlier, checked instructions are needed for the fault-tolerance scheme. When a checked instruction is encountered, the elements of the frame are compared to the redundant copies to identify if an SEU has occurred. Furthermore, in the checked instructions, the local heap is also committed to the actual heap if no errors are present. For this, a function called commit is formalized as follows:

$$\text{commit}(lh, H) = H'$$

where

$$\begin{aligned} &\forall ref \in dom(H) : \\ &\quad \forall f \in H[ref].fields : \\ &\qquad H'[ref].fieldVal(f.name) = \begin{cases} e.val & \text{if } \exists e \in lh : \\ & (e.ref = ref \wedge \\ & e.fname = f.name) \\ H[ref].fieldVal(f.name) & \text{otherwise} \end{cases} \end{aligned}$$

Commit takes a local heap and a heap as parameters, and accordingly updates the fields on the heap to reflect the entries in the local heap.

Figure 3.7 on the next page shows the semantic rules for objects.

$$\text{new}\;\frac{\begin{array}{c} m.instruction(pc) = \text{new}\ classId \\ lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ cName = m.class.constantPool.classRef(classId) \\ H' = \text{commit}(lh, H) \quad (ref, H'') = \text{alloc}(cName, H') \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), \langle pc_{cp}, lv_{pc}, os_{pc}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H'', \langle m, (\epsilon, \epsilon), (pc+1, pc_r+1), (lv, lv_r), (ref : os, ref : os_r), \\ \langle pc+1, lv, ref : os\rangle\rangle : CS\rangle \end{array}}$$

$$\text{getfield}\;\frac{\begin{array}{c} m.instruction(pc) = \text{getfield}\ fieldId \\ ref \neq null \quad ref_r \neq null \quad obj = H[ref] \quad obj_r = H[ref_r] \\ FD = obj.class.constantPool.fieldDesc(fieldId) \\ FD_r = obj_r.class.constantPool.fieldDesc(fieldId) \\ lhval = lh[ref, FD.name] \\ val = \begin{cases} obj.fieldVal(FD.name) & \text{if } lhval = \perp \\ lhval & \text{otherwise} \end{cases} \\ lhval_r = lh_r[ref_r, FD_r.name] \\ val_r = \begin{cases} obj_r.fieldVal(FD_r.name) & \text{if } lhval_r = \perp \\ lhval_r & \text{otherwise} \end{cases} \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : os, ref_r : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (val : os, val_r : os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{putfield}\;\frac{\begin{array}{c} m.instruction(pc) = \text{putfield}\ fieldId \\ ref \neq null \quad ref_r \neq null \quad obj = H[ref] \quad obj_r = H[ref_r] \\ FD = obj.class.constantPool.fieldDesc(fieldId) \\ FD_r = obj_r.class.constantPool.fieldDesc(fieldId) \\ lh' = lh[ref, FD.name \mapsto val] \\ lh'_r = lh_r[ref', FD'.name \mapsto val_r] \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : val : os, ref_r : val_r : os_r), \\ ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (lh', lh'_r), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \end{array}}$$

Figure 3.7: The semantic rules concerning objects.

### 3.3.4 Methods

As was described earlier, the option for static methods was removed. TinyBytecode contains two other ways of calling methods, namely virtual and native. The virtual call is used on normal method invocation and native is used when accessing methods outside the virtual machine, i.e. interacting with the operating system to perform I/O operations.

When using the virtual method call a method lookup is performed. For this, the function lookup is used which takes a method descriptor and a class as parameters. Lookup is formalized as follows:

$$\text{lookup}(MD, c) = \begin{cases} \perp & \text{if } c = \perp \\ m' & \text{if } \exists m' \in c.methods : (m'.name = MD.name \\ & \wedge\ m'.argType = MD.argtype) \\ \text{lookup}(MD, c.super) & \text{if } \nexists m' \in c.methods : (m'.name = MD.name \\ & \wedge\ m'.argType = MD.argType) \end{cases}$$

If the class contains a method with the name and argument types equal to the provided method descriptor that method is returned. If no method is found, then lookup is called recursively on the super class of the provided class. This lookup process is done to find the correct overload of a given method in the class hierarchy. If no method is found, and lookup is used on the super class of `Object`, $\perp$ is returned instead.

The scope of this project is to introduce fault-tolerance in a virtual machine for TinyBytecode, but since native methods are executed outside the virtual machine's control, special attention must be given. Since native methods are executed at the level of the operating system, our virtual machine will have no control during this time. Potentially, native methods can modify the heap of the virtual machine as well as push values onto the operand stack when they terminate. If SEUs affect the execution of native methods, the virtual machine will have no way of detecting it. Therefore, as a restriction, SEUs are not allowed to occur in the execution of native methods. Furthermore, the behaviour of native methods are restricted in TinyBytecode to only be able to push a value onto the operand stack on termination. No modifications to the heap are allowed.

Figure 3.8 on the following page and 3.9 on page 27 show the semantic rules for invoke and return. In the invoke rules a special notation is used, $|n|$, which denotes the size of that element.

$$\text{invokevirtual}\;\frac{\begin{array}{c} m.instruction(pc) = \text{invokevirtual } methodId \\ lv' = ref : v_1 : \cdots : v_n \quad ref \neq null \\ lv'_r = ref_r : v_{1_r} : \cdots : v_{n_r} \quad ref_r \neq null \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lv' = lv'_r \quad lh = lh_r \\ H' = \text{commit}(lh, H) \quad obj = H[ref] \\ MD = obj.class.constantPool.methodDesc(methodId) \\ m' = \text{lookup}(MD, obj.class) \quad n = |m'.argType| \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : v_1 : \cdots : v_n : os, \\ ref_r : v_{1_r} : \cdots : v_{n_r} : os_r), \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \\ \langle H', \langle m', (\epsilon, \epsilon), (0,0), (lv', lv'), (\epsilon, \epsilon), \langle 0, lv', \epsilon \rangle\rangle \\ : \langle m, (\epsilon, \epsilon), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), \langle pc+1, lv, os \rangle\rangle : CS \rangle \end{array}}$$

$$\text{invokenative}_v\;\frac{\begin{array}{c} m.instruction(pc) = \text{invokenative } methodId \\ ref \neq null \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \quad H' = \text{commit}(lh, H) \\ ref : v_1 : \cdots : v_n = ref_r : v_{1_r} : \cdots : v_{n_r} \\ MD = H[ref].class.constantPool.nativeMethodDesc(methodId) \\ n = |MD.argType| \quad MD.retType = void \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : v_1 : \cdots : v_n : os, \\ ref_r : v_{1_r} : \cdots : v_{n_r} : os_r), \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \\ \langle H', \langle m, (\epsilon, \epsilon), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), \langle pc+1, lv, os \rangle\rangle : CS \rangle \end{array}}$$

$$\text{invokenative}_r\;\frac{\begin{array}{c} m.instruction(pc) = \text{invokenative } methodId \\ ref \neq null \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \quad H' = \text{commit}(lh, H) \\ ref : v_1 : \cdots : v_n = ref_r : v_{1_r} : \cdots : v_{n_r} \\ MD = H[ref].class.constantPool.nativeMethodDesc(methodId) \\ n = |MD.argType| \quad MD.retType \neq void \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : v_1 : \cdots : v_n : os, \\ ref_r : v_{1_r} : \cdots : v_{n_r} : os_r), \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \\ \langle H', \langle m, (\epsilon, \epsilon), (pc+1, pc_r+1), (lv, lv_r), (val : os, val : os_r), \\ \langle pc+1, lv, val : os \rangle\rangle : CS \rangle \end{array}}$$

Figure 3.8: The semantic rules concerning method invocation.

$$\text{return}_1\;\frac{\begin{array}{c} m.instruction(pc) = \text{return } t \\ t = void \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \\ H' = \text{commit}(lh, H) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : \epsilon\rangle \Rightarrow \\ \langle H', \epsilon\rangle \end{array}}$$

$$\text{return}_2\;\frac{\begin{array}{c} m.instruction(pc) = \text{return } t \\ t = void \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \\ H' = \text{commit}(lh, H) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle \\ : \langle m', (lh', lh'_r), (pc', pc'_r), (lv', lv'_r), (os', os'_r), ChkP'\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m', (lh', lh'_r), (pc', pc'_r), (lv', lv'_r), (os', os'_r), ChkP'\rangle : CS\rangle \end{array}}$$

$$\text{return}_3\;\frac{\begin{array}{c} m.instruction(pc) = \text{return } t \\ t \neq void \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \quad v = v_r \\ H' = \text{commit}(lh, H) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v : os, v' : os_r), ChkP\rangle : \epsilon\rangle \Rightarrow \\ \langle H', v\rangle \end{array}}$$

$$\text{return}_4\;\frac{\begin{array}{c} m.instruction(pc) = \text{return } t \\ t \neq void \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lh = lh_r \quad v = v_r \\ H' = \text{commit}(lh, H) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v : os, v_r : os_r), \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle \\ : \langle m', (lh', lh'_r), (pc', pc'_r), (lv', lv'_r), (os', os'_r), \langle pc'_{cp}, lv'_{cp}, os'_{cp}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m', (lh', lh'_r), (pc', pc'_r), (lv', lv'_r), (v : os', v : os'_r), \\ \langle pc'_{cp}, lv'_{cp}, v : os'_{cp}\rangle\rangle : CS\rangle \end{array}}$$

Figure 3.9: The semantic rules concerning method return.

### 3.3.5 Exceptions

This section presents the aspect of exceptions with regards to programmer thrown exceptions as well as the handle rules, which are used by the throw rule as well as the virtual machine errors and runtime exceptions presented in Section 3.3.6. When a throw instruction is encountered, prior to that, an exception object was allocated and instantiated. The function tryHandle is used to check whether or not an exception handler exists in the current frame that can handle the thrown exception. If a handler exists, the program counter jumps to the address specified by the handler. If a handler does not exist, the call-chain is traversed in order to find a handler, and if no handler is found, the program terminates.

For the function tryHandle, it is assumed that the list of exception handlers located in each method are ordered in a certain way. The list must be ordered such that the more specialized cases are placed first and the more general cases last. This means that the first handler found for a given exception is always the most specialized, and is thus the one which should be used.

$$\text{tryHandle}(m, pc, EC) = \begin{cases} m.\mathit{handlers}(i).\mathit{target} & \text{if } \exists i \in \mathbb{N}_0 : \\ & \quad (\ h = m.\mathit{handlers}(i) \\ & \quad \wedge \text{validHandler}(h, pc, EC) \\ & \quad \wedge \forall j \in \mathbb{N}_0 : \\ & \qquad (\ h' = m.\mathit{handlers}(j) \\ & \qquad \wedge \text{validHandler}(h', pc, EC)\ ) \\ & \quad \Rightarrow\ i \leq j\ ) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{validHandler}(h, pc, EC) = \begin{cases} \mathit{true} & \text{if } h.\mathit{from} \leq pc \leq h.\mathit{to} \wedge EC \leq h.\mathit{type} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Figure 3.10 on the facing page shows the semantic rules for exceptions.

$$\text{throw}\;\frac{\begin{array}{c} m.instruction(pc) = \text{throw} \\ lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad ref = ref_r \quad os = os_r \\ H' = \text{commit}(lh, H) \\ ref \neq null \quad EC = H'[ref].class \quad EC \leq \text{Throwable} \\ pc' = \text{tryHandle}(m, pc, EC) \\ localHandler = \langle m, \epsilon, \epsilon, pc', pc', lv, lv, ref : \epsilon, ref : \epsilon, \langle pc', lv, ref : \epsilon \rangle\rangle \\ \text{nextFrame} = \begin{cases} \langle ref \rangle & \text{if } pc' = \perp \\ localHandler & \text{otherwise} \end{cases} \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, lh, lh_r, pc, pc_r, lv, lv_r, ref : os, ref_r : os_r, \\ \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \langle H', \text{nextFrame} : CS \rangle \end{array}}$$

$$\text{handle}_1\;\frac{\begin{array}{c} lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ EC = H[ref].class \quad \text{tryHandle}(m, pc, EC) = pc' \end{array}}{\begin{array}{c} P \vdash \langle H, \langle ref \rangle : \langle m, lh, lh_r, pc, pc_r, lv, lv_r, os, os_r, \\ \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \\ \langle H, \langle m, lh, lh_r, pc', pc', lv, lv_r, ref : \epsilon, ref : \epsilon, \langle pc', lv, ref : \epsilon \rangle\rangle : CS \rangle \end{array}}$$

$$\text{handle}_2\;\frac{\begin{array}{c} lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ EC = H[ref].class \quad \text{tryHandle}(m, pc, EC) = \perp \end{array}}{\begin{array}{c} P \vdash \langle H, \langle ref \rangle : \langle m, lh, lh_r, pc, pc_r, lv, lv_r, os, os_r, \\ \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS \rangle \Rightarrow \langle H, \langle ref \rangle : CS \rangle \end{array}}$$

Figure 3.10: Semantic rules for exceptions.

