

Majchrzak, Tim A.; Kuchen, Herbert

Working Paper

Muggl: The Muenster generator of glass-box test cases

ERCIS Working Paper, No. 10

Provided in Cooperation with:

University of Münster, European Research Center for Information Systems (ERCIS)

Suggested Citation: Majchrzak, Tim A.; Kuchen, Herbert (2011) : Muggl: The Muenster generator of glass-box test cases, ERCIS Working Paper, No. 10, Westfälische Wilhelms-Universität Münster, European Research Center for Information Systems (ERCIS), Münster

This Version is available at:

<https://hdl.handle.net/10419/58417>

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.



European
Research
Center for
Information
Systems

Working Paper No. 10

Majchrzak, T. A. ■
Kuchen, H. ■

Muggl:
**The Muenster Generator of
Glass-box Test Cases**



Working Papers

ERCIS – European Research Center for Information Systems

Editors: J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein,
H. Kuchen, U. Müller-Funk, U. W. Thonemann, G. Vossen

Working Paper No. 10

Muggl: The Muenster Generator of Glass-box Test Cases

Tim A. Majchrzak, Herbert Kuchen

ISSN 1614-7448

cite as: Tim A. Majchrzak, Herbert Kuchen: Muggl: The Muenster Generator of Glass-box Test Cases. In: Working Papers, European Research Center for Information Systems, number 10. Eds.: Becker, J. et al., Münster 2011.

Contents

Working Paper Sketch	1
1 Introduction	3
2 Related Work	3
3 The Java Virtual Machine	4
3.1 Basic Concepts	4
3.2 Data Types	5
3.3 Architecture and Memory Abstraction	6
3.4 Class Files	7
3.5 Execution Unit	9
3.6 Instruction Set	9
4 Muggl Basics	13
4.1 Execution Principles	13
4.2 Symbolic Execution	17
4.3 Constraint Solving	21
4.4 Test Case Generation	22
4.5 Architecture	23
4.6 Execution example	25
5 Distinct Features	25
5.1 Iterative-Deepening Depth-First Search	26
5.2 Step-by-Step GUI	28
5.3 Configurability	31
5.4 Class File Inspection	34
5.5 Control-Flow and Data-Flow Coverage	34
5.6 Test Case Elimination	37
5.7 Native Wrapper	37
5.8 Dynamic Optimization	38
5.9 Array Generation	39
5.10 Java Class File Representation by Java Classes	40
6 Experimental Evaluation	43
6.1 Tested Examples	43
6.2 Qualitative Results	44
6.3 Quantitative Results	44
7 Ongoing Research	46
7.1 Features in Preparation	46
7.1.1 Combination of Logic and Object-Oriented programming paradigms	47
7.1.2 Data Structure Generators	47
7.1.3 Solver Improvements	48
7.2 Future Work	48
8 Conclusion	49
References	53

List of Figures

Figure 1: Muggl's main GUI	14
Figure 2: Parameter selection	15
Figure 3: Editing an array's values	15
Figure 4: The windows that displays the log file	17
Figure 5: Normal execution has been finished	18
Figure 6: Symbolic execution window	20
Figure 7: Symbolic execution windows reporting successful test case generation	21
Figure 8: The architecture of Muggl	23
Figure 9: Step-by-step execution window for normal execution	30
Figure 10: Step-by-step execution window for symbolic execution	32
Figure 11: One tab of the options dialogue	33
Figure 12: Editing class path entries	34
Figure 13: Class file inspection window	35
Figure 14: Class file inspection of a method's definition	35
Figure 15: Class file inspection of the constant pool	36
Figure 16: UML class diagram for <code>de.wwu.muggl.vm.classfile</code>	41
Figure 17: UML class diagram for <code>de.wwu.muggl.vm.classfile.structures</code>	41
Figure 18: UML class diagram for <code>de.wwu.muggl.vm.classfile.structures.attributes</code>	42
Figure 19: UML class diagram for <code>de.wwu.muggl.vm.classfile.structures.constants</code>	43

List of Tables

Table 1:	Types supported by the JVM	6
Table 2:	Type-sensitive bytecode instructions	12
Table 3:	Experimental results for test case generation and elimination.	45
Table 4:	Additional experimental results	46

List of Listings

- Figure 1: Basic struct of a Java class file 8
- Figure 2: Basic struct of the code attribute of a Java class file 9
- Figure 3: Main loop of the JVM 10
- Figure 4: Frame execution loop of the JVM 10
- Figure 5: Muggl's instruction interface 11
- Figure 6: Simple symbolic execution example 17
- Figure 7: Symbolic execution example with conditional jump 19
- Figure 8: Symbolic execution example with a choice that is no real choice 22
- Figure 9: Example of a test case file generated by Muggl 24
- Figure 10: Binary search in Java 26
- Figure 11: Bytecode for binary search—part 1 (initialization) 26
- Figure 12: Bytecode for binary search—part 2 (while body) 27
- Figure 13: Unpropitious loop 28
- Figure 14: Java memory puzzle 29
- Figure 15: Bytecode for JavaMemoryPuzzle 29
- Figure 16: Java memory puzzle solution 30
- Figure 17: Java memory puzzle solution—our simple version 30

Working Paper Sketch

Type

Research Report

Title

Muggl: The Muenster Generator of Glass-box Test Cases.

Authors

Tim A. Majchrzak is a research associate at the Department of Information Systems of the University of Münster. Herbert Kuchen is a Full Professor at the Department of Information Systems and Chair of Practical Computer Science.

For inquiries, please contact Tim A. Majchrzak (tima@ercis.de).

Abstract

Testing is a task that requires much effort, yet it is essential for developing software. Automated test case generation (TCG) promises to relieve humans of manual work. We introduce Muggl (the Muenster generator of glass-box test cases), which is developed at our institute. Muggl generates test cases for Java bytecode. It symbolically executes code and uses constraint solving techniques. While papers on Muggl have already been published, no comprehensive introduction of the tool exist. This working paper fills this gap.

Keywords

software test, testing, test case generation, TCG, test automation, test tool

1 Introduction

Software testing requires much effort. Despite innovations with regard to testing techniques, it still is mainly a manual task [21]. It is particularly time-consuming if a high code-coverage is to be reached. At the same time, testing is an essential task when developing software. Without software tests, it is almost impossible to achieve the desired level of quality. An approach towards relieving humans of manually designing test cases is automated test case generation (TCG). TCG tools promise to generate tests with no or little intervention by testers.

In this working paper, we introduce Muggl. The **M**uenster **g**enerator of **g**lass-box test cases has been used in papers already [19, 20], but it has not been comprehensively described, yet. We intend to fill this gap by describing Muggl's main working principles and basic features. Furthermore, we explain advanced functions and present results from experimental evaluation. Additionally, future research is sketched.

Muggl is based on GlassTT [24] which was designed at the European Research Center for Information Systems (ERCIS). Since some design principles were changed, Muggl was rewritten from scratch. However, it incorporates the constraint solver developed for GlassTT [15]. Much effort and knowledge has been put into designing Muggl. As of February 2011, it consists of 528 classes with 3 234 methods, 68 interfaces, and 44 436 lines of code (LOC) (excluding comments) in 2,67 MiB of source code. Moreover, a documentation of 1 250 HTML files with a total size of over 20 MiB has been prepared.

This paper is organized as follow. Section 2 discusses work related to Muggl. An introduction into the Java Virtual Machine (JVM) is given in Section 3. In Section 4 the basic principles of Muggl and its architecture are described. Section 5 introduces the distinct features of the tool. Experimental results are compiled in Section 6. Section 7 highlights ongoing research and Section 8 draws a conclusion.

2 Related Work

For reasons of scope, we will neither discuss the history of automated testing nor only loosely related approaches in the paper but focus on comparable approaches that have recently been published. Using symbolic execution for TCG is no new approach (cf. e.g. [12]). Symbolic execution even is discussed in textbooks on testing such as [16, 33].

Mostly older work that relates to Muggl is discussed in the papers on its predecessor [24, 23, 15]. A general overview of automated test case generation is given in [28] and [7]. Work that relates to Muggl's strategies for elimination of redundant test cases is discussed in [19].

Five approaches are particularly notable and can be compared to Muggl:

- DOYLE and MEUDEEC describe *IBIS* [6]. It uses symbolic execution of Java bytecode in combination with constraint solving. *IBIS* internally works with a *Prolog* representation of class files. However, the constraint solver is an external component. *IBIS* is not finished; it does not even cover the full set of bytecode instructions. Unfortunately, no progress has been reported in the recent years.

- FISCHER and KUCHEN have developed a tool for testing functional logic programs (in particular for Curry) [9, 10]. It is very similar in concept to Muggl. There is a number of differences due to the hardly compatible programming paradigms, though. For example, Muggl has to cope with data structures such as arrays and objects, which do not even exist in Curry. Moreover, FISCHER

and KUCHEN do not currently include a constraint solver. Nevertheless, their approach is very interesting since it underlines the general feasibility of our (and their) approach.

■ *Pex* [30, 11, 31] has been developed by Microsoft research. It is a TCG tool for .NET-based programs. Despite some different ideas of handling specialties that differ between Java and .NET, and despite being based on path-bounded *model checking*, Pex is closely related to Muggl. However, it does not use an own virtual machine but instruments the *.NET profiling API*. Its constraint solver *Z3* [4] is different in concept to Muggl's solver. It is a satisfiability modulo theory (SMT) solver which natively supports data structures such as arrays. Research on Pex is ongoing.

■ *Symstra* [36] pursues simple ideas as Muggl. However, its techniques resemble symbolic model checking.

■ *ATGen* [22] is a TCG tool for Ada. Despite some similarities, it does not have sophisticated features such as coverage tracking.

3 The Java Virtual Machine

Muggl implements the Java Virtual Machine (JVM) specifications and provides a *normal* (i.e. non-symbolic) JVM that is extended by symbolic execution. Therefore, the JVM is briefly introduced. The reader is supposed to be roughly familiar with the Java programming language [2] and the Java Runtime Environment (JRE). The following summary is based on Oracle's specification [17].¹

3.1 Basic Concepts

The JVM has been built to execute Java code. A number of concepts have been built into it [17, Chapter 2]. The most important concepts are:

■ The JVM is an abstract computing machine that serves ample purposes. It provides multi-threading.

■ Whereas Java code by concept is platform-independent, platform-dependent version of the JVM exist. The JVM is the interface between Java bytecode execution and hardware and operation system.

■ The JVM executes bytecode. It does not necessarily have to be compiled from Java code. Many other programming languages can be compiled to `class` files, which are executed by the JVM.

■ Bytecode is verified before run by the JVM.

■ It natively supports *Unicode* for string representation.

■ The JVM enforces strong typing (see Section 3.2).

■ A number of special classes are provided that are handled differently than other classes. This includes `java.lang.Object`, the root class, and `java.lang.String`, which is cached by the JVM.

■ Several different forms of variables are provided by the JVM: static *class variables*, *instance*

¹To be precise, the specification was released by Sun before it was acquired by Oracle.

variables (members), *array components*, *method parameters*, *constructor parameters*, *exception-handler parameters*, and *local variables*.

- Virtual invocation is supported. The method to be invoked is determined at runtime.
- Methods can be overloaded.
- Objects are instantiated by using a constructor. A constructor is compiled to a method with a special signature.
- Interfaces and abstract methods are supported. A class might have multiple interfaces but only single inheritance is supported.
- Classes and interfaces can be nested, i.e. *inner classes* are supported.
- Arrays of any type are supported. Arrays can be multidimensional.
- The JVM provides automatic memory management.
- *Exceptions* can be used to interrupt the normal control flow. They are passed from the invoked method to the invoking method until the affected code is surrounded by a *try-catch* block of the appropriate exception type. Caught exceptions are handled by the *exception handler*. All exceptions inherit type `java.lang.Throwable`. `java.lang.Error` and `java.lang.RuntimeException` are unchecked, i.e. they do not need to be added to a method's signature even if they might be thrown. Any other exception of type `java.lang.Exception` or one of its subclasses has to be checked and thus declared by the method's signature. Exception handling is an important function of the JVM. For reasons of brevity, it is not further discussed here. For details please refer to [17, Chapter 2.16].

Besides these concepts, the JVM makes assumptions with regard to data types, `class` file format, memory abstraction and execution. They are discussed in the following sections.

3.2 Data Types

Java is statically typed. Type conversions are explicit; for primitive types, conversion instructions exist. Reference types have to be casted. Even though Java supports automatic (*un-*) *boxing* of classes that wrap primitive types (such as `java.lang.Integer` for **int**), this is compiled to an explicit operation in Java bytecode.

The types supported by the JVM are compiled in Table 1. The prefix is used for type-specific bytecode instructions (see Section 3.6).

All types can be explicitly used but the *return address*. It is only internally used by instructions for exception handling. Boolean types are not supported by bytecode instructions; values of type `boolean` can be generated, but working with them is done with instructions for type **int**. While casting to `byte`, `char`, and `short` is included as narrowing instructions, calculations with these values are done using the corresponding instructions for integers.

Additional details on typing and conversions is given in chapters 2.4ff. and 3.2ff. of the JVM specification [17].

²The actual memory utilization of many types is higher.

type	format	length ²	bytecode prefix	initial value
boolean	Boolean value	1	z	false
byte	signed integer	8	b	(byte) 0
char	Unicode sign	16	c	\u0000
double	IEEE754 float	64	d	0.0
float	IEEE754 float	32	f	0.0f
int	signed integer	32	i	0
long	signed integer	64	l	0L
reference	reference pointer	32	a	null
return address	address pointer	32	a	(no default)
short	signed integer	16	s	(short) 0

Table 1: Types supported by the JVM

3.3 Architecture and Memory Abstraction

Implementing a JVM requires to be able to read `class` files and to execute them in accordance with the specification. Many other details are left to the developer's discretion, e.g. the garbage collection algorithm used and runtime optimizations applied.

The JVM is an abstract machine that basically can be implemented on any architecture. It reads `class` files and executes bytecode instructions on the hardware it runs on. Main components are the *class loader*, the *verifier*, and the *execution unit*. Class loading and verification are important parts of executing Java bytecode. Verification is particularly important for the JVM since class files can originate from untrusted sources. Runtime problems known from other programming languages such as *buffer overflows* or *dangling pointers* [8] cannot happen in the VM. Problems that might occur when executing instructions are either explicitly declared as exceptions that might be thrown or avoided by verifying `class` files. Linking is done dynamically; in particular, classes are loaded at runtime when they are needed. Loading classes can be done from several sources, i.e. directories and `jar` archives. The execution unit relies on classes provided by the class loader and checked by the verifier. It is the virtual processor of the JVM and described in Section 3.5.

The following runtime *data areas* are used by the JVM:

■ *pc register*

For each thread executed in the JVM, the `pc` register contains the address of the instruction currently being executed. The address equals the offset into the byte array of instructions (cf. Section 3.6).

■ *JVM stacks*

The JVM is a stack-based machine. Each thread has a JVM stack on which so called *frames* are stored. Frame are top level units of execution and wrap methods currently executed.

■ *Heap*

The heap of the JVM is shared by all threads executed in it. Any objects created during execution are stored on the heap. They can be accessed by their *reference value*. Memory management of the heap is not specified by the JVM. However, objects no longer in use have to be taken care of. In general, this is done by using a *garbage collection algorithm* which is invoked cyclically.

■ *Method area*

The method area is created at JVM startup and shared among threads. It contains data from loaded classes such as the *runtime constant pool* and additional structures required during execution.

■ *Runtime constant pool*

Each class loaded into the JVM has a constant pool of its own. Constant pools comprise constant values used by this class, for example in invoking methods or as strings required during execution (to initialize objects of type `java.lang.String`). Class constant pools are loaded into the runtime constant pool.

■ *Native method stacks*

Not all methods executed by the JVM are implemented in Java. Some are written in languages which are compiled to code native to the machine the JVM runs on. In order to execute them and to get results from these methods, native method stacks are used.

Moreover, each thread has a memory area of its own and can be referenced by a thread number. While the JVM specification strictly dictates the layout of some data areas and other conventions such as floating point arithmetics, it is relatively vague on other issues of implementation. For example, no particular internal structure for objects is mandated.

For actual execution, the JVM uses so called *frames* which are stored on the JVM stacks. A frame stores data and partial results, and it is used to perform dynamic linking and to handle exceptions. With each method invoked, a new frame is generated. A frame comprises of the following components:

■ *Local variables*

Each frame comprises an array of local variables. The maximum number of local variables is determined statically. Local variables are used to access method parameters (arguments) and to store intermediary results. Values of type `long` and type `double` occupy two consecutive local variables for historic reasons. All other values (including references) occupy one local variable.

■ *Operand stack*

Each frame uses an operand stack on its own. All instructions that perform calculations use the operand stack, i.e. operations are not directly performed on local variables. Instead, variables are loaded onto the stack and the calculation is performed. Then, another calculation follows until either the topmost element of the stack is returned and execution of the frame finishes or an element is stored into a local variable.

3.4 Class Files

A Java `class` file is a stream of 8 bit bytes. Data is saved as 8 bit (`u1`), 16 bit (`u2`), 32 bit (`u4`), or 64 bit unsigned integers which are constructed by reading in one byte respectively two, four, or eight consecutive bytes. Data items with more than one byte are stored in big-endian order (with the high bytes first) [17, Chapter 4]. The `class` file has to be read sequentially; random access is impossible since information on the length of blocks of associated data is usually part of this data. The basic *struct* (similar to C notation) is shown in Listing 1.

In general, `class` files have a tree-like structure. Variable areas contain data of fixed length and might contain additional data of variable length, thus branching further. The tree structure can also be observed in Muggl's class representation of Java classes (see Section 5.10).

The main elements of a `class` files can be described as follows. Entries are in the same order as encountered in the files.

1. Each file begins with the *magic word* `0xCAFEBABE`.
2. Two 32 bit values describe the version number of the class. It is the required version of the virtual machine; it is not a number set at the programmer's discretion but inserted by the compiler.


```

1  ClassFile {
2      u4 magic;
3      u2 minor_version;
4      u2 major_version;
5      u2 constant_pool_count;
6      cp_info constant_pool[constant_pool_count - 1];
7      u2 access_flags;
8      u2 this_class;
9      u2 super_class;
10     u2 interfaces_count;
11     u2 interfaces[interfaces_count];
12     u2 fields_count;
13     field_info fields[fields_count];
14     u2 methods_count;
15     method_info methods[methods_count];
16     u2 attributes_count;
17     attribute_info attributes[attributes_count];
18 }

```

Listing 1: Basic struct of a Java class file [17, Chapter 4]

3. The *constant pool* contains string constants, class and interface names, field names, and other constants.

4. *Access flags* are bit values that denominate flags of the class such as the modifiers (*public*, *protected*, and *private*) and whether the class is an interface or abstract.