### 3.3.6 Virtual Machine Errors and Runtime Exceptions

The virtual machine errors and runtime exceptions are not to be confused with the representation of programmer thrown exceptions as presented in Section 3.3.5. The semantic rules presented in this section are the internal exceptions that can occur during execution of programs. These exceptions are handled differently from programmer thrown exceptions. Instead of assuming that allocation of a new exception object has taken place prior to these rules, it uses preallocated exception objects. The references to these objects are stored in a global reference table defined in Section 3.1.3.

Figure 3.11 on the next page shows the semantic rules for virtual machine errors and runtime exceptions.

$$\text{new}_{exc}\frac{\begin{array}{c} m.instruction(pc) = \text{new } classId \\ lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ H' = \text{commit}(lh, H) \\ cName = m.class.constantPool.classRef(classId) \\ \bot = alloc(cName, H') \quad ref = P.exceptionRef(0) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H', \langle ref\rangle : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{comp}_{exc}\frac{\begin{array}{c} m.instruction(pc) = \text{comp } op \\ lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ op \in \{div, mod\} \quad (v_2 = 0 \lor v_{2_r} = 0) \\ ref = P.exceptionRef(2) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (v_1 : v_2 : os, \\ v_{1_r} : v_{2_r} : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle ref\rangle : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \end{array}}$$

$$\text{getfield}_{exc}\frac{\begin{array}{c} m.instruction(pc) = \text{getfield } fieldId \\ lh = lh_r \quad pc = pc_r \quad lv = lv_r \quad os = os_r \\ (ref = null \lor ref_r = null) \\ ref' = P.exceptionRef(1) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : os, ref_r : os_r), ChkP\rangle : CS\rangle \Rightarrow \\ \langle H, \langle ref'\rangle : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS\rangle \end{array}}$$

Figure 3.11: Semantic rules for runtime exceptions.

### 3.3.7 Single-Event Upsets

This section presents the semantic rules modelling the errors described in the fault model presented in Section 2.1. For clarification, a new operator is introduced for this section, namely $\asymp$. Whenever $\asymp v$ is read, it means that an SEU has changed the encoding of $v$. Furthermore, to specify how it is changed, the notation $\asymp_1$ is used to denote that the encoding is different by one bit etc.

Figure 3.12 on the facing page and 3.13 on page 32 shows the semantic rules for single-event upsets.

$$\text{pc-flip}\;\frac{pc' \asymp_1 pc}{\begin{array}{c}P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc', pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle\end{array}}$$

$$\text{pc-flip}_r\;\frac{pc' \asymp_1 pc_r}{\begin{array}{c}P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc'), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle\end{array}}$$

$$\text{os-flip}\;\frac{elem' \asymp_1 elem}{\begin{array}{c}P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os_1 : elem : os_2, os_r), \\ ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os_1 : elem' : os_2, os_r), \\ ChkP\rangle : CS_2\rangle\end{array}}$$

$$\text{os-flip}_r\;\frac{elem' \asymp_1 elem_r}{\begin{array}{c}P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_{1_r} : elem_r : os_{2_r}), \\ ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_{1_r} : elem' : os_{2_r}), \\ ChkP\rangle : CS_2\rangle\end{array}}$$

Figure 3.12: Semantic rules for errors on the operand stacks and program counters.

$$\text{lv-flip}\;\dfrac{\begin{array}{c} val = lv[i] \\ val' \asymp_1 val \end{array}}{\begin{array}{c} P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv[i \mapsto val'], lv_r), (os, os_r), ChkP\rangle : CS_2\rangle \end{array}}$$

$$\text{lv-flip}_r\;\dfrac{\begin{array}{c} val = lv_r[i] \\ val' \asymp_1 val \end{array}}{\begin{array}{c} P \vdash \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r[i \mapsto val']), (os, os_r), ChkP\rangle : CS_2\rangle \end{array}}$$

$$\text{lh-flip}\;\dfrac{\begin{array}{c} ref' \asymp_1 ref \quad fname' \asymp_1 fname \quad val' \asymp_1 val \\ \text{flippedEntry} = \\ (\langle ref', fname, val\rangle \vee \langle ref, fname', val\rangle \vee \langle ref, fname, val'\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, CS_1 : \langle m, (lh_1 : \langle ref, fname, val\rangle : lh_2, lh_r), (pc, pc_r), (lv, lv_r), \\ (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh_1 : \text{flippedEntry} : lh_2, lh_r), (pc, pc_r), (lv, lv_r), \\ (os, os_r), ChkP\rangle : CS_2\rangle \end{array}}$$

$$\text{lh-flip}_r\;\dfrac{\begin{array}{c} ref' \asymp_1 ref \quad fname' \asymp_1 fname \quad val' \asymp_1 val \\ \text{flippedEntry} = \\ (\langle ref', fname, val\rangle \vee \langle ref, fname', val\rangle \vee \langle ref, fname, val'\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, CS_1 : \langle m, (lh, lh_{1_r} : \langle ref, fname, val\rangle : lh_{2_r}), (pc, pc_r), (lv, lv_r), \\ (os, os_r), ChkP\rangle : CS_2\rangle \Rightarrow \\ \langle H, CS_1 : \langle m, (lh, lh_{1_r} : \text{flippedEntry} : lh_{2_r}), (pc, pc_r), (lv, lv_r), \\ (os, os_r), ChkP\rangle : CS_2\rangle \end{array}}$$

Figure 3.13: Semantic rules for errors in the local memories and local heaps.

In the aforementioned rules the notation $CS_1 : \cdots : CS_2$ and $os_1 : \cdots : os_2$ are used. This denotes that the occurrence of the flipped element can be in any of the elements on the operand stack, in any frame on the call stack, but only the one element is affected.

### 3.3.8 Rollback

This section introduces the rules needed to recover from SEUs. When an SEU occurs, a rollback to the checkpoint is performed. This also means that a rule for each version of a checked instruction is needed. Only a few rules are presented for clarification, written in a generalized format.

$$\text{invokevirtual}_{RB}\ \frac{\begin{array}{c} m.\mathit{instruction}(pc) = \text{invokevirtual } methodId \\ (pc \neq pc_r \lor lv \neq lv_r \lor os \neq os_r \lor lh \neq lh_r) \\ \text{validate}(\langle pc_{cp}, lv_{cp}, os_{cp}\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), \\ \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (\epsilon, \epsilon), (pc_{cp}, pc_{cp}), (lv_{cp}, lv_{cp}), (os_{cp}, os_{cp}), \\ \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \end{array}}$$

$$\text{return}_{RB}\ \frac{\begin{array}{c} m.\mathit{instruction}(pc) = \text{return } t \\ (pc \neq pc_r \lor lv \neq lv_r \lor os \neq os_r \lor lh \neq lh_r) \\ \text{validate}(\langle pc_{cp}, lv_{cp}, os_{cp}\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (\epsilon, \epsilon), (pc_{cp}, pc_{cp}), (lv_{cp}, lv_{cp}), (os_{cp}, os_{cp}), \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \end{array}}$$

$$\text{new}_{RB}\ \frac{\begin{array}{c} m.\mathit{instruction}(pc) = \text{new } classId \\ (pc \neq pc_r \lor lv \neq lv_r \lor os \neq os_r \lor lh \neq lh_r) \\ \text{validate}(\langle pc_{cp}, lv_{cp}, os_{cp}\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (os, os_r), \langle pc_{cp}, lv_{pc}, os_{pc}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (\epsilon, \epsilon), (pc_{cp}, pc_{cp}), (lv_{cp}, lv_{cp}), (os_{cp}, os_{cp}), \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \end{array}}$$

$$\text{getfield}_{exc_{RB}}\ \frac{\begin{array}{c} m.\mathit{instruction}(pc) = \text{getfield } fieldId \\ (pc \neq pc_r \lor lv \neq lv_r \lor os \neq os_r \lor lh \neq lh_r \lor ref \neq ref_r) \\ (ref = null \lor ref_r = null) \\ \text{validate}(\langle pc_{cp}, lv_{cp}, os_{cp}\rangle) \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : os, ref_r : os_r), \\ \langle pc_{cp}, lv_{pc}, os_{pc}\rangle\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, (\epsilon, \epsilon), (pc_{cp}, pc_{cp}), (lv_{cp}, lv_{cp}), (os_{cp}, os_{cp}), \\ \langle pc_{cp}, lv_{cp}, os_{cp}\rangle\rangle : CS\rangle \end{array}}$$

Figure 3.14: The semantic rules concerning rollbacks in the presence of errors.

# CHAPTER 4

# IMPLEMENTATION

As mentioned in Chapter 2 the Tiny Virtual Machine is based on the Java Virtual Machine specification, and as such contains a subset of the elements found in the Java virtual machine, with added redundancy.

In this chapter the concrete implementation of the TVM will be presented and explained.

## 4.1 Class File Format

The first action performed by the TVM is loading the class files necessary for executing a given program. Each class is described in a class file, which is a binary file that is loaded and interpreted by the TVM.

Table 4.1 shows the overall structure of a class file.

| **Element name** | **Size in *bytes*** | **Description** |
|---|---|---|
| Magic number | 2 | Two bytes which identify the file as a class file. |
| CP count | 2 | The size of the constant pool |
| Constant pool | CP count | Contains elements described in Section 4.1.1 |
| This | 2 | A reference to this class in the constant pool |
| Super | 2 | A reference to the super class of this in the constant pool |
| Method count | 2 | The number of methods in this class |
| Methods | | Contains elements described in Section 4.1.2 |

Table 4.1: The layout of a class file.

The file starts with a *magic number* which is a two byte value used to correctly identify it as a class file. The value is used in case an arbitrary file is loaded by the TVM which doesn't follow the TinyBytecode class file format. If a file is loaded and it does not start with the magic number, the TVM will not attempt to parse it and will halt, informing the user that an invalid class file was provided. Although any number would work, we chose to use the same magic number as is specified by the Java virtual machine specification: `0xCAFEBABE`.

### 4.1.1 Constant Pool

The constant pool is a collection of constants which is specific to a class. The constant pool is a lookup table which is used at runtime to get information about a class such as its fields, methods and class dependencies.

All constant pool elements consist of a *tag* which is a 1 byte value which denotes which type of constant pool element it is. The following are the different types of elements which the constant pool can contain:

**ClassName**

Tag: 1 *byte*, Tag identifies the element type
ClassName: 2 *bytes*, String reference, the name of this class

The ClassName element describes a class. It contains a single reference which points to a string element containing the name of the class.

**FieldDescriptor**

Tag: 1 *byte*, Tag identifies the element type
ClassName: 2 *bytes*, String reference, the name of this field's class
FieldName: 2 *bytes*, String reference, the name of this field
FieldType: 2 *bytes*, Type reference, the type of this field

The FieldDescriptor element describes a field in a class. It contains three references where the first two are string references to the name of the class and the name of the field respectively. The third is a reference to a type element which is the type of the field.

**MethodDescriptor**

Tag: 1 *byte*, Tag identifies the element type
ClassName: 2 *bytes*, String reference
MethodName: 2 *bytes*, String reference
ArgCount: 2 *bytes*, the number of arguments this method takes
ArgTypes: 2 * ArgCount *bytes*, Type references, the types of every argument to this method
RetType: 2 *bytes*, Type reference, the return type of this method.

The MethodDescriptor element describes a method in a class. The actual implementation of the method is not described in the MethodDescriptor element, but is instead described as an element in the Methods section of the class in which it is implemented. The MethodDescriptor element only describes the information required to call the method.

It contains two references to string elements containing the class name and method name respectively. It also contains a list of type element references called ArgTypes which describes the types of all the arguments for the method. The length of this list is described by the ArgCount number. Finally, it contains a type reference which describes the return type of the method.

**NativeMethodDescriptor**

Tag: 1 *byte*, Tag identifies the element type
Library: 2 *bytes*, String reference
MethodName: 2 *bytes*, String reference
ArgCount: 2 *bytes*, the number of arguments this method takes
ArgTypes: 2 * ArgCount *bytes*, Type references, the types of every argument to this method

The NativeMethodDescriptor element describes a native method. It is quite similar to the MethodDescriptor element as it also contains a method name reference, the same kind of descriptor for the method arguments. However, unlike the MethodDescriptor, the NativeMethodDescriptor contains a library string reference instead of a class name reference. Since a native method is not implemented in TinyBytecode the concept of classes does not apply to it. Instead, the Library string contains the path to the native library in which the method to be called is located.

Furthermore, the NativeMethodDescriptor does not contain a return type reference. This is due to the constraint placed on native methods in TinyBytecode where all native methods must return an integer value.

**String**

Tag: 1 *byte*, Tag identifies the element type
Length: 2 *bytes*
Bytes: Length *bytes*

The String element contains a string. It consists of a length value and a number of bytes equal to the length.

**Type**

Tag: 1 *byte*, Tag identifies the element type
Type: 1 *byte*, one of "b"(byte), "i"(integer), "z"(boolean), "l"(object), "v"(void)
ClassName: 2 *bytes*, String reference, optional, only present if Type = "l"

The Type element describes a type. The Type value can be either one of the primitive types "b", "i", "z" or "v", which are: byte, integer, boolean or void respectively, or it can be an object type denoted with "l". If the Type value is an object type, the Type element will contain an extra string element reference for the class name of the object.

### 4.1.2 Methods

This section of the class file describes the implementation of the methods in the class. Each method consists of a Method element:

**Method**

MethodDescriptor: 2 *bytes*, MethodDescriptor reference
Max stack: 2 *bytes*, maximum size of the operand stack
Max locals: 2 *bytes*, maximum elements in local memory
Code length: 2 *bytes*, the amount of instructions, including instruction parameters
Code: Code length *bytes*, the code for the method, including exception handlers and instruction parameters
Handler count: 2 *bytes*, the amount of exception handlers in this method
Handlers: 8 * Handler count *bytes*, an array of Handler elements

The Method element contains a reference to the MethodDescriptor element in the constant pool to which it belongs. It also contains its maximum number of elements on the operand stack as well as the maximum number of local variables in the method.

Most importantly, the Method element contains the TinyBytecode which is executed when the method is called. The TinyBytecode is stored in the Code field which is preceded by a value describing the length of the code contained within the field.

Finally, the Method element optionally contains a number of exception handlers. If the number of handlers is larger than 0 then the Method element will describe the handlers in the following format:

**Handler**

Start pc: 2 *bytes*, the address of the first instruction that this handler covers
End pc: 2 *bytes*, the address of the last instruction that this handler covers
Handler pc: 2 *bytes*, the address of the first instruction in the actual handler
Type: 2 *bytes*, Type reference, this handler handles exception that extend this type

The handler element contains two values which describe the range of instructions where this handler will be called if an error occurs. Is also contains a reference to the first instruction of the handler, which the program counter will jump to if an exception is caught. Finally, it contains a reference to the type of exception the handler catches.

## 4.2 The Tiny Virtual Machine

The TVM is developed in Java. The choice of language was made mostly based on previous experience with Java. Choosing a language we were familiar with allowed us to spend less time on developing the virtual machine as we did not have to spend time adjusting to an unfamiliar syntax. However, due some of the built-in optimizations in Java a number of unexpected problems became apparent during development which ended up costing additional time for debugging. These optimizations and their effects will be presented in Section 4.2.2.

Figure 4.1 on the next page shows a partial UML class diagram of the TVM. The diagram shows the structure of the TVM, with auxiliary methods such as getters and setters removed from each class.

### 4.2.1 The TinyVM Class

The `TinyVM` class is the main class, containing the `main` method which is executed when starting the TVM. On start-up it loads the class file corresponding to the class name provided as an argument to the `main` method. This is done with the `load` method which takes a class name as its only parameter and finds the appropriate class file and parses it. For each ClassName element in the constant pool of the class file the class name is added to a list of classes to be loaded. The TVM iterates through this list until it is empty, loading all classes it contains.

The built-in exception classes of TinyBytecode are automatically added to this list even if no class references them.

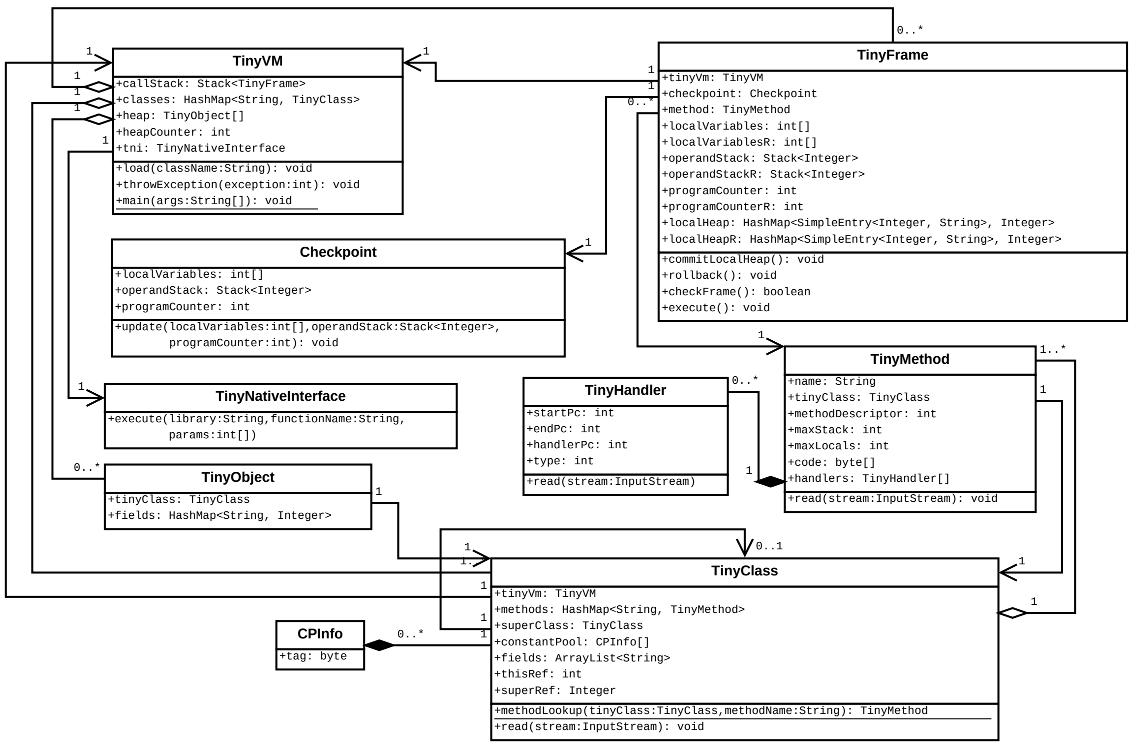
TinyVM
+callStack: Stack<TinyFrame>
+classes: HashMap<String, TinyClass>
+heap: TinyObject[]
+heapCounter: int
+tni: TinyNativeInterface
+load(className:String): void
+throwException(exception:int): void
+main(args:String[]): void
TinyFrame
+tinyVm: TinyVM
+checkpoint: Checkpoint
+method: TinyMethod
+localVariables: int[]
+localVariablesR: int[]
+operandStack: Stack<Integer>
+operandStackR: Stack<Integer>
+programCounter: int
+programCounterR: int
+localHeap: HashMap<SimpleEntry<Integer, String>, Integer>
+localHeapR: HashMap<SimpleEntry<Integer, String>, Integer>
+commitLocalHeap(): void
+rollback(): void
+checkFrame(): boolean
+execute(): void
Checkpoint
+localVariables: int[]
+operandStack: Stack<Integer>
+programCounter: int
+update(localVariables:int[],operandStack:Stack<Integer>, programCounter:int): void
TinyNativeInterface
+execute(library:String,functionName:String, params:int[])
TinyHandler
+startPc: int
+endPc: int
+handlerPc: int
+type: int
+read(stream:InputStream)
TinyMethod
+name: String
+tinyClass: TinyClass
+methodDescriptor: int
+maxStack: int
+maxLocals: int
+code: byte[]
+handlers: TinyHandler[]
+read(stream:InputStream): void
TinyObject
+tinyClass: TinyClass
+fields: HashMap<String, Integer>
TinyClass
+tinyVm: TinyVM
+methods: HashMap<String, TinyMethod>
+superClass: TinyClass
+constantPool: CPInfo[]
+fields: ArrayList<String>
+thisRef: int
+superRef: Integer
+methodLookup(tinyClass:TinyClass,methodName:String): TinyMethod
+read(stream:InputStream): void
CPInfo
+tag: byte
0..*
1
1..*
0..1

Figure 4.1: TVM class diagram.

Once all necessary classes have been loaded the TVM creates an instance of `TinyObject` which is an instance of the class initially loaded by the TVM. This object is pushed onto the heap, whereafter the TVM finds the `main` method of the initially loaded class and calls it on the object, pushing an instance of `TinyFrame` onto the call stack.

The TVM then enters a loop which keeps calling `execute` on the topmost `TinyFrame` instance on the call stack until the stack is empty.

```
tinyVm.getHeap()[tinyVm.getHeapCounter()] = new
     TinyObject(tinyVm.getClasses().get(className));
tinyVm.incrementHeapCounter();

TinyMethod mainMethod =
     tinyVm.getClasses().get(className).getMethods().get("main()");
int[] localVariables = new int[mainMethod.getMaxLocals()];
tinyVm.getCallStack().push(new TinyFrame(tinyVm, localVariables, mainMethod));

while(!tinyVm.getCallStack().isEmpty()) {
   tinyVm.getCurrentFrame().execute();
   tinyVm.incrementInstructionCounter();
}
```

Listing 4.1: The main loop of the TVM.

Listing 4.1 on the facing page shows the above mentioned procedure as implemented in the TVM.

### 4.2.2 `TinyFrame` Class

As shown in Figure 4.1 on the preceding page the `TinyFrame` class contains a reference to an instance of `TinyMethod`. When the `execute` method is called the `TinyFrame` instance retrieves the `code` array from the `TinyMethod` object and gets the byte at the address corresponding to its `programCounter` field. This value is used to load the correct instruction to execute.

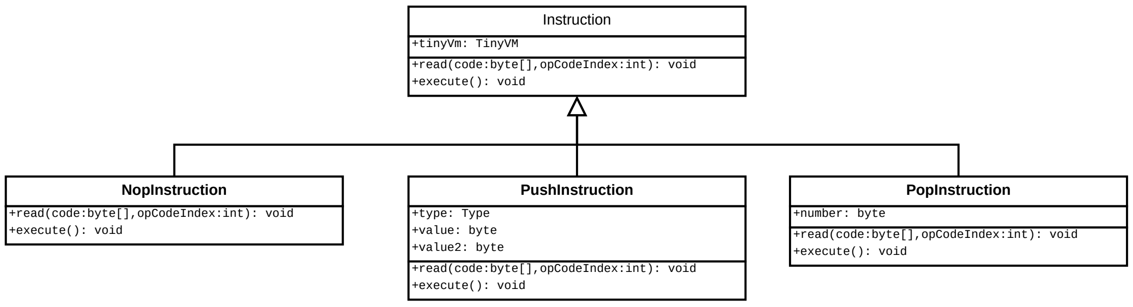

Figure 4.2: Three examples of how instructions are implemented in the TVM.