5. *this_class* is an entry into the constant pool denoting the name of the class.

6. *super_class* is an entry into the constant pool denoting the super class' name of the class, if any. A super class is a class this class inherits from.

7. There can be a variable number of interfaces. Each interface is denoted by an index into the constant pool that gives its name.

8. Fields, i.e. members, and methods are given in structures of their own.

9. Each class can have a number of arguments which are given by the *attribute_info* structure.

Due to their importance, some structures have to be described with more detail. A comprehensive list of their structs and characteristics is given in chapter 4 of the JVM specification [17].

■ *Constant pool*

The constant pool has a number of entries that denote class and interface names, method names and identifiers, values of numerical variables used during execution, and data of strings used during execution. All of them but `CONSTANT_Utf8_info` reference other constant pool entries. Ultimately, entries of structure `CONSTANT_Utf8_info` are references by all constant pool entries since they contain actual literal sequences.

■ *Fields*

Fields are described by access flags, a name, a descriptor, and attributes.

■ *Methods*

Methods are described by access flags, a name, a descriptor, and attributes. Bytecode instructions are encapsulated by an own attribute. Descriptors are used to denote parameter types and return types.

There is a high number of *attribute* structures. Attributes for example comprise information on in-

ner classes, tables to link bytecode instructions to line number in source code, and information on local variables that are used in frames. The most notable attribute contains the code of methods. Its struct is given in Listing 2.

```

1  Code_attribute {
2      u2 attribute_name_index;
3      u4 attribute_length;
4      u2 max_stack;
5      u2 max_locals;
6      u4 code_length;
7      u1 code[code_length];
8      u2 exception_table_length;
9      {      u2 start_pc;
10             u2 end_pc;
11             u2 handler_pc;
12             u2 catch_type;
13         } exception_table[exception_table_length];
14     u2 attributes_count;
15     attribute_info attributes[attributes_count];
16 }

```

Listing 2: Basic struct of the `code` attribute of a Java `class` file [17, Chapter 4]

The attribute comprises data on the maximum number of stack entries and the maximum number of local variables needed, and the bytecode instructions given as a sequence of up to 2^{16} bytes. Additionally, it can contain an exception table structure that is used for handling exceptions. It represents what `try-catch` and `finally` indicate in source code. Finally, the attribute might have nested attributes.

3.5 Execution Unit

The execution unit controls execution and manages access to the data area. The basic processing can best be explained with pseudo code (Listing 3). The pseudo code has a Java like notation rather than a more abstract one. Simply speaking, frames are popped from the JVM stack until it is empty or only a return value is left on it. A frame is pushed onto the JVM frame when a method is invoked.

Executing a frame can be described in a similar style (Listing 4). The frame execution is a nested loop of the main JVM loop. `executeInstruction()` leads to the execution of the instruction. Simply speaking, the loop moves from instruction to instruction either by incrementing the `pc` or by explicit jumps and executes the instructions. This is continued until either a `return` instruction was executed or an uncaught exception was thrown. Execution in low level programming languages could be realized as a large `switch` statement that has an entry for each bytecode instruction. In high level programming languages (and in Muggl), it is possible to have an interface `Instruction`, which is implemented by each instruction. Executing an instruction leads to a call of `Instruction.execute()`. As an example, Muggl's interface for instructions is given in Listing 5 (comments have been omitted).

The illustrations in this section is simplified. More details on the execution unit are given in several chapters of the JVM specification, namely chapters 2.17, 3, 5, and 8 [17].

3.6 Instruction Set

The instructions of a method in a Java `class` are described by an array of byte values. Each instruction is denominated by a one byte value, i.e. there are 256 possible instructions. However,

```

1 void mainLoop() {
2     Frame frame;
3     int pc;
4     while (!JVM_Stack.isEmpty()) {
5         Object object = JVM_Stack.pop();
6         // Check the type.
7         if (!(object instanceof Frame)) {
8             // It might be the last object.
9             if (JVM_Stack.isEmpty()) {
10                // It is the return value of the application – end execution.
11                finishExecution(object);
12                return;
13            }
14            // Push the value to the currently executed frame.
15            frame = (Frame) this.stack.pop();
16            frame.getOperandStack().push(object);
17        } else {
18            // Continue execution with the popped frame.
19            frame = (Frame) object;
20        }
21
22        pc = frame.getPc();
23        executeFrame(frame);
24    }
25 }

```

Listing 3: Main loop of the JVM

```

1 // this.pc denotes the pc of the JVM.
2 void frameLoop(Frame frame)
3     Method method = frame.getMethod();
4     Instruction[] instructions = method.getInstructionsAndOtherBytes();
5
6     // Activate the frame.
7     frame.setActive(true);
8     while (frame.isActive()) {
9         // Save the pc.
10        int pc = this.pc;
11
12        // Execute the instruction.
13        executeInstruction(instructions[pc]);
14
15        // Jumped too far? pc cannot overflow but when it exceeds 65535, counting
16        // continues at 0.
17        if (this.pc >= Limitations.MAX_CODE_LENGTH) {
18            this.pc -= Limitations.MAX_CODE_LENGTH;
19        }
20
21        // Increment pc to the next instruction's offset if no jump has been made.
22        if (this.pc == pc) {
23            this.pc += 1 + instructions[pc].getNumberOfOtherBytes();
24        }
25
26        // Check if a return instruction has been executed.
27        if (this.returnFromCurrentExecution) {
28            return;
29        }
30    }

```

Listing 4: Frame execution loop of the JVM

with 205 instructions in existence, not all possible numbers are used. Of these 205 instructions, not all are compiled to class files and one is reserved but not used (anymore). Each instructions might have a fixed or variable number of *other bytes* that further specify it. This number can only be determined by sequentially reading the array of instructions.

```

1 public interface Instruction {
2
3     void execute(Frame frame) throws ExecutionException;
4
5     void executeSymbolically(Frame frame) throws ExecutionException;
6
7     String getName();
8
9     String getNameWithOtherBytes();
10
11    int getNumberOfOtherBytes();
12
13 }

```

Listing 5: Muggl's instruction interface

Since the JVM is a stack-based machine, most instructions do not have additional bytes. Most of them expect one or more values they require for computation to be on the operand stack. Therefore, these instructions do not need additional bytes like they would do on a register machine. Typical examples are arithmetic operators, which do not have additional bytes.

There are several ways to structure the instructions. The first distinction made in the working paper is by type sensibility. A great number of instructions regard types while the remaining instructions are general ones. Typed instructions are compiled in Table 2. The instructions to load and store local variables also exist in variants that do not have an additional byte specifying the local variable number. The variable number is inherent to them. This concerns the instructions `dload_0`, `dload_1`, `dload_2`, `dload_3`, `fload_0`, `fload_1`, `fload_2`, `fload_3`, `iload_0`, `iload_1`, `iload_2`, `iload_3`, `lload_0`, `lload_1`, `lload_2`, `lload_3`, `aload_0`, `aload_1`, `aload_2`, `aload_3`, `dstore_0`, `dstore_1`, `dstore_2`, `dstore_3`, `fstore_0`, `fstore_1`, `fstore_2`, `fstore_3`, `istore_0`, `istore_1`, `istore_2`, `istore_3`, `lstore_0`, `lstore_1`, `lstore_2`, `lstore_3`, `astore_0`, `astore_1`, `astore_2`, and `astore_3`. Unsurprisingly, they load and store variables from position 0, 1, 2 or 3 instead of using position specified by an additional byte.

Most instructions are self-explanatory by their category. Some of them require further explanation:

- Whereas `xconst` for numeric types pushes numeric values, `aconst_null` is used to push `null`.
- `bipush` and `sipush` are used to push values of type `byte` and `short`.
- `iinc` directly increments an variable of type `int`.
- Long comparison (`lcmp`) pushes an `int` value of 0, 1, or `-1` onto the stack if the first considered value is equal, greater, or less than the second considered value.
- The corresponding instructions for floating point numbers (`dcmp1`, `dcmpg`, `fcmp1`, and `fcmpg`) work similarly but also take `NaN` (*not a number—a special float value*) into consideration.
- `if_xcmpOP` is based on the comparison of two values popped from the operand stack, whereas `ifOP` compares a value to zero in case of `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, and `ifle` and to `null` in case of `ifnull` and `ifnonnull`.