Each instruction in the TVM is implemented as a subclass of the class `Instruction` as partially shown in Figure 4.2. The `Instruction` class is an abstract class providing the `read` and `execute` methods which are overridden by all of its subclasses. When executing an instruction the `execute` method of the `TinyFrame` class first creates an appropriate instance of a subclass of `Instruction`, whereafter it calls the `read` method followed by the `execute` method on the `Instruction` instance.

```java
public void execute() {
   Instruction instruction = null;

   OpCode opCode = OpCode.get(method.getCode()[codePointer]);
   switch(opCode) {
   case NOP:
      instruction = new NopInstruction(tinyVm);
      break;
   case POP:
      instruction = new PopInstruction(tinyVm);
      break;
   case PUSH:
      instruction = new PushInstruction(tinyVm);
      break;
       ...
   }
   instruction.read(method.getCode(), codePointer);
   instruction.execute();
}
```

Listing 4.2: The `execute` method of the `TinyFrame` class.

Listing 4.2 shows the implementation of the `execute` method in `TinyFrame`. Calling `execute` on an instance of `Instruction` affects the TVM in accordance with the semantics presented in Chapter 3.

```java
@Override
public void read(byte[] code, int opCodeIndex) {
   type = Type.get(code[opCodeIndex + 1]);
   value = code[opCodeIndex + 2];
   if(type == Type.INT || type == Type.REF) {
      value2 = code[opCodeIndex + 3];
   }
}

@Override
public void execute() {
   tinyVm.getCurrentFrame().getOperandStack().push(new Integer(getValue()));
   tinyVm.getCurrentFrame().getOperandStackR().push(new Integer(getValue()));

   if(type == Type.INT || type == Type.REF) {
      tinyVm.getCurrentFrame().incrementProgramCounter(4);
      tinyVm.getCurrentFrame().incrementProgramCounterR(4);
   } else {
      tinyVm.getCurrentFrame().incrementProgramCounter(3);
      tinyVm.getCurrentFrame().incrementProgramCounterR(3);
   }
}
```

Listing 4.3: The `PushInstruction` class.

Listing 4.3 on the facing page shows the `read` and `execute` methods of the `PushInstruction` class. The `read` method implements the code necessary to correctly read a `push` instruction. It reads a single byte to determine the type of the value to be pushed. Depending on the type it then either reads one or two bytes which it stores in the fields `value` and `value2`.

When executing the instruction in the `execute` method the operand stack is retrieved from the current frame and the previously read value is pushed onto the stack. An auxiliary method `getValue` is used to get the correct value depending on whether the value was one or two bytes long. The same is done for the redundant operand stack `operandStackR`.

Finally depending on the type of the push instruction the program counter and redundant program counter are both incremented by either 4 or 3. The next instruction is then executed in the same manner.

### Checked Instructions

When executing checked instruction such as `invokevirtual` the `TinyFrame` class provides three methods which are used by the appropriate instructions: `checkFrame`, `commitLocalHeap` and `rollback`.

To illustrate how these are implemented the `execute` method of the `InvokeVirtualInstruction` class will be used as shown in Listing 4.4 on the next page.

```java
public void execute() {
   if(tinyVm.getCurrentFrame().checkFrame()) {
      tinyVm.getCurrentFrame().commitLocalHeap();

      ...

      TinyMethod method = TinyClass.methodLookup(
         tinyVm.getClasses().get(className.getBytesString()), methodName);
      int[] localVariables = new int[method.getMaxLocals()];
      for(int i = 0; i < methodDescriptor.getArgCount() + 1; i++) {
         localVariables[i] = tinyVm.getCurrentFrame().getOperandStack().pop();
         tinyVm.getCurrentFrame().getOperandStackR().pop();
      }

      tinyVm.getCurrentFrame().incrementProgramCounter(3);
      tinyVm.getCurrentFrame().incrementProgramCounterR(3);

      tinyVm.getCurrentFrame().getCheckpoint().update(
           tinyVm.getCurrentFrame().getLocalVariables().clone(),
           tinyVm.getCurrentFrame().getOperandStack(),
           tinyVm.getCurrentFrame().getProgramCounter());

      tinyVm.getCallStack().push(
         new TinyFrame(tinyVm, localVariables, method));
   } else {
      tinyVm.getCurrentFrame().rollback();
   }
}
```

Listing 4.4: The `execute` method of the `InvokeVirtualInstruction` class.

At the start of each checked instruction the `checkFrame` method is called on the current frame. The `checkFrame` method compares the fields: `localVariables`, `operandStack`, `programCounter` and `localHeap` to their corresponding redundant counterparts. If no difference is found the `checkFrame` method returns `true` and the instruction then calls `commitLocalHeap`. This method copies the values from the `localHeap` of the current frame to the `heap` in `TinyVM`, whereafter it empties both `localHeap` and `localHeapR`. The instruction then proceeds to execute according to the semantics in Chapter 3. In this case the instruction pops the arguments for the virtual method to be called from the operand stack and puts them in the local variables of the frame for the new method. The values are also popped on the redundant operand stack but are not used. The program counter and the redundant program counter are both incremented appropriately.

Finally, before pushing the new frame onto the call stack, the checkpoint for the current frame is updated with the current local variables, operand stack and program counter.

In case the `checkFrame` method had returned `false`, due to a bitflip, the entire instruction would have been skipped and instead the `rollback` method would have been called on the current frame.

Listing 4.5 shows the `rollback` method of `TinyFrame`.

```
public void rollback() {
   localHeap.clear();
   localHeapR.clear();

   operandStack.clear();
   operandStackR.clear();

   for(int i = 0; i < checkpoint.operandStack.size(); i++)  {
      operandStack.add(new
          Integer(checkpoint.operandStack.get(i).intValue()));
      operandStackR.add(new
          Integer(checkpoint.operandStack.get(i).intValue()));
   }

   localVariables = checkpoint.getLocalVariables().clone();
   localVariablesR = checkpoint.getLocalVariables().clone();

   programCounter = checkpoint.getProgramCounter();
   programCounterR = checkpoint.getProgramCounter();
}
```

Listing 4.5: The `rollback` method of the `TinyFrame` class.

The `rollback` method starts by clearing both local heaps and operand stacks. It then copies all of the values from the saved operand stack in the checkpoint into both operand stacks in the current frame. After this, the local variables from the checkpoint are copied into the local variable arrays of the current frame. Finally, the program counters are updated to the value from the checkpoint.

### Java Complications

The `rollback` instruction reveals some of the complications that arise when using Java to implement redundancy.

The operand stack in TVM is implemented as a field of the type `Stack<Integer>`. The `Stack<E>` is a class provided by the `java.util` package, which provides an implementation of a stack with a given generic type `E`. A restriction is placed on generic types in Java where it is impossible to instantiate generic types with primitive types [12]. This means that in the case of an integer stack one cannot create a field with the primitive type `Stack<int>` but instead one must use the wrapper class for integers `Stack-<Integer>`. This presents a number of problems because Java handles operators on objects differently than on primitives.

If one were to create two `int` variables `x` and `y` and set `x = y`, `x` would receive the value of `y`. Modifying `y` after this would leave `x` unaffected. If one were instead to create the same two variables but using the type `Integer` and set `x = y`, `x` would share `y`'s reference. This means that any modification done on `y` would be reflected by `x`.

Furthermore, this means that methods such as `clone` which can be used on primitive collections such as `int[]` as shown with the local variable arrays in Listing 4.5 on the previous page, do not work as expected on `Stack<Integer>`. This is the reason why the `rollback` method iterates over the operand stack and copies the values manually while using the `clone` method on local variables.

Another issue with using the `Integer` class is the caching mechanism it implements. When using the shorthand assignment for `Integer` which allows a programmer to write `Integer x = 5;` instead of `Integer x = new Integer(5);`, all values between -128 and 127 will be cached in the `Integer` class [11]. This means that two variables `x` and `y` of the type `Integer` which have both been initialized with the value 5, will share the same reference. So modifying either one will affect the other.

While the reasons for these decisions are justifiable and documented they still provided a deal of frustration during the development of the TVM due to their unintuitive nature. When implementing the methods `rollback` and `checkFrame` seemingly unexplainable errors were occurring due to integer values being copied incorrectly because of caching and reference assignment.

### 4.2.3 Native Methods

Native methods in TVM are implemented using the Java Native Interface (JNI). The JNI provides a way to execute native code from Java by marking methods with the keyword `native`, telling Java that the methods implementation is provided by a native library. In TVM the `TinyNativeInterface` class provides the implementation for calling native methods.

Listing 4.6 shows the implementation of the `TinyNativeInterface` class.

```java
public class TinyNativeInterface {
   public native int execute(String library, String function, int[] params);

   static {
      System.loadLibrary("tni");
   }
}
```

Listing 4.6: The `TinyNativeInterface` class.

Since the implementation of the `execute` method is written in native code, the class file is quite small as it only loads the `tni` library and provides the execute method to the rest of the TVM.

The `tni` native library is where the actual implementation for executing native code in the TVM is. Java does not support dynamic loading and unloading of native libraries.

The `System.loadLibrary` method does not have a corresponding method for unloading libraries, which means that once a library has been bound to a class it cannot be swapped for another. This presents a problem since we want to enable the TVM to run arbitrary native code at any given time. The solution to the problem is delegating the loading and unloading of native libraries to our own native library `tni` which is the only library Java itself needs to load.

As shown in Listing 4.6 on the preceding page the `execute` method takes two strings and an integer array as parameters. These values are passed to the `tni` library which then loads the library provided by the `library` string and executes the function provided by the `function` string with the parameters contained in the `params` array.

The implementation of the `tni` library is shown in Listing 4.7.

```c
#include <jni.h>
#include "tni.h"
#include <windows.h>

typedef int(CALLBACK* NativeFunctionType)(int *params, int size);

JNIEXPORT jint JNICALL Java_dk_aau_d101f14_tinyvm_TinyNativeInterface_execute
  (JNIEnv * env, jobject obj, jstring library_path, jstring function_name,
      jintArray parameters) {
   ...
   library_dll = LoadLibrary(library_path_string);

   if (library_dll != NULL) {
      ...
      library_func = (NativeFunctionType)GetProcAddress(library_dll,
          function_name_string);

      if (library_func != NULL) {
         ...
         return library_func(native_params, params_length);
      }
      else {
         FreeLibrary(library_dll);
         return 0;
      }
   }
   else {
      FreeLibrary(library_dll);
      return 0;
   }
}
```

Listing 4.7: The implementation of the `tni` library.

The JNI follows a naming convention which allows the JVM to map a function marked as `native` to an actual implementation in a loaded native library. In the case of the `execute` method the corresponding native function starts with `Java_` and ends with `_execute` with the class name including the package name in between. Native JNI includes special native types which correspond to Java types. The types `jint`, `jobject`, `jstring`, `jintArray` correspond to the Java types `int`, `Object`, `String` and `int[]`. The function uses a Windows-specific function `LoadLibrary` which loads any library given as a string. If the library is successfully loaded the requested function is located. If the function is found the function is called with an array containing the parameters and the return value is returned to the JVM. Finally, the library is unloaded using the function `FreeLibrary`.

This implementation allows the TVM to run arbitrary native code by running the `execute` method in the `TinyNativeInterface` class.

Listing 4.8 shows a native function `print_int` which prints an integer.

```
__declspec(dllexport) int printint(int *params, int size) {
   printf("%d", params[0]);
   return 0;
}
```

Listing 4.8: The `print_int` function which prints an integer.

The `print_int` function can be called by the TVM by calling `tni.execute( "print.dll", "print_int", new int[]{5});`, which would make the TVM print `5`.

CHAPTER 5

# TESTING THE TINY VIRTUAL MACHINE

In this chapter we will describe the tests we have performed on the Tiny Virtual Machine, as well as their results. In Section 5.1 the tool Byteman, which has been used in the tests, is described. Section 5.2 describes the design behind the tests. Finally, Section 5.3 presents the results of the tests, as well as an analysis of them.

## 5.1 Byteman

To test the Tiny Virtual Machine we needed to be able to inject bitflips at various points during execution. While it would be possible to hardcode these injections it would be tedious and hard to automate. Instead of changing the actual TVM to facilitate these tests, we used a tool called Byteman [1] to automate the process. Byteman is a rule-based extension of Java, which can interrupt execution flow at specified locations and insert extra code. Byteman is executed with a target Java program, as well as a script which defines how the program should be treated.

The general rule structure is shown in Listing 5.1.

```
RULE <rule name>
CLASS <class name>
METHOD <method name>
BIND <bindings>
IF <condition>
DO <actions>
ENDRULE
```

Listing 5.1: The general rule structure of Byteman.

Byteman rules are named to differentiate them, as one Byteman file can contain multiple rules. Each rule is given a reference to the class which is to have its execution interrupted. The method should refer to the method in the class which should be the target of interruption. It is possible to specify where the interruption should happen, for example just before or just after the method. Bindings are optional, but they enable the binding of variables in the script to variables or method return values in the Java virtual machine. It is possible to make the rule conditional, which means that the extra code is only executed if the condition evaluates to true. Finally, the actions specify how Byteman should modify the execution of the Java virtual machine. This could for example be calling a method, or changing the value of a variable.

## 5.2 Test Design

This section will explain the motivation behind, as well as the structure of the tests that we have performed on the TinyVM.

The first question when designing the tests is when and where the bitflips should be introduced. The where was settled in Chapter 2, where the operand stack, local variables, program counter, as well as local heap were identified to be the vulnerable areas. While bitflips can occur in any of the frames in the tests, for the examples in this section we will assume that they occur in the current frame, as this produces the most immediate effects.

The when is more difficult, as bitflips can occur at any point in time in the real world. However, as unchecked instructions only modify the vulnerable areas the bitflip should be contained there until a checked instruction checks these for a discrepancy and, hopefully, remedies it. So for unchecked instructions injecting the bitflip between the execution of two instructions should suffice. Bitflips being injected between instructions can be seen as an approximation of an average bitflip. The time where a bitflip could cause the most damage is just after the check in a checked instruction. To illustrate this in greater detail the rule for `invokevirtual` will be used.

$$\text{invokevirtual}\;\frac{\begin{array}{c} m.instruction(pc) = \text{invokevirtual } methodId \\ lv' = ref : v_1 : \cdots : v_n \quad ref \neq null \\ lv'_r = ref_r : v_{1_r} : \cdots : v_{n_r} \quad ref_r \neq null \\ pc = pc_r \quad lv = lv_r \quad os = os_r \quad lv' = lv'_r \quad lh = lh_r \\ H' = \text{commit}(lh, H) \quad obj = H[ref] \\ MD = obj.class.constantPool.methodDesc(methodId) \\ m' = \text{lookup}(MD, obj.class) \quad n = |m'.argType| \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, (lh, lh_r), (pc, pc_r), (lv, lv_r), (ref : v_1 : \cdots : v_n : os, \\ ref_r : v_{1_r} : \cdots : v_{n_r} : os_r), \langle pc_{cp}, lv_{cp}, os_{cp} \rangle\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m', (\epsilon, \epsilon), (0,0), (lv', lv'), (\epsilon, \epsilon), \langle 0, lv', \epsilon\rangle\rangle \\ : \langle m, (lh, lh_r), (pc+1, pc_r+1), (lv, lv_r), (os, os_r), \langle pc+1, lv, os\rangle\rangle : CS\rangle \end{array}}$$

`invokevirtual` is a checked instruction, which means that the two copies of the vulnerable areas need to be identical for the instruction to complete. This check can be seen in the premise of the rule, and is repeated here:

$$pc = pc_r \quad lv = lv_r \quad os = os_r \quad lv' = lv'_r \quad lh = lh_r$$

If a bitflip occurs just after this check succeeds the instruction is executed using a corrupted value. The effects of such bitflips depend on where they occur, so they will be described for each vulnerable area below.

### 5.2.1 Bitflips in The Local Heap

The local heap is used in the premise of the rule, to update the heap:

$$H' = \text{commit}(lh, H)$$

Thus, a bitflip in the local heap can corrupt the value of a field, or cause the TVM to crash because it tries to update the value of a field that does not exist. However, only the primary copy of the local heap is used in this update. As such, bitflips in the redundant copy have no effect, as the local heaps are cleared after being committed to the heap.

### 5.2.2 Bitflips in The Operand Stack

Bitflips in the operand stack can either fall in the arguments to the instruction, or past them. Bitflips in the redundant copy of the arguments will do nothing as only their main copy is used past the check. A bitflip in the reference to the object, which the method is being invoked, on will either cause an exception or a crash, or cause the method to be invoked on the wrong object. A bitflip in the rest of the arguments will cause the method to have an invalid initial configuration, with corrupted values in both local variables, as well as the checkpoint:

Parameters on the operand stack for invokevirtual: $lv' = ref : v_1 : \cdots : v_n$
New frame configuration: $\langle m', (\epsilon,\epsilon), (0,0), (lv', lv'), (\epsilon,\epsilon), \langle 0, lv', \epsilon\rangle\rangle$

Bitflips outside the arguments also depend on whether the they occur in the redundant copy or not. Bitflips in the redundant copy will stay in the frame, and should be cleared by a rollback at the next checked instruction. To illustrate what would happen if the bitflip were to occur in the main copy, we will look at the frame's new configuration after the instruction has been executed:

$$\langle m,(lh,lh_r),(pc+1,pc_r+1),(lv,lv_r),(os,os_r),\langle pc+1,lv,os\rangle\rangle : CS\rangle$$

The $\langle pc+1,lv,os\rangle$ towards the end of the rule is the new updated checkpoint. As can be seen the main copy of the program counter, local variables, as well as operand stack are used. This means that if a bitflip occurs in the main operand stack, or indeed in the main program counter or local variables, the corrupted value will be saved in the checkpoint. This means that if a rollback occurs the bitflip will be copied into the redundant copy, which would cause execution to continue with corrupted values, probably causing a crash or a silent data corruption.

### 5.2.3 Bitflips in The Local Variables

Bitflips in the local variables would behave similarly to bitflips in operand stack, outside of the method arguments. Bitflips in the redundant copy should be cleared by a rollback, while bitflips in the main copy will be saved in the checkpoint.

### 5.2.4 Bitflips in The Program Counter

The program counter is arguably the most dangerous place for a bitflip to occur. Bitflips in the main program counter will cause the TVM to attempt to get instructions from an incorrect location. If this location is inside the memory of the program, random instructions will be executed, which might cause a rollback, should a checked instruction be encountered. If the bitflip is in the main program counter it will be saved in the checkpoint, meaning that even a rollback would not clear the error. If the program counter is outside of the programs memory the outcome would depend on the platform. Some platforms might allow reading of such memory, in which case random instructions would be executed, like above. On other platforms, like Java which the Tiny Virtual Machine is implemented on, this triggers an exception.

### 5.2.5 Test Structure

Based on the above analysis some tests will inject bitflips after the execution of instructions, while others will inject bitflips immediately after the check in checked instructions. The bitflips will all be in the form of a single flipped bit in a single value in either the main or redundant copy of the operand stack, the local variables, the program counter, or the local heap. We believe this will yield good results, some illustrating average bitflips, and some worst-case bitflips.

All tests are run on the same class, called `ArrayTest`, which uses two other classes, `Array` and `Node`. Since no compiler for TinyBytecode exists these classes have been hand-coded, in a way that hopefully matches what a compiler would have produced.

These classes implement a linked list and an insertion sort algorithm. The test method creates a linked list with 10 elements, which is then sorted, and printed to the console. For implementation of the `ArrayTest` class, see Appendix A.

While different tests will be run, most of them attempts to target as many different places in the execution as possible. To this end, each test includes several thousand runs of the program, with each attempting to inject a single bitflip. For each run during a test a random value between 1 and the number of instructions executed in a normal run, roughly 11,000, is generated. Once the total number of instructions executed surpasses this number the Byteman test script is set up to inject a bitflip at the next possibility. A thing to note, this will not produce complete results of the fault-tolerance scheme, but it will provide an indication of recovery and failure rates.

Each test produces a log file, which contains information for each individual run, as well as statistics for for the entire test. These files contain the exact configuration of where the bitflip was injected and the effect of it. Because of this, every run can be replicated to determine the exact sequence of events caused by the bitflip. When testing first began, these logs revealed mistakes in the test framework, which could then be fixed. Additionally, during tests we became aware of mistakes in the implementation of the TVM. Whenever errors were encountered, they were corrected after which all tests were rerun. The results presented in the following sections are from the final tests after error correction was made to the TVM.

## 5.3 Results

In this section the results of the tests will be presented. In Section 5.3.1 the results from the "average"-case test, where bitflips are injected after the execution of an instruction will be presented. In Section 5.3.2 the results from the worst-case tests, where bitflips were injected after frame validation will be presented. Then, in Section 5.3.3 the results from a test with a more realistic distribution of bitflips will be presented. In Section 5.3.4 bitflips in the program counter, and how to handle them, will be discussed in detail. In Section 5.3.5 tests that target specific weaknesses in the implementation will be presented and discussed. Finally, the results from a performance test will be presented in Section 5.3.6.

In most of this section results from tests will be grouped into different categories. In Table 5.1 on the next page the different categories, along with a short description, can be seen.

| | |
|---|---|
| No Bitflip | No bitflip was injected. No valid bitflip injection point was found. |
| Recovery | Bitflip was detected and corrected, terminated with expected output. |
| Masked | Bitflip was not detected, but terminated with expected output. |
| SDC | Silent Data Corruption - Bitflip was not detected, and terminated with unexpected output. |
| NullRef | Execution terminated because of an unhandled Tiny Virtual Machine Null Reference Exception. |
| NullPtr | The JVM running the Tiny Virtual Machine crashed because of an unhandled Java Null Pointer Exception. |
| OoM | Execution terminated because of an unhandled Tiny Virtual Machine Out of Memory Exception. |
| DBZ | Execution terminated because of an unhandled Tiny Virtual Machine Division by Zero Exception. |
| Invalid Field | Execution terminated because the Tiny Virtual Machine tried to write to a non existing field in an object. |
| AIOoB | The JVM running the Tiny Virtual Machine crashed because of an unhandled Java Array Index Out of Bounds Exception. |
| Other | Other Java Exceptions. The JVM running the Tiny Virtual Machine crashed. |

Table 5.1: Table of category and effect.

### 5.3.1 Test of Components After Instruction Execution

In this section the results of the 5000 runs on each of the components will be presented. The bitflips for this test was injected between execution of instructions in the TVM. The results are shown in Table 5.2 on the facing page.

| | OS | OS_R | PC | PC_R | LH | LH_R | LV | LV_R |
|---|---|---|---|---|---|---|---|---|
| No Bitflip | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Recovery | 3879 | 3917 | 543 | 3540 | 4666 | 4662 | 4706 | 4727 |
| Masked | 1095 | 1083 | 132 | 1460 | 0 | 0 | 271 | 273 |
| SDC | 0 | 0 | 96 | 0 | 0 | 0 | 0 | 0 |
| NullRef | 0 | 0 | 49 | 0 | 0 | 0 | 0 | 0 |
| NullPtr | 24 | 0 | 153 | 0 | 0 | 0 | 10 | 0 |
| OoM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DBZ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Invalid Field | 2 | 0 | 1 | 0 | 0 | 0 | 2 | 0 |
| AIOoB | 0 | 0 | 3292 | 0 | 334 | 338 | 11 | 0 |
| Other | 0 | 0 | 734 | 0 | 0 | 0 | 0 | 0 |

Table 5.2: Results from all tests where bitflips were injected after the execution of an instruction.