Besides these type-sensitive instructions, further instructions can be categorized as follows:

- Fetching literals from the constant pool: `ldc`, `ldc_w`, and `ldc2_w`.
- Popping, duplicating, and swapping elements from the operand stack: `pop`, `pop2`, `dup`, `dup_x1`, `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2`, and `swap`.

category	opcode	byte	char	double	float	int	long	short	reference
push constants	xconst			dconst_0	fconst_0	iconst_0	lconst_0		aconst_null
				dconst_1	fconst_1	iconst_1	lconst_1		
						iconst_2			
						iconst_3			
				iconst_4					
				iconst_5					
				iconst_m1					
	xipush	bipush						sipush	
load and store to local variable	xload			dload	float	iload	lload		aload
	xstore			dstore	fstore	istore	lstore		astore
load and store from array	xaload	baload	caload	daload	faload	iaload	laload	saload	aaload
incrementation	xastore	bastore	castore	dastore	fastore	iastore	lastore	sastore	aastore
	xinc					iinc			
arithmetic operations	xadd			dadd	fadd	iadd	ladd		
	xdiv			ddiv	fdiv	idiv	ldiv		
	xmul			dmul	fmul	imul	lmul		
	xrem			drem	frem	irem	lrem		
	xneg			dneg	fneg	ineg	lneg		
	xsub			dsub	fsub	isub	lsub		
bit shifting	xshl					ishl	lshl		
	xshr					ishr	lshr		
	xushr					iushr	lushr		
logic operations	xand					iand	land		
	xor					ior	lor		
	xxor					ixor	lxor		
long and float comparison	xcmp						lcmp		
	xcmpl			dcmpl	fcmpl				
	xcmpg			dcmpg	fcmpg				
conversion	d2x				d2f	d2i	d2l		
	f2x			f2d		f2i	f2l		
	i2x	i2b	i2c	i2d	i2f	i2i	i2l	i2s	
	l2x			l2d	l2f	l2i			
conditional jumps						if_icmpeq			if_acmpeq
						if_icmpne			if_acmpne
						if_icmplt			
						if_icmpge			
						if_icmpgt			
						if_icmple			
						ifeq			ifnull
						ifne			ifnonnull
						iflt			
						ifge			
					ifgt				
					ifle				
return from method invocation	xreturn			dreturn	freturn	return	lreturn		areturn

Table 2: Type-sensitive bytecode instructions

- Switches: `tableswitch` and `lookupswitch`.
- Return from method with void return type: `return`.
- Field access: `getstatic`, `pustatic`, `getfield`, and `putfield`.
- Method invocation: `invokevirtual`, `invokespecial`, `invokestatic`, `invokeinterface`, and `invokedynamic`.

- Object initialization: `new`.
- Array initialization: `anewarray`, `newarray`, and `multianewarray`.
- Unconditional jump: `goto` and `goto_w`.
- Checking reference types: `checkcast` and `instanceof`.
- Synchronization: `monitorenter` and `monitorexit`.
- Widened access to other instructions: `wide`.
- Explicitly throwing an exception: `athrow`.
- Getting an array's length: `arraylength`.
- Instructions for the implementation of `finally`: `jsr`, `jsr_w`, and `ret`.
- Doing nothing: `nop`.
- Unused opcode: `xxxunusedxxx`. This opcode is not used for historic reasons.
- Reserved for debugging: `breakpoint`, `impdep1`, and `impdep2`. These instructions should not be compiled to Java `class` files. They are e.g. used by debuggers.

Information in executing each instruction and on exceptions that it might throw is compiled in [17, Chapter 6].

4 Muggl Basics

Firstly, Muggl's execution principles will be introduced. We then explain the basics of symbolic execution and constraint solving. Finally, the architecture is sketched and an execution example is given.

4.1 Execution Principles

Instead of compiling and executing source code, Muggl utilizes `class` files consisting of Java bytecode [17]. In Order to execute class files, Muggl has a Java virtual machine (JVM) of its own. It generally follows the specification (as sketched in Section 3) but does not implement some of the requirements. Most notably, this applies to threading. Muggl adheres to the Java 1.4 specification and also implements most of the additions that were made to the language with the 1.5 and 1.6 releases.

There are two main advantages of using bytecode instead of source code. Firstly, compiler optimizations are taken into account. Secondly, a high number of languages besides Java can be compiled to Java bytecode and thus executed. Muggl implements the full set of 205 instructions and hence is capable of executing all programs compiled for the JVM.

Whilst Muggl does not need its graphical user interface (GUI) to be utilized, it is the most convenient way to use it. When launching Muggl, the file selection windows shows up (Figure 1). Its main purpose is to set up execution. On the left hand side, either a directory or a so called `jar`

file can be selected. The latter is an archive that stores several class files as an library. It might also be compressed. `jar` files and derived formats (such as `war` and `ear`) are the common way to distribute Java programs. Muggl is capable of browsing `jar` files to locate `class` files stored in their internal structure. This integration is seamless; users can browse them as if they were directories.

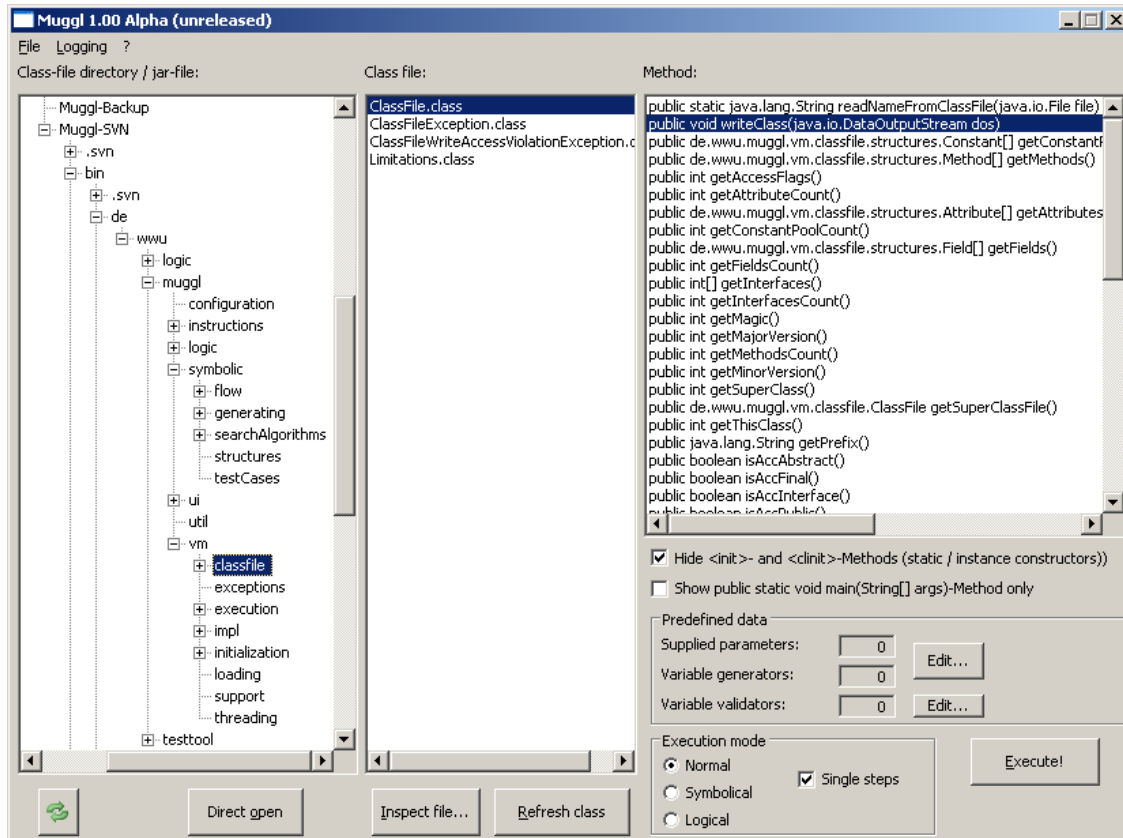


Figure 1: Muggl's main GUI

Once the desired directory is found, a listing of `class` files appears in the middle of the window. Selecting a `class` file shows its methods on the right hand side. It also is possible to inspect the internal structure of the class file, which is interesting for debugging. Class file inspection with Muggl is explained in Section 5.4. If class files are changed while working with Muggl, they have to be reloaded with the “Refresh class” button since Muggl’s class loader (see Section 4.5) caches class files. This is done both for performance reasons, and to adhere to the specification [17].

The method list by default does not list constructors and static constructors. In terms of `class` file representation, they have the method names `<init>` and `<clinit>`. Constructors are used to construct objects and are explicitly invoked on object initialization. Static constructors initialize static fields of classes and may even contain further code. They are automatically executed by the JVM when a class is loaded for the first time. Methods in Java can have four levels of visibility [17]: `public`, `protected`, `default` i.e. `package visible`, and `private`. Since Muggl allows single methods to be executed, it by defaults lists all methods of a class regardless of their visibility modifier. However, it can be selected to only display methods with signature `public static void main(String... args)`. The reference implementation of the JVM by Oracle allows only such methods as entry points of Java programs.

In the lower right area, the windows offers the possibility to further set up execution. The only way to provide parameters to a program executed in the official JVM is to supply `String` arguments

to the `main` methods (or the `null` reference).³ Since Muggl can execute arbitrary methods, the parameters expected by a method have to be provided explicitly. Otherwise, execution fails if the executed methods tries to access the respective parameter. Muggl offers the parameter selection windows for this purpose (Figure 2). Parameters of primitive types, their respective wrapper objects (of types such as `java.lang.Integer` or `java.lang.Double`), and strings can be set. For objects, `null` can be set. Alternatively, a loader object can be provided which instantiates the object provided to the methods. The latter feature is not yet finished and currently disabled. Any parameter can be set to be *undefined* as well, which is important for symbolic execution as explained in the next section.

If a parameter is set to be *undefined*, which also is the default setting, accessing it by a `load` instruction will halt Muggl's JVM with an `de.wwu.muggl.vm.execution.ExecutionException` being thrown. This is an internal exception type of Muggl; it is processed by Muggl and *not* thrown within the executed program. The exception indicates that a state was encountered which is not defined. In Java, all variables have default values. However, the JVM specification does not describe arbitrary method invocation without supplying arguments. In fact, doing this would lead to a prior exception because the calling method is expected to have at least as many elements on its operand stack as the invoked method expects as arguments.