As has been mentioned earlier, we only wanted to target "live" values. Therefore, the amount of masked bitflips, which normally is quite high, was expected to be low. Masked bitflips appear to be most common when the bitflips occur on the operand stack. This might seem counterintuitive, since values on the operand are what is used by the program to perform all calculations. If one of these values are flipped, the execution of the program might be expected to deviate from the normal execution. One possible explanation for these masked bitflips could be branches. If, for example, a branch is activated when the values 7 and 1 are different, the values will still be different if a bit is flipped in either, causing the bitflip to be masked. Bitflips in the local variables also have a high mask rate, however, this is to be expected, as corrupted values there might never be read again, or might be overwritten with a correct value. The most problematic component seems to be the program counter, where both the recovery rate and the mask rate are low. Bitflips in the program counter account for roughly 66.5% of crashes in these tests. The overall recovery rate of the Tiny Virtual Machine is shown in Table 5.3.

| | Recovery | Masked | SDC | Crash |
|---|---|---|---|---|
| Percentage | 76.6% | 10.79% | 0.24% | 12.37% |

Table 5.3: Results from testing components after instruction execution in percentages.

The fault-tolerance scheme was designed for this case, and therefore the recovery rate is supposed to be high. This is not a comprehensive test, but it does indicate how the fault-tolerance scheme performs. The results here represent the average case, where the fault-tolerance should perform well. A more interesting scenario is presented in the following section, where we will test how the fault-tolerance scheme performs in the worst case.

## 5.3.2 Test of Components After Frame Validation

The bitflips for this test was injected after frame validation in the checked instructions. Like the tests in the previous section, all tests here were run 5000 times on each component. The results are shown in Table 5.4.

| | OS | OS_R | PC | PC_R | LH | LH_R | LV | LV_R |
|---|---|---|---|---|---|---|---|---|
| No Bitflip | 1 | 1 | 0 | 0 | 330 | 374 | 0 | 0 |
| Recovery | 336 | 405 | 436 | 3258 | 0 | 0 | 3313 | 3553 |
| Masked | 957 | 4594 | 1470 | 1742 | 3 | 4626 | 1458 | 1467 |
| SDC | 2804 | 0 | 102 | 0 | 2857 | 0 | 35 | 0 |
| NullRef | 902 | 0 | 8 | 0 | 6 | 0 | 179 | 0 |
| NullPtr | 0 | 0 | 95 | 0 | 1028 | 0 | 8 | 0 |
| OoM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DBZ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Invalid Field | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AIOoB | 0 | 0 | 2304 | 0 | 776 | 0 | 7 | 0 |
| Other | 0 | 0 | 585 | 0 | 0 | 0 | 0 | 0 |

Table 5.4: Results from all tests where bitflips were injected after frame validation.

As can be seen, the recovery rate of is a lot lower in this test than the previous, which is a result of this test injecting bitflips at the most vulnerable place in the TVM. The recovery rate as well as the masked rate of the local variables are relatively high compared to the other components. This might be caused by most of the bitflips being injected in a frame which is not the topmost one, which simply causes a rollback to a valid checkpoint once the affected frame is reached. And of those bitflips that do occur in the topmost frame, roughly half will be injected during a return instruction, which causes the frame to be destroyed, effectively masking the bitflip. The only vulnerable areas during a return instruction are the topmost value on the operand stack, which is returned, and the local heap, which is committed to the heap.

The `invokevirtual` instruction is more vulnerable, as the operand stack is used to create the new frame, the local heap is committed, and the operand stack, program counter and local variables are all saved to the checkpoint. This means that bitflips injected in the topmost frame during an `invokevirtual` instruction are rarely masked or recovered from, and often cause crashes or even silent data corruptions.

The recovery and failure rates of the Tiny Virtual Machine for this test have been summed up in Table 5.5 on the next page.

| | Recovery | Masked | SDC | Crash |
|---|---|---|---|---|
| Percentage | 28.25% | 40.79% | 14.50% | 14.73% |

Table 5.5: Results from frame validation tests in percentage.

### 5.3.3 Test of Random Components

In the previous sections the results of testing each component 5000 times on both conditions were presented. To simulate a case which would look more like what could happen in the real world, we have performed another test. This test is run 10000 times where the bitflip can happen after either instruction execution or frame validation. Furthermore, the target component of the test is chosen at random during execution. The results of this test is shown in Table 5.6.

| | OS | OS_R | PC | PC_R | LH | LH_R | LV | LV_R |
|---|---|---|---|---|---|---|---|---|
| No Bitflip | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Recovery | 900 | 1024 | 215 | 1272 | 41 | 60 | 1745 | 1865 |
| Masked | 309 | 350 | 47 | 526 | 0 | 10 | 2 | 2 |
| SDC | 75 | 0 | 32 | 0 | 6 | 0 | 0 | 0 |
| NullRef | 57 | 0 | 17 | 0 | 0 | 0 | 1 | 0 |
| NullPtr | 6 | 0 | 63 | 0 | 2 | 0 | 0 | 0 |
| OoM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DBZ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Invalid Field | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AIOoB | 0 | 0 | 1110 | 0 | 2 | 0 | 0 | 0 |
| Other | 0 | 0 | 260 | 0 | 0 | 0 | 0 | 0 |
| Total | 1348 | 1374 | 1744 | 1798 | 51 | 70 | 1748 | 1867 |

Table 5.6: Results from the random test.

A quick examination of the results show, that bitflips in the redundant copies are either recovered or masked. The test program does not do much manipulation of the heap, which causes the local heap to be empty most of the time. If an attempt is made to inject a bitflip in the local heap while it is empty, the test is set up to attempt to inject a fault in a different component. Because of this, bitflips in the local heaps are rare.

These results also reaffirm that the program counter is the most troublesome to control of all the components in our fault model. Most of the crashes experienced are due to bitflips in the program counter. In Section 5.3.4 we will discuss a way to control the program counter.

### 5.3.4 Handling the Program Counter

As has been made clear, bitflips targeting the program counter are difficult to handle. This is mainly because the Tiny Virtual Machine is implemented in Java, which does not allow reading arbitrary bytes of memory. Because of this the code in the TVM is implemented in a Java array, meaning that bitflips in the program counter has a large chance of producing an index out of bounds exception. Therefore, we have extended the fault-tolerance scheme to, hopefully, better handle this type of bitflip. This does further reduce performance, which will be discussed in Section 5.3.6 on page 62.

Bitflips in the program counter can make the TVM crash, and therefore we need to validate the program counters before the execution of each instruction. The frames on the call stack contain two program counters, one primary and one redundant. These values should always be kept in registers, and because of this, a simple comparison of the two before the execution of the next instruction should be relatively cheap. A bitflip can of course still hit the program counter after this check is performed, and before the program counter is used to look up the instruction. If this happens, then the scenario will be as the previous cases with bitflips injecting after instruction execution and frame validation. Because of this, we have not made any tests to document this outcome. Instead, we made two tests with 5000 runs each, with the program counter check enabled. One of these test injected bitflips after instruction execution, and the other after frame validation to document the effect of the program counter check.

By including this program counter check, the recovery rate of 5000 runs, where bitflips were injected after instruction execution, has improved to 100%. Substituting these results in for the old results without the program counter check changes the overall results to what can be seen in Table 5.7.

| | Recovery | Masked | SDC | Crash |
|---|---|---|---|---|
| Percentage | 87.74% | 10.46% | 0% | 1.80% |

Table 5.7: Percentages of outcomes after instruction execution with program counter check.

The recovery rate has been increased by 11%, while lowering the remaining. The masked rate remains almost the same, with a small decrease of 0.3%. In the results for instruction execution the risk of a run ending in a silent data corruption was very small, 0.24%. With the program counter check this is reduced to 0%, for the tests we made. The amount of crashes have also been reduced greatly by the program counter check, from 12.37% to 1.8%.

When injecting bitflips after frame validation, the program counter check did not have as high an effect as in the previous case. Without program counter check the amount of recoveries was 436 out of 5000 runs. With the program counter check this is increased to

3160 and the masked cases remains approximately the same as before. This results in a decrease of crashes caused by bitflips in the program counter by almost 90%. Substituting these results in for the old results changes the overall results for bitflips injected after frame validation to what is shown in Table 5.8.

| | Recovery | Masked | SDC | Crash |
|---|---|---|---|---|
| Percentage | 35.06% | 40.76% | 14.36% | 8.09% |

Table 5.8: Percentages of outcomes after frame validation with program counter check.

The overall recovery rate is increased by roughly 7% which comes from the decrease in crashes. The program counter check does not reduce the amount of silent data corruptions in this case.

### 5.3.5 Qualitative Tests

In this section we will perform tests of certain cases, where we have identified possible problems. We want to explore what happens when we inject bitflips in these cases on our implementation. Each case deals with a specific value in a specific component. For each case a test will be performed, in which each run will flip a different bit in the target value. Each test will include a run for each bit in the target value, to ensure that all possibilities are explored.

#### Operand Stack

When an `invokevirtual` or return instruction is executed the operand stack contains the arguments for the instruction. For `invokevirtual` instructions the first of these arguments is the object reference, which the method should be invoked on, while for return the first, and only, argument is the return value. Should a bitflip occur in any of these elements after the frame validation of an `invokevirtual` or return call has succeeded, then the TinyVM will attempt to execute the instruction with corrupted values.

As an example of this we ran a test which flipped a bit in the argument to the method which is supposed to add an 8 to the linked list. Flipping the 1 bit in this argument produced the result 0123456799, where the 8 has been changed to a 9. Flipping the 8 bit, on the other hand, produced the results 0012345679. Flipping other bits in this argument has similar effects, in all cases causing the 8 to be corrupted before the list is sorted. The same test with bitflips injected after instruction execution caused a recovery regardless of which bit was flipped.

In another test, a bitflip was injected in the return value of the method `Array.GetVal` which is called when sorting the linked list. This method returns a value of an element in the linked list, which in this case is used to swap values with another element. When the bitflip is injected after frame validation the corrupted value is returned, and then written to one of the other elements' value when the swap is performed. This causes a silent data corruption, regardless of which bit is swapped. If the bitflip is injected after instruction execution the TVM recovers in all cases.

### Program Counter

Bitflips in the program counter generally produce one of two outcomes. If the program counter is still within the instruction array the outcome is undefined, as random values will be interpreted as instructions. If the program counter is larger than the size of the instruction array the JVM crashes because of an array index out of bounds exception. An interesting case to examine in depth for a bitflip in the program counter, would be when the bitflip is masked. An example of this can be made in the method `GetVal` from the `Array` class as shown in Listing 5.2.

```
 1 "load REF 0",
 2 "getfield 6",
 3 "store REF 3",
 4 "load INT 1",
 5 "push INT 0",
 6 "if EQUALS 68",
 7 "push INT 0",
 8 "store INT 2",
 9 "load REF 3",
10 "getfield 22",
11 "store REF 3",
12 "load INT 2",
13 "push INT 1",
14 "comp ADD",
15 "store INT 2",
16 "load INT 1",
17 "load INT 2",
18 "if NEQUALS 31",
19 "load REF 3",
20 "getfield 24",
21 "return INT"
```

Listing 5.2: getVal method from Array class.

In this method, the program counter is 64 after the 17th instruction. Bitflips in this value behave largely as expected. Flipping bit 0, 3, and 4 cause the TVM to recover and produce the correct output. Almost all other bits cause the JVM to crash because of unhandled exceptions. The only exception is bit 6, which causes the bitflip to be masked, meaning that the program produced the correct value even though the bitflip was never caught.

Upon review of the TVM configuration at the time of the bitflip, we discovered the reason for the masked bitflip. For a bitflip in the program counter to be masked in this manner some special conditions must hold. The method must not change any of its parameters, which are stored in local variables, during its execution. Furthermore, the program counter's binary representation should only have one 1-bit, such as 64, which is $0b1000000$. If that particular bit is flipped, then the method is effectively restarted, with whatever was previously on the operand stack remaining untouched at the stacks bottom. This has another potential side effect. In a class file the maximum operand stack size is declared. Since there is garbage on the operand stack, this maximum is likely to be exceeded which will result in an unhandled exception. If this does not happen, then there is yet another condition that needs to be fulfilled. A branch must be followed before any checked instruction is encountered, as this sets the two program counter components to the same value. This happens because, even though the two program counter components are different at this point, the two operand stacks are identical, meaning that both the main and redundant component follow the branch in the same way. If a checked instruction comes first, then the bitflip will be recovered.