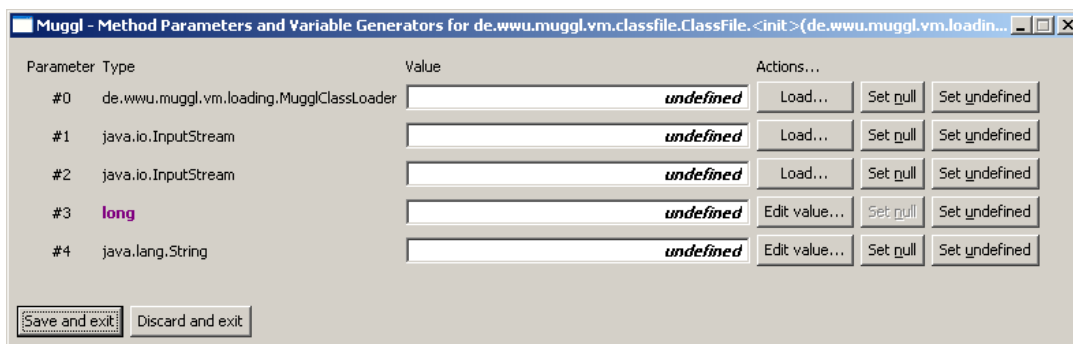


Figure 2: Parameter selection

Muggl also provides functionality to set up arrays passed to methods (Figure 3). It can create and modify arrays of arbitrary types. Arrays values can be set for primitive types, their wrapper classes, and for strings. For arrays of other objects, elements can be set to null. In the future, it will also be possible to instantiate distinct objects.

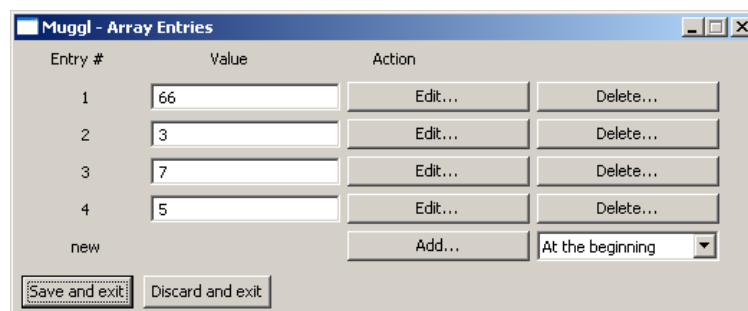


Figure 3: Editing an array's values

Besides constant parameters, so called *generators* and *validators* can be defined. Both are experimental features currently in preparation (see Section 7.1).

³Of course, programs accept further input during execution, e.g. through reading files, by receiving input from the console, from network access, or from interaction in GUI windows.

Finally, the execution mode can be chosen. “Normal” execution describes the standard behavior of the JVM as defined by the specification. “Symbolic” enables symbolic execution as described in the following section. The “Logical” mode is similar to the symbolic one but optimized for execution of programs that combine Java with logic programming. This work is an experimental feature mentioned in Section 7.1.1. Furthermore, “single steps” can be selected to enable Muggl’s step-by-step execution and debugging mode. It is described in Section 5.2.

The menu offers access to the option dialogue, which is described in Section 5.3. Furthermore, it offers the possibility to set a logging level. Muggl uses the log4j framework, the Java standard facility for logging. Log4j offers various logging levels (*fatal*, *error*, *warn*, *info*, *debug*, and *trace*), which indicate a varying criticality of the logged event (cf. [14]). Muggl’s code makes extensive usage of logging capabilities:

- *Fatal* is not used because such events indicate that the system could not recover. Even if Muggl’s JVM crashes, Muggl recovers operation and can fork another JVM.

- Events of level *error* are only rarely reported. They indicate states of the JVM that were not expected and in any case hint to flaws in Muggl’s implementation. If an error occurs, execution in the JVM cannot be continued. The log file should be examined carefully in order to locate the defect in Muggl.

- Warnings (level *warn*) are not as rare as errors. They are not encountered during normal execution, though. Warnings indicate missing implementation of features in Muggl or serious configuration problems. If Muggl logs a warning, it gracefully proceeds but results might be imprecise. An example would be missing test cases. There are options that toggle Muggl’s behavior with regard to logging warning. In some cases, the graceful behavior might be unwanted and instead of it a halt desired.

- Events of level *info* are logged every now and then and are expected. Only notable events are logged; typically, a few messages are logged per minute of program execution.

- *Debug* messages concern sub modules of Muggl that report their operation in detail. This log level will produce a considerable amount of messages.

- *Trace* is the most verbose option. Muggl will even log the most subtle changes in its state. While this log level is helpful for detailed debugging and understanding Muggl’s operation, it generates MiBs of log files within a few seconds of execution.

Log files can either be viewed by directly opening the log file or by using Muggl’s log window (Figure 4). Log messages are logged with a timestamp relative to Muggl’s start, the affected thread, the log level, a category (such as *general*, *execution*, *symbolic*), and the log message. Log files are numbered, i.e. existing log files are not overridden but each instance of Muggl writes to a log file of its own.

If execution is started, the execution window appears (Figure 5). It reports execution’s progress. It will also report any uncaught exception that has been thrown or the successful finish of execution. Execution in the official JVM does not return any value (but an internal return code to the JVM). If Muggl directly executes other methods than the `main` method, there might be a returned value. Muggl will directly display it if it is a primitive type. Otherwise, it will use the returned object’s `toString()` method to display the return value. Thereby, reasonable output can be given for most returned objects.

Time	Thread	Level	Category	Message
0	main	INFO	Muggl general	Logging started. Current logging level is INFO
16	main	INFO	Muggl general	Basic initialization finished. Starting Muggl with up to 66.650.112 Bytes (63,66 Megabytes) of memory available.
17156	main	INFO	Muggl general	Parsing class E:\Daten\Uni-Arbeit\Promotion\Muggl-SVN\bin\test\papers\binaryTree\BinaryTree.class
28141	main	INFO	Muggl general	Parsing class E:\Daten\Uni-Arbeit\Promotion\Muggl-SVN\bin\test\papers\kruskal\EdgesGenerator.class
35953	main	INFO	Muggl general	Parsing class C:\Programme\Java\jdk1.6.0_07\jre\lib\rt.jar\java/lang/String.class
36234	main	INFO	Muggl general	Parsing class C:\Programme\Java\jdk1.6.0_07\jre\lib\rt.jar\java/lang/Character.class
36391	main	INFO	Muggl general	Parsing class C:\Programme\Java\jdk1.6.0_07\jre\lib\rt.jar\java/lang/Object.class
36516	main	INFO	Muggl general	Parsing class C:\Programme\Java\jdk1.6.0_07\jre\lib\rt.jar\java/lang/Throwable.class
36531	Thread-2	INFO	Muggl execution	Starting a virtual machine (thread #13)
42578	Thread-2	INFO	Muggl execution	The virtual machine was halted with an InterruptedException, since the execution was aborted.
42656	Thread-1	INFO	Muggl execution	Waiting for the next virtual machine to start failed with an InterruptedException (probably the execution was aborted). Now checking if another virtual machine is ready, stopping otherwise.
42672	main	INFO	Muggl execution	Execution of an application is finished. Free memory: 3.742.944 Bytes (3,57 Megabytes) of 66.650.112 Bytes (63,66 Megabytes). Cleaning up...
42687	main	INFO	Muggl execution	Cleanup finished. Freed 672.568 Bytes (656,8 Kilobytes).
65469	main	INFO	Muggl general	Parsing class E:\Daten\Uni-Arbeit\Promotion\Muggl-SVN\bin\de\wwu\muggl\symbolic\generating\impl\DoubleIncrementGenerator.class
74094	main	INFO	Muggl general	Parsing class E:\Daten\Uni-Arbeit\Promotion\Muggl-SVN\bin\de\wwu\muggl\ui\gui\components\ClassInspectionComposite\$1.class
74906	main	INFO	Muggl general	Parsing class E:\Daten\Uni-Arbeit\Promotion\Muggl-SVN\bin\de\wwu\muggl\ui\gui\components\ArrayEntriesComposite.class
76547	Thread-7	INFO	Muggl execution	Starting a virtual machine (thread #19)

Figure 4: The windows that displays the log file

4.2 Symbolic Execution

To derive test cases, Muggl executes bytecode symbolically. Input parameters are not treated as constants but used as logic variables [5, 3]. During symbolic computation, variables are bound to terms. Terms are made up of constants, operation symbols (such as arithmetic operators), and variables. Symbolic execution can be explained with a simple example. Consider Listing 6 which is the bytecode for method `int add(int a, int b)`. The column *offset* shows the bytecode instruction's offset in the array of instructions. It is successive but contains "gaps" since several instructions consist of additional bytes (cf. with the JVM specification explainer earlier). The second column denotes the bytecode instruction along with additional bytes. The third column represents the corresponding statement in Java. Finally, the fourth column provides an explanation.

offset	bytecode	Java	explanation
			<i>method invocation</i>
00	iload_1	<i>a</i>	Push int <i>a</i> onto the stack.
01	iload_2	<i>b</i>	Push int <i>b</i> onto the stack.
02	iadd	<i>a + b</i>	Add the two topmost int variables after popping them from the stack and push the result.
03	ireturn	return <i>a + b</i>	Pop the topmost int from the stack and return it.

Listing 6: Simple symbolic execution example

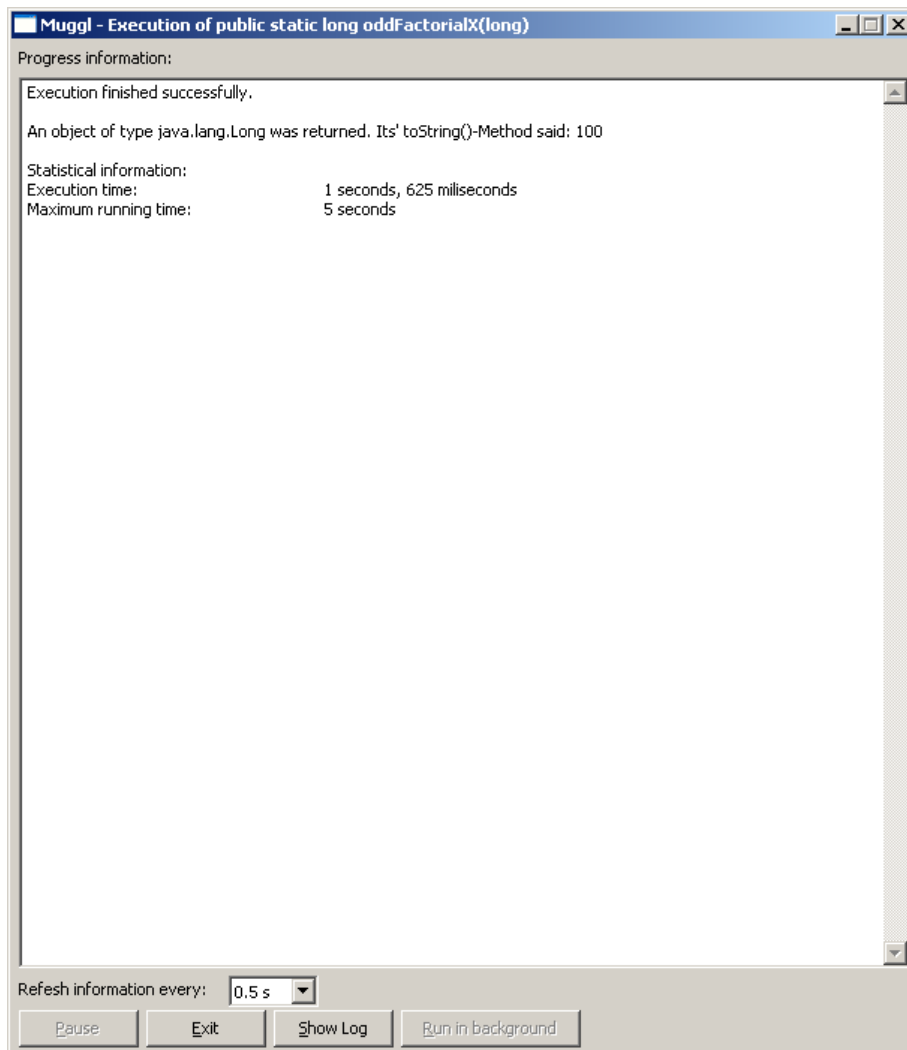


Figure 5: Normal execution has been finished

Of course, symbolic execution will be more complex in realistic examples. Consider Listing 7 for method `int addOrSub(int a, int b)`, which adds some more complexity. A conditional jump is taken with regard to a comparison of a and b .

Each time a conditional jump is encountered, Muggl creates a *choice point*. Along with the choice point, it saves information about the expression that describes the choice taken. Moreover, a *trail* is set up. Trails are a stack data structure (or, more general, a last-in-first-out structure). For each instruction executed after creating the choice point, entries are pushed onto the trail that describe the JVM state changes induced by that instruction. If another choice point is created, its trail is used and a reference to the *parent* choice point is saved.

If a `return` instruction is executed and no more frames are on the JVM frame stack, execution in the normal JVM would be finished. The same applies to an exception that is thrown without being caught. For symbolic execution, this indicates that a solution has been found. Both the expression that describes the returned value and the constraints build for the parameters are saved. Execution, however, is not finished.

Muggl *backtracks* to the last choice point created. Backtracking describes the process of popping elements from the trail and using the information stored in them to restore the state the JVM had at

offset	bytecode	Java	explanation
	<i>method invocation</i>		
00	iload_0	a	Push int a onto the stack.
01	iload_1	b	Push int b onto the stack.
02	if_icmpge 0 7	if ($a < b$)	Jump to offset 09 if ($a < b$).
05	iload_0	a	Push int a onto the stack.
06	iload_1	b	Push int b onto the stack.
07	isub	$a - b$	Subtract the two topmost int variables after popping them from the stack and push the result.
08	ireturn	return $a - b$	Pop the topmost int from the stack and return it.
09	iload_0	a	Push int a onto the stack.
10	iload_1	b	Push int b onto the stack.
11	iadd	$a + b$	Add the two topmost int variables after popping them from the stack and push the result.
12	ireturn	return $a + b$	Pop the topmost int from the stack and return it.

Listing 7: Symbolic execution example with conditional jump

the time of generating the choice point. The process can be seen as a kind of reversed execution. Despite some computational effort from backtracking, using a trail is much more efficient than saving the full virtual machine (VM) state for every choice point. In general, choice points are generated very often and only tiny bits of the state space change between choice points. Saving full states would be a waste of memory.

Once backtracking is finished, Muggl evaluates the negated constraint saved for the choice point. If it can be satisfied, computation continues for the alternative branch of the conditional jump. Otherwise, backtracking is continued to the parent choice point. Once a choice point has no parent and further backtracking is started, the execution is known to be finished. All possible states have been discovered. Test case generation can be started (see Section 4.4).

Symbolic execution forms a search tree; conditional jumps lead to branches. Muggl processes the search tree depth-first. Using a breadth-first strategy would render the concept of using trails almost useless, since keeping the state would require additional measurements. Besides the above described choice point for conditional jumps, further choice points exist for situations in that branches are inserted into the search tree. This can both reflect the behavior of instructions (such as `switch` which has a variable number of edges) and artificial actions. The latter is for example used to generate array values. Muggl does not treat arrays symbolically (but single array values). Instead, it uses a so called *array generator*. This is reflected by using an adequate choice point. On backtracking, it is checked whether more arrays can be created. If so, execution continues with a new array. Otherwise, backtracking continues. The choice point interface and Muggl's search algorithm are flexible to process choice points for varying purposes.

Each path taken through the bytecode corresponds to a distinct test case. If you consider Listing 7, two test cases would be generated. Writing them in a form $\langle \{p_1 \dots p_n\}, r \rangle$ with p_x parameters and r the results from computation, yields $\langle \{a, b\} | (a < b), (a - b) \rangle$ and $\langle \{a, b\} | (a \geq b), (a + b) \rangle$. With concrete values, the test cases could be $\langle \{2, 5\}, -3 \rangle$ and $\langle \{4, 3\}, 7 \rangle$.

For practical problems, the above described procedure becomes extremely complex. Even worse, it is prone to run infinitely. Loops are compiled by using a conditional jump. If they have variable boundaries, symbolic execution might continue endlessly. Therefore, Muggl employs a high number of sophisticated techniques to keep the level of complexity low, to counter the so called state-space explosion, and to speed up execution. Many of them are discussed in the remainder of this article.

Symbolic execution is run in the same windows that the concrete information is run in. However, much more detailed information can be provided (see Figure 6). First of all, general execution information are shown. This includes running time, number of executed frames and instructions, and instantiated classes along with statistic calculations with these figures. Furthermore, data with regard to the search algorithm and constraint solving is provided.

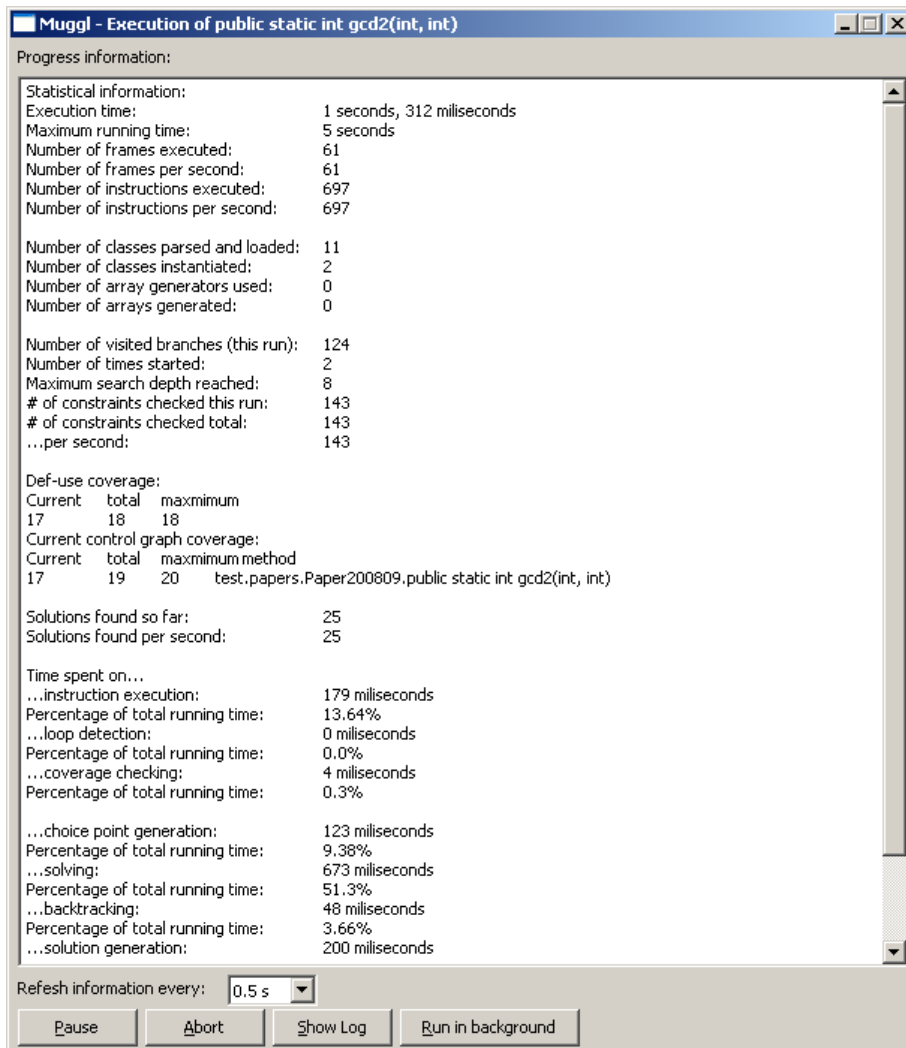


Figure 6: Symbolic execution window

If enabled, the window also shows data on code coverage. Moreover, the number of yet found solutions can be observed. Finally, information on the time tracked for various operations is shown along with its share of the total running time. Differences in shares of running time found for distinct programs help to understand how test case generation works for them and where bottlenecks in Muggl might reside. The window offers buttons to pause and abort execution, to set the delay between refreshes, and to show the log file. Furthermore, Muggl can be minimized by clicking "Run in background". It appears as an icon in the task bar and reports end of execution by a speech balloon (or similar means in accordance with the operating system).