### Local Variables

An interesting case for the local variables would be where they are used as counters in a loop. A case of this could be like the snippet from a method shown in Listing 5.3.

```
"push INT 0",
"store INT 2",
"push INT 9",
"store INT 3",
"load INT 3",
"load INT 2",
"if EQUALS 105",
"push INT 1",
"load REF 1",
"getfield 14",
"comp ADD",
"load REF 1",
"putfield 14",
"push INT 1",
"load INT 2",
"comp ADD",
"store INT 2",
"goto 56"
```

Listing 5.3: A looping construct from an example method.

This method increments a field of an object 9 times. The first instruction pushes 0 onto the operand stack, and then stores it into the local variables at index 2. This variable is used as the counter in the loop, which continues until the counter hits 9. If the local variable at index 2 is flipped at any point during this loop to a value greater than 9, the TVM will simply keep looping. If the bitflip causes the value to be less than, or equal to 9 the loop eventually terminates, after which the TVM recovers.

If the redundant copy is the target, then the result will be a recovery.

### 5.3.6 Performance

In this section we will present the performance tests performed on the Tiny Virtual Machine with and without fault-tolerance.

In each performance test, the TVM was set to execute the class `ArrayTest` 5000 times, while logging execution time and memory usage in a file. The execution time was measured from immediately before executing the first instruction, until immediately after executing the last. The memory was measured with the code *runtime.totalMemory*() – *runtime.freeMemory*() after the last instruction, the result of which is the total memory used by the TVM.

Three performance tests were run: Two on the fault-tolerant implementation, one with the default configuration, and one with a program counter check mentioned in Section 5.3.4. For the final test we used a version of the TinyVM, which had been modified to not include any of our fault-tolerance measures.

The average values from each of these tests can be seen in Table 5.9.

| | TinyVM | TinyVM + PC check | TinyVM - fault-tolerance |
|---|---|---|---|
| Time used | 0.0414 seconds | 0.0422 seconds | 0.0275 seconds |
| Memory usage | 2.600 MB | 2.600 MB | 1.960 MB |

Table 5.9: Performance test results.

These results show that for `ArrayTest`, our fault-tolerance increases execution time to $0.0414/0.0275 \approx 150.5\%$, and memory usage to $2.600/1.960 \approx 132.7\%$. Checking the program counters before each instruction increases run time to $0.0422/0.0414 \approx 102.0\%$.

It should be noted that since the TinyVM is implemented in Java its performance is quite poor compared to a native implementation. Therefore, these results should only be seen as an guideline for the performance cost of our fault-tolerance scheme.

## 5.4 Test Summary

When bitflips occur between instructions they are either masked or recovered by the Tiny Virtual Machine 87.39% of the time. For worst-case bitflips which are injected right after frame validation this number drops to 69.04%.

The most critical data component in the TVM are the program counters. To counter this we ran tests with a specific check for the program counters every instruction. In these, bitflips injected between instructions were masked or recovered 98.2% of the time. For bitflips injected after frame validation the number was 75.82%.

For the performance tests we created a special version of the TVM with all fault-tolerance removed. Comparing this to the ordinary TVM indicated that our fault-tolerance increases execution time to 150.5%, and memory usage to 132.7%. The additional program counter check increased runtime to 102% as compared to the ordinary TVM, without affecting memory usage.

CHAPTER 6

# RELATED WORK

In this chapter related work on fault-tolerance will be described. Related work described in the previous report [16] will not be repeated here. Section 6.1 lists a few works related to fault-tolerance in Java. Section 6.2 describes two methods of detecting faults based on symptoms.

## 6.1 Reliability in Java

There exists several works related to reliability in Java. While most are not directly related to fault-tolerance, we thought them worth to include here. Instead of dedicating a section to each of these, this section will include a brief description of the most relevant, as well as how they relate to our work.

### 6.1.1 Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) [2] is not directly related to fault-tolerance. RTJS sets out to provide a Java-like platform that lets programmers correctly reason about the temporal behaviour of executing software. Their motivation is to enable the creation of real-time systems using an object-oriented language, which is similar to our motivation.

### 6.1.2 Safety-Critical Java

Safety Critical Java (SCJ) [8] is based on RTSJ, and is designed to enable creating safety-critical applications using Java. SCJ includes guidelines for making applications for SCJ, as well as for creating a conforming implementation of the SCJ platform itself. SCJ does not focus specifically on single-event upsets, but instead on defining how the execution of an application is impacted by faults. One of the goals of SCJ is to ease the certification process of creating safety-critical applications, for example for the aerospace sector.

SCJ is a very large and complex specification, which requires programmers to adapt a very specific program structure. We have instead focused on a simpler system, where the platform itself attempts to handle all faults, and programmer restrictions are minimal.

### 6.1.3 Jasmin/Jasper

Jasmin is an assembler for the Java virtual machine, which converts ASCII encoded Java classes to binary Java class files. Jasmin is the oldest Java assembler and it is used in many compilers for Java [9].

Jasper is Java disassembler capable of generating ASCII files from binary Java class files [14]. Although Jasper supports the Jasmin format, development of the two was done separately.

It would have been possible to use these tools to create development tools for Tiny-Bytecode, such as a compiler. However, we chose to focus on the actual execution of programs, leaving such tools for future work.

### 6.1.4 Fault-Tolerant JVM

Napper *et al* [10] created a fault-tolerant Java virtual machine, by converting it into a deterministic state machine. In this approach two JVM's are run side by side, one primary, and one secondary. The primary JVM logs each computational step and sends it to the secondary. If the primary JVM fails, the secondary can perform recovery based on the computational log from the primary. This is a similar approach to the first patent filed for a fault-tolerant JVM by Holiday [5]. In this approach, objects are checkpointed and sent to the secondary JVM and in case the primary fails, the secondary has the same objects allocated and can continue execution. Our approach to create a fault-tolerant virtual machine is different because we do not require a primary-backup architecture, but instead try to incorporate a fault-tolerant scheme within a single running virtual machine.

## 6.2 Symptom-based Error Detection

Symptom-based error detection is one branch of error detection which attempts to handle errors, such as single-event upsets, while incurring a minimal performance overhead. The way this is generally done is by doing as little extra work as possible, while waiting for a symptom of an error to appear, and then handling the error at that point. Double modular redundancy is, arguably, a type of symptom-based error detection, where a discrepancy between the two modules is the symptom that is being monitored. However, symptom-based error detection schemes normally have much less performance overhead than double modular redundancy.

### 6.2.1 Perturbation-based Fault Screening

Racunas *et al.* [13] proposed a method of detecting faults in software based on so-called perturbations. A perturbation in this context means a "departure from established program behaviour". A perturbation can be a symptom of a fault, however, they can also be caused by natural changes in program behaviour, for example because of program input. Take for instance a static instruction, which has produced a value between 0 and 16, the last 1000 times it has been executed. If this value history was used as the established program behaviour, an output of 50 from the same static instruction might be seen as a perturbation.

To detect perturbations various fault screeners are used [13]. A fault screener deploys some strategy to determine if an output of a given static instruction is a perturbation or not. Whenever such a perturbation is detected some action is performed to attempt to remedy the underlying fault. Since perturbations can be natural in a correctly executing program, screeners should attempt to minimize the rate of these false-positives, to minimize performance overhead.

#### Screeners

This section will describe a few of the fault screeners used by Racunas *et al.*

**Extended History Screener**
This screener keeps a history of 64 recent unique values, as well as 64 unique delta values between successive values, per static instructions. To manage instructions that generate too many unique values for the screener to handle, static instructions that generate more than one perturbation for every 100 instances are ignored.

**Dynamic Range Screener**
This screener builds a representation of the valid value space by dividing it into resizeable segments. A segment is simply a range from one value to another. If a new value is not contained in any of these segments a perturbation is triggered, and the new value is added to a segment in such a way as to minimize the total size of the segments.

**Invariance-based Screener**
This screener maintains a bitmask associated with each static instruction. Additionally, the most recent value generated by the instruction is stored. Then, as new values are generated, the bitmask is updated to indicate all bits that have been invariant over all the values. A delta bitmask is also maintained, and this is modified

in a similar way, only using the delta between values, instead of the values themselves. If a value triggers a change in both of these bitmasks, a perturbation is triggered.

Whenever a perturbation is registered it must be handled somehow. Racunas *et al.* use a flush of the processor pipeline to this end, and this is a relatively cheap and effective remedy. Another possibility would be to use a checkpoint recovery system.

### 6.2.2 ReStore

Wang *et al.* [17] proposed the ReStore processor architecture, which uses symptom-based fault-detection along which hardware supported checkpoints to provide low-cost fault-tolerance. These checkpoints consists of a copy of the register file, as well as data to enable restoring the memory to the state it had when the checkpoint was created. This checkpoint is updated once every 10–1.000 instructions, depending on configuration. The actual symptom detectors attempt to detect symptoms of faults, such as bitflips, and are located in various parts of the processor pipeline. In general, a symptom will cause the system to perform a rollback to the checkpoint, hopefully clearing the erroneous data from the system. Wang *et al.* describe 3 different symptom detectors, though more could be imagined.

**Exceptions**
This detector simply treats memory access and alignment exceptions as symptoms of faults. If the same exception does not reappear during re-execution it was most likely caused by a soft-error. If it does reappear the exception is handled normally.

**Branch misprediction**
This detector utilizes the branch predictors already present in many processors. In addition to predicting the outcomes of branches, these predictors also produce a confidence value, which denotes how accurate the prediction is. The detector treats branch mispredictions with high confidence as symptoms. How high the confidence needs to be for the misprediction to be considered a symptom can then be configured, based on performance requirements.

**Cache miss**
Some faults might change a memory address without changing it enough for an exception to occur. This detector is based on the observation that programs use the processor cache for a large part of its memory accesses. Therefore a cache miss might be a symptom of a fault. To reduce the frequency of false positive symptoms reported by this detector a confidence value could be implemented and used in a similar way as the previous detector.

## 6.3 Transactional Memory

In the approach proposed in Section 2.3, a local heap was used to keep track of changes made to main memory and only committed to main memory when deemed safe. This concept is fairly similar to a concept known as *transactional memory*. Transactional memory was proposed by Herlihy and Moss [4] as an alternative to the usual lock-based approach with mutual exclusion. Herlihy and Moss add a small transactional primary cache to a processor besides its regular non-transactional primary cache. These caches are exclusive, an entry may reside in one of them, but not in both. For the transactional memory approach to work, Herlihy and Moss introduced *transactional tags*, added transactional instructions to the processor, and modified the caching protocol. Entries residing in the small transactional cache can have one of the following tags: `EMPTY`, `NORMAL`, `XCOMMIT`, or `XABORT`. When a transactional operation is encountered two entries are cached, one with the `XCOMMIT` tag which means to discard this value on commit, and one with the `XABORT` tag which means to discard this value on abort. Changes to the cached entry are made to the one with the `XABORT` tag. When a transaction commits, cache lines with the tag `XCOMMIT` are changed to `EMPTY`, since the transaction committed this value is no longer needed. Cache lines with tag `XABORT` are changed to `NORMAL`, which means that they hold committed data. When a transaction aborts it is the exact opposite, since the modified data is not needed, but instead the initial data.

Transactional memory proposed by Herlihy and Moss makes use of modified hardware for the approach. The approach used in this report, make use of no such hardware, and relies solely on software. This is more similar to another approach to transactional memory called `software transactional memory` proposed by Shavit and Touitou [15]. This approach extends the original transactional memory, but simulated in software alone.

# CHAPTER 7

# RECAPITULATION

In this chapter, we will discuss the findings made throughout this project in Section 7.1, and in Section 7.2 elaborate on elements for future work.