After execution has been finished, the information so far provided in the window is frozen (see Figure 7). It can be observed later. In the upper part, progress with test case elimination is reported (as explained in Section 5.6). Finally, Muggl reports that it wrote a test case. The button "Open test case file" is activated which directly launches an editor showing the JUnit test case.

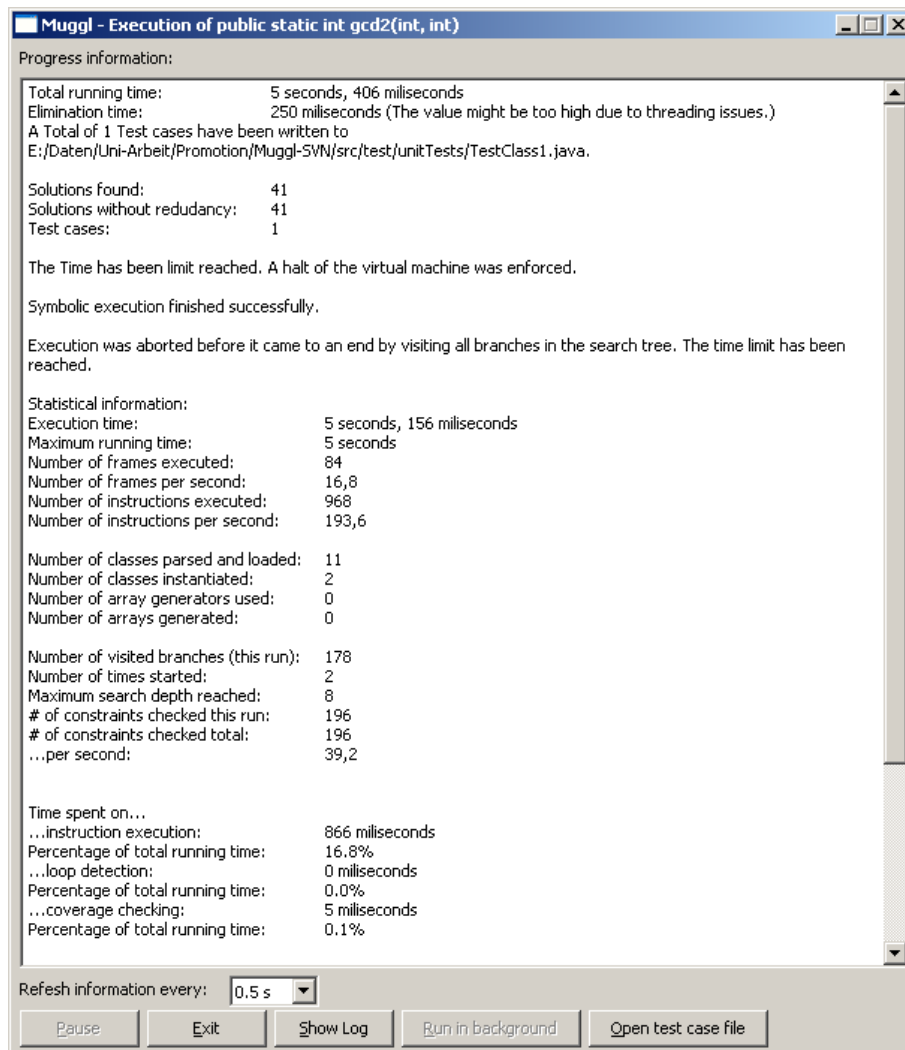


Figure 7: Symbolic execution windows reporting successful test case generation

4.3 Constraint Solving

To be able to decide whether expressions are solvable, or not, Muggl uses a so called *constraint solver*. Its merits can be illustrated by using an example. Consider Listing 8 which displays the method `int choice(int a)`. The method compares a to b and returns either a or $-a$ in accordance with the comparison's result. Symbolic execution does not need to consider both results, though. For semantic reasons, there is not really a choice: b is calculated as two times a . Thus, $(a < b + 5)$ will always be satisfied. Even $(a < b)$ would always be satisfied. If a choice point is created at the instruction with offset 08, the expression is checked by the constraint solver. Since it can determine that $(a \geq b + 5)$ cannot be fulfilled, only the branch leading to offset 12 needs to be considered. In realistic examples, this dramatically reduces the number of states that have to be explored.

Constraint solving is helpful even if the situation is not as obvious as in this example. Consider that b would again be a parameter and would not be calculated as $b = a * 2$. In this case, $(a < b + 5)$ would form a constraint that is used for further computation and that might help to decide on expressions encountered at a later point of execution.

offset	bytecode	Java	explanation
	<i>method invocation</i>		
00	iload_0	<i>a</i>	Push int <i>a</i> onto the stack.
01	iconst_2	2	Push int 2 onto the stack.
02	imul	$a * 2$	Multiply the two topmost int variables after popping them from the stack and push the result.
03	istore_1	$b = a * 2$	Pop the topmost int variable from the stack and store it as local variable #1 (<i>b</i>).
04	iload_0	<i>a</i>	Push int <i>a</i> onto the stack.
05	iload_1	<i>b</i>	Push int <i>b</i> onto the stack.
06	iconst_5	5	Push int 5 onto the stack.
07	iadd	$b + 5$	Add the two topmost int variables after popping them from the stack and push the result.
08	if_icmpge 0 5	if ($a < b + 5$)	Jump to offset 13 if ($a < b + 5$).
11	iload_0	<i>a</i>	Push int <i>a</i> onto the stack.
12	ireturn	return <i>a</i>	Pop the topmost int from the stack and return it.
13	iload_0	<i>a</i>	Push int <i>a</i> onto the stack.
14	ineg	$-a$	Pop the topmost value from the stack, negate it, and push it.
15	ireturn	return $-a$	Pop the topmost int from the stack and return it.

Listing 8: Symbolic execution example with a choice that is no real choice

Constraint solving becomes very complex for real problems. Instead of single linear constraints, systems of arbitrary non-linear constraints are built during execution. Muggl's constraint solver [15] offers a variety of solving strategies and performs well for most problems. It offers the representation of all primitive data types of Java. Both linear and non-linear constraints are solved. Linear inequations are solved based on the Fourier-Motzkin algorithm; non-linear equations are solved by search space bisection. A detailed discussion for the solver is out of scope of this working paper, though.

4.4 Test Case Generation

After finishing symbolic execution, all solutions are committed to Muggl's solution processor. The processor serves two goals: Firstly, it eliminates redundant test cases. Secondly, it generates JUnit test case files.

In general, test case generation should lead to a number of test cases that is as small as possible while still being suitable to detect all defects of a program. A small number of test cases is easier to check for humans, and it executes much faster. Two mechanisms are used to keep the number of test cases small. The solution processor has a built-in redundancy checker. It deletes test cases that are semantically equal. Due to unpropitious bounds, solutions with the same constraints are created for some programs. The second mechanism eliminates test cases based on their contribution to control-flow and data-flow coverage. It is described in Section 5.6.

Test case generation usually reduces the number of test cases dramatically. In typical examples, the number of kept test cases is two magnitudes lower than the number of generated ones. For the remaining test cases, Muggl generates a JUnit test case file. This test case file can be executed directly in order to test the program under consideration.

An example is given in Listing 9. It has not been manually modified but for comments of the header being shortened. This is denoted by “[...]”. All test case files written by Muggl have an explanation

of how they have been generated and what the basic setting during test case generation were. Each test case file is a class of its own. Directly following the class definitions, class members are set up. They are used in the actual tests. Muggl writes variable names that hint to their value whenever possible. By choosing class members, using constants in method bodies is avoided. The `setUp()` method is used for further initialization. `testIt()` contains the actual tests. They are designed as JUnit assertion. For a method call with specified parameters, the asserted result is given. If actual and asserted result differ, the test fails. It is moreover possible to check for exceptions. In this case, a `try-catch` block encloses a method call. `catch` does not contain any code. However, directly following the method call is an explicit `fail(String)` which denominates that the test has not been passed. Finally, the class contains a `main` method. It is useful if the test case file is not executed by JUnit because it will invoke JUnit in order to be run.

While the resulting JUnit file has a clear structure, generating it requires a considerable amount of code. All variables used require proper initialization. In particular, each variable of a distinct value should only be defined once. Moreover, objects have to be initialized. Muggl is capable of setting up code that initializes arrays and simple objects. It cannot create code that uses a constructor and a variety of method calls to set up an object. This task can become arbitrarily complex because how to construct an object has to be derived from the existing object when saving a solution. No feasible solution how to do this is known. However, there are means of circumventing this problem which will be dealt with in the ongoing research (cf. Section 7).

4.5 Architecture

An overview of Muggl's architecture is given in Figure 8. Providing large class diagrams is not helpful. With its total of 528 classes with 3 234 methods and 68 interfaces, even models of Muggl's subsystems are very large. A low-level view is only needed to extend Muggl or to use its components in other projects. For this purpose, an extensive API documentation is provided that describes every class and the interrelations between classes and packages.

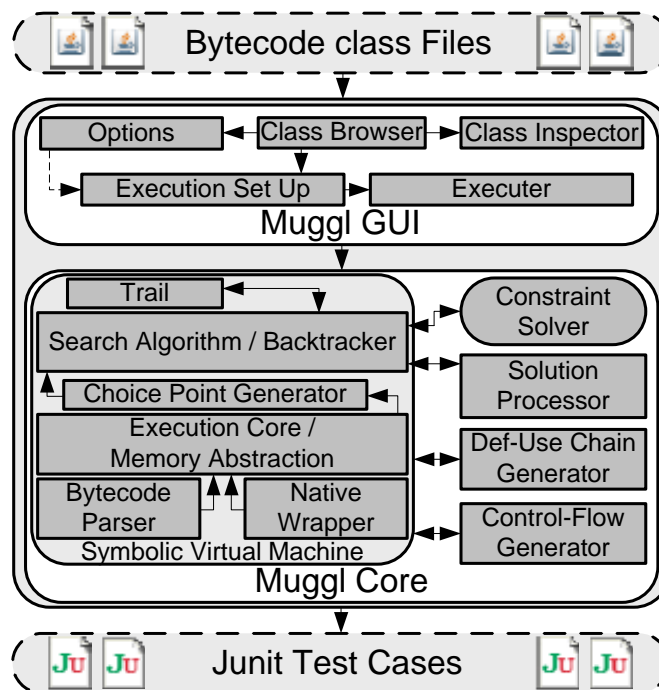


Figure 8: The architecture of Muggl


```

1 package test.unitTests;
2
3 import org.junit.Before;
4 import org.junit.Test;
5 import static org.junit.Assert.*;
6
7 /**
8  * This class has been generated by Muggl for the automated testing of method
9  * test.papers.Paper200809.binSearch(int v, int[] a).
10 * [...]
11 *
12 * The total number of solutions found was 200. [...]
13 * By eliminating solutions [...] the total number of solutions could be reduced by
14   197 to the final number of 3 test cases.
15 *
16 * Covered def-use chains:      60 of 60
17 * Covered control graph edges: 43 of 48
18 *
19 * This file has been generated on Monday, 07 February, 2011 18:27.
20 *
21 * @author Muggl 1.00 Alpha (unreleased)
22 */
23 public class TestClass2 {
24     // Fields for test parameters and expected return values.
25     private int intm1;
26     private int int0;
27     private int[] array0 = {-1,1};
28     private int[] array1 = {0};
29     private int[] array2 = null;
30
31     /**
32     * Set up the unit test by initializing the fields to the desired values.
33     */
34     @Before public void setUp() {
35         this.intm1 = -1;
36         this.int0 = 0;
37     }
38
39     /**
40     * Run the tests on test.papers.Paper200809.binSearch(int v, int[] a).
41     */
42     @Test public void testIt() {
43         assertEquals(
44             this.intm1, test.papers.Paper200809.binSearch(this.int0, this.array0));
45         assertEquals(
46             this.int0, test.papers.Paper200809.binSearch(this.int0, this.array1));
47         try {
48             test.papers.Paper200809.binSearch(this.int0, this.array2);
49             fail("Expected a java.lang.NullPointerException to be thrown.");
50         } catch (java.lang.NullPointerException e) {
51             // Do nothing - this is what we expect to happen!
52         }
53     }
54
55     /**
56     * Invoke JUnit to run the unit tests.
57     * @param args Command line arguments, which are ignored here. Just supply null.
58     */
59     public static void main(String args[]) {
60         org.junit.runner.JUnitCore.main("test.unitTests.TestClass2");
61     }
62 }

```

Listing 9: Example of a test case file generated by Muggl

Muggl is separated into graphical user interface (GUI) and execution core. The execution core also comprises the solver. While being embedded for performance reasons, it is possible to use the solver as a component of its own.

The GUI's components are discussed throughout this paper. It has been designed for seamless work with Muggl but—from a technical standpoint—has no special features.⁴ The GUI is run in a thread of its own to guarantee performance and to avoid that GUI problems (such as delayed rendering) affect the execution core and vice versa.

Besides the symbolic JVM (SJVM), a number of additional components constitute the execution core. Most notably, these are the solution processor, and components that generate and track control-flow and data-flow. The JVM adheres to the JVM specification [17] and utilizes concepts required by it. This for example comprises the bytecode parser including components to represent `class` files, and the memory abstraction. Internally, it consists of the implementation of the bytecode instructions, algorithms for dynamic binding, the exception handler, a `String` cache, and a large number of auxiliary components. Moreover, a wrapper for native methods (cf. Section 5.7) ensures executability of native methods.

If the JVM is switched to symbolic mode, it processes the symbolic implementations of bytecode instructions. All components required for symbolic execution are invoked. Most notably, this is the choice point generator which invokes the search algorithm. The latter comprises the functionality to perform backtracking. Speaking of additional data structures, specifically the trail has to be mentioned.⁵

Muggl's architecture has been designed for modularity. Most components are interchangeable. Key components provide interfaces that ensure adaptability. For example, all implementations of virtual machines (such as the *normal* JVM, the symbolic, and the logic one) are inherited from an abstract Java virtual machine. The search algorithm processes choice points; it does not need to know what kind of choice point it actually handles. This focus on object-oriented design principles and patterns does not only keep Muggl flexible, maintainable, and extensible but also greatly aids the understanding of its source code.

4.6 Execution example

For clarity, a realistic example that Muggl can process is given. Listing 10 shows an implementation of binary search in Java. It is a simple and convenient implementation of the algorithm. If the searched element is not found in the specified array, `-1` is returned. Otherwise, the element's position is returned. No explicit exception is thrown if the second argument is `null`. Thus, invoking the methods and providing it with a `null` instead of an array will result in a `NullPointerException` being thrown on line 3.

The resulting bytecode along with execution information is shown in Listing 11 and 12. After symbolic execution, three distinct test cases remain. The JUnit test case file shown in Listing 9 (p. 24) has been generated for this binary search algorithm.

5 Distinct Features

The described functionality of Muggl constitutes a considerable amount of its code already. However, Muggl has a number of distinct features that either significantly improve the results from

⁴Particularities, such as used for step-by-step processing, are built into Muggl's core and merely visualized by the GUI.

⁵Strictly speaking, there is a number of trails—one per choice point.

```

1 public static int binSearch(int v, int[] a) {
2     int low = 0;
3     int up = a.length - 1;
4     while (low <= up) {
5         int mid = (low + up) / 2;
6         if (v < a[mid]) {
7             up = mid - 1;
8         } else if (v > a[mid]) {
9             low = mid + 1;
10        } else {
11            return mid;
12        }
13    }
14    return -1;
15 }

```

Listing 10: Binary search in Java

offset	bytecode	Java	explanation
	<i>method invocation</i>		
00	iconst_0	0	Push int 0 onto the stack.
01	istore_2	<i>low</i> = 0	Pop the topmost int variable from the stack and store it as local variable #2 (<i>low</i>).
02	aload_1	<i>a</i>	Push array <i>a</i> onto the stack.
03	arraylength	<i>a.length</i>	Pop the topmost array from the stack and get its number of elements.
04	iconst_1	1	Push int 1 onto the stack.
05	isub	<i>a.length</i> - 1	Subtract the two topmost int variables after popping them from the stack and push the result.
06	istore_3	<i>up</i> = <i>a.length</i> - 1	Pop the topmost int variable from the stack and store it as local variable #3 (<i>up</i>).
07	goto 0 45	while	Jump to offset 52.

Listing 11: Bytecode for binary search—part 1 (initialization)

symbolic execution or greatly enhance efficiency. Muggl would theoretically work without them, but is improved by incorporating these features. They are described in the following sections.

5.1 Iterative-Deepening Depth-First Search

A depth-first search of the tree of possible paths through a bytecode program is preferred to a breath-first attempt. Besides the much lower memory footprint, a number of solutions is found much faster. Even though both search strategies yield the same solutions after full execution, breath-first is no suitable approach in almost any case. Even for programs of low complexity, full enumeration often is not possible or at least takes very long. Depth-first quickly reaches leaf nodes in the search-tree. Consequently, solutions can be saved.

The problem depth-first search faces is that for some branches disadvantageous characteristics can be observed during symbolic execution. Execution then spends much time discovering such branches without finding solutions. Much execution time passes until a solution is found and the branch left by backtracking; in unfortunate cases, no solution is found due to semantic restrictions, which means that searching the branch was useless. While it is desirable to find all possible test cases, TCG has to be fast for economic reasons. Therefore, a *reasonable* number of test cases should be found within a small amount of time.

10	iload_2	<i>low</i>	Push int <i>low</i> onto the stack.
11	iload_3	<i>up</i>	Push