## 7.1 Conclusion

In this report we modified the semantic rules of TinyBytecode to include a fault-tolerance scheme. This scheme includes duplicate components in frames, extended frames with checkpoints, and the concept of checked instructions. TinyBytecode was first designed in a previous semester project, and in addition to the above mentioned modifications the language's semantics were extended with rules for exceptions, faults, virtual machine exceptions and fault recovery. Furthermore, native methods, which are often left out in research about fault-tolerance, were added to TinyBytecode. A virtual machine was implemented for TinyBytecode to test how the designed scheme performed.

A total of 105.000 runs were made to test recovery and failure rates as well as performance. Tests were performed in two distinct cases: one with bitflips injected after instruction execution, and one with bitflips injected after frame validation. Results showed that in the case of injecting bitflips after instruction execution, the Tiny Virtual Machine recovered in 76.6% of the runs. If the bitflips were injected after frame validation, the Tiny Virtual Machine recovered in 28.25% of the runs. Even if the bitflips were injected in the most vulnerable place of the TVM, a degree of fault-tolerance was still provided.

The scheme was designed to handle faults in the operand stack, the program counter, the local heap, and the local variables. However, faults in the program counter proved problematic, and therefore additional measures were implemented to detect and correct bitflips in the program counter. Furthermore, the fault model chosen was at most one bitflip throughout a program execution. The scheme handles this case, but we estimate that the design of the scheme should be resilient enough to handle more than one bitflip at a time.

In a final test the performance of the TVM was compared to that of another version of the TVM which had had all fault-tolerance measures removed. This test showed that the cost of the fault-tolerance scheme was an increase in runtime by 50.5% and an increase in memory consumption by 32.7%.

## 7.2 Future Work

This section will discuss what lies ahead for the development of the TVM.

### Performance

One of the main things that needs to be done for the TinyVM in the future, is to improve the performance. Currently, each frame contains redundant copies of almost every component. Along with this, each frame also contains a checkpoint, which contains nearly the same information as a frame. Because of this, a frame uses almost three times the memory that a regular frame with no redundancy would. The only component that a frame has but a checkpoint does not is the local heap. However, the local heap is cleared by every checked instruction, and since invokevirtual is checked, only the local heap in the topmost frame will ever have any contents. Therefore, all but the topmost frame could be reduced to only contain a checkpoint. This checkpoint would then be used to restore the primary and redundant components of the frame when it became "live" again. This would save a sizeable amount of memory, as less redundant data would be necessary. This change could, however, increase execution times, as the checkpoint would have to be restored whenever a method returned.

Various things could be done to improve the execution time of the Tiny Virtual Machine. The first step would be to create an implementation in a more low-level language, to avoid the overhead encountered from running a virtual machine on the JVM. As for the actual architecture of the TVM one option could be to use multi-level checkpoints and local heap. With this modification the local heap would be global like the normal heap, and checkpoints would only be in a fraction of the frames. Every invoke and return would then not need to cause a checkpoint update and local heap commit. This would allow smaller methods to avoid the overhead incurred by creating a checkpoint, while, theoretically, only adversely affecting performance when faults do occur. How often checkpoints should be updated could then either be configured by developers, or decided dynamically by the TVM.

While performance is important, especially for embedded systems, we chose to focus on the task of implementing fault-tolerance in a language that could easily be the target of a Java compiler.

### Native Methods

Native methods have been discussed in the report, and are, as far as we know, not commonly included in fault-tolerance work. We identified potential problems by transferring control out of the TinyVM, and made restrictions to native methods for this reason. In Java native methods are able to modify the state of the entire JVM, including the heap, operand stack, and even private variables. This approach was problematic for us, as we wanted to create semantics for our invokenative instruction, and allowing it to change every value in every frame would have been difficult. Instead we went with a very limited invokenative, which is only able to receive a number of arguments, and return one integer. Allowing native methods to modify the heap would make them much more useful. A native method could be allowed to modify an object if they were passed a reference to it, or create an object if they returned a reference to it.

Native methods are a powerful tool that would probably be important in a real implementation of the Tiny Virtual Machine. However, in this project we chose to focus on the fault-tolerance, leaving the language as simple as possible. Native methods were included only to allow the TVM to perform input and output.

### Transactional Memory

In our fault-tolerance scheme, we created the local heap to prevent faults from propagating to the heap. We created this solution and found out that a similar technique, transactional memory had been developed. As future work, it would be a good idea to see whether the software transactional memory could improve the performance as compared to our own solution.

### Garbage Collection

One problem, which is common to fault-tolerance work, is the exclusion of garbage collection. In our implementation, objects are statically allocated and never garbage collected. Potential problems can arise, from never clearing the heap, but doing so is problematic for fault-tolerance. The garbage collection in Java happens asynchronously and provides no guarantees if when it runs. Therefore, designing a fault-tolerant garbage collection will be of high interest.

## Development Tools

Lastly, the TinyVM is not useful, unless a tool is created which can automate the process of writing programs. Currently, all test programs are written in json format, and translated to binary files using an encoder. This is not optimal, and development of a compiler for TinyBytecode is an essential part of the future work. Furthermore, a bytecode verifier for TinyBytecode would also be a very convenient tool to have for any programmers who choose to use use the TinyVM.

# BIBLIOGRAPHY

[1] Byteman project. `http://byteman.jboss.org/`. Last Visited on 2014-05-12.

[2] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[3] René Rydhof Hansen. *Flow Logic for Language-Based Safety and Security*. PhD thesis, Technical University of Denmark, 2005.

[4] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 1993)*, pages 289–300, 1993.

[5] M.R. Holiday. Fault-tolerant Java virtual machine, 2002. US Patent 6,421,739.

[6] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010.

[7] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® virtual machine specification. `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`, February 2013. Last Visited on 2013-11-26.

[8] D Locke, BS Andersen, B Brosgol, M Fulton, T Henties, JJ Hunt, JO Nielsen, K Nilsen, M Schoeberl, J Vitek, and A Wellings. Safety critical java specification. *The Open Group, UK*, 2013.

[9] Jon Meyer. Jasmin assembler for Java. `http://jasmin.sourceforge.net`, 2004.

[10] Jeff Napper, Lorenzo Alvisi, and Harrick Vin. A fault-tolerant Java virtual machine. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2003.

[11] Oracle. Java™ platform standard ed. 7: Class integer. `http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html`, 2014. Last Visited on 2014-05-28.

[12] Oracle. The Java™ tutorials: Restrictions on generics. `http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html`, 2014. Last Visited on 2014-05-28.

[13] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S. Mukherjee. Perturbation-based fault screening. *High Performance Computer Architecture*, pages 169–180, 2007.

[14] Chris Rathman. Jasper disassembler for Java. `http://www.angelfire.com/tx4/cus/jasper/`, 2000.

[15] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[16] Emil Čustić, Martin Fjordvald, Martin Sørensen, Rune Hansen, and Sebastian Lybæk. Fault-tolerance: An exploration of software-based solutions. AUB Report Library, 2013. Student Report, 9th Semester.

[17] Nicholas J. Wang and Sanjay J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

# Appendix A

# Array Implementation

```
{
    "constant_pool": [
        {
            "id": 0,
            "tag": 1,
            "class_name": 1
        },
        {
            "id": 1,
            "tag": 4,
            "bytes": "Array"
        },
        {
            "id": 2,
            "tag": 1,
            "class_name": 3
        },
        {
            "id": 3,
            "tag": 4,
            "bytes": "Object"
        },
        {
            "id": 4,
            "tag": 1,
            "class_name": 5
        },
        {
            "id": 5,
            "tag": 4,
            "bytes": "Node"
        },
        {
            "id": 6,
            "tag": 2,
            "class_name": 1,
            "field_name": 7,
            "field_type": 8
        },
        {
            "id": 7,
            "tag": 4,
            "bytes": "start"
        },
        {
            "id": 8,
            "tag": 5,
            "type": "l",
            "class_name": 5
        },
        {
            "id": 9,
            "tag": 2,
            "class_name": 1,
            "field_name": 10,
            "field_type": 11
        },
        {
            "id": 10,
            "tag": 4,
            "bytes": "length"
        },
        {
            "id": 11,
            "tag": 5,
            "type": "i"
        },
        {
            "id": 12,
            "tag": 3,
            "class_name": 3,
            "method_name": 13,
            "ret_type": 14
        },
        {
            "id": 13,
            "tag": 4,
            "length": 6,
            "bytes": "<init>"
        },
        {
            "id": 14,
            "tag": 5,
            "type": "v"
        },
        {
            "id": 15,
            "tag": 3,
            "class_name": 1,
            "method_name": 13,
            "arg_types": [11],
            "ret_type": 14
        },
        {
            "id": 16,
            "tag": 3,
            "class_name": 1,
            "method_name": 17,
            "arg_types": [11],
            "ret_type": 11
        },
        {
            "id": 17,
            "tag": 4,
            "bytes": "GetVal"
        },
        {
            "id": 18,
            "tag": 3,
            "class_name": 1,
            "method_name": 19,
            "arg_types": [11, 11],
            "ret_type": 14
        },
        {
            "id": 19,
            "tag": 4,
            "bytes": "SetVal"
        },
        {
            "id": 20,
            "tag": 3,
            "class_name": 1,
            "method_name": 21,
            "arg_types": [11],
            "ret_type": 14
        },
        {
            "id": 21,
            "tag": 4,
            "bytes": "AddVal"
        },
        {
            "id": 22,
            "tag": 2,
            "class_name": 4,
            "field_name": 23,
            "field_type": 8
        },
        {
            "id": 23,
            "tag": 4,
            "bytes": "next"
        },
        {
            "id": 24,
            "tag": 2,
            "class_name": 4,
            "field_name": 25,
            "field_type": 11
        },
        {
            "id": 25,
            "tag": 4,
            "bytes": "value"
        }
    ],
    "this": 0,
    "super": 2,
    "methods": [
        {
            "method_descriptor": 15,
            "max_stack": 2,
            "max_locals": 3,
            "code": [
                "load REF 0",
                "invokevirtual 12",
                "new 4",
                "store REF 2",
                "load INT 1",
                "load REF 2",
                "putfield 24",
                "push INT 1",
                "load REF 0",
                "putfield 9",
                "load REF 2",
                "load REF 0",
                "putfield 6",
                "return VOID"
            ]
        },
        {
            "method_descriptor": 16,
            "max_stack": 2,
            "max_locals": 4,
            "code": [
                "load REF 0",
                "getfield 6",
                "store REF 3",
                "load INT 1",
                "push INT 0",
                "if EQUALS 68",
                "push INT 0",
                "store INT 2",
                "load REF 3",
                "getfield 22",
                "store REF 3",
                "load INT 2",
                "push INT 1",
                "comp ADD",
                "store INT 2",
                "load INT 1",
                "load INT 2",
                "if NEQUALS 31",
                "load REF 3",
                "getfield 24",
                "return INT"
            ]
        },
        {
            "method_descriptor": 18,
            "max_stack": 3,
            "max_locals": 5,
            "code": [
                "load REF 0",
                "getfield 6",
                "store REF 4",
                "load INT 1",
                "push INT 0",
                "if EQUALS 68",
                "push INT 0",
                "store INT 3",
                "load REF 4",
                "getfield 22",
                "store REF 4",
                "load INT 3",
                "push INT 1",
                "comp ADD",
                "store INT 3",
                "load INT 1",
                "load INT 3",
                "if NEQUALS 31",
                "load INT 2",
                "load REF 4",
                "putfield 24",
                "return VOID"
            ]
        },
        {
            "method_descriptor": 20,
            "max_stack": 4,
            "max_locals": 4,
            "code": [
                "load REF 0",
                "getfield 9",
                "push INT 1",
                "comp ADD",
                "load REF 0",
                "putfield 9",
                "load REF 0",
                "getfield 6",
                "store REF 2",
                "load REF 2",
                "getfield 22",
                "push REF 0",
                "if EQUALS 60",
                "load REF 2",
                "getfield 22",
                "store REF 2",
                "GOTO 31",
                "new 4",
                "store REF 3",
                "load INT 1",
                "load REF 3",
                "putfield 24",
                "load REF 3",
                "load REF 2",
                "putfield 22",
                "return VOID"
            ]
        }
    ]
}
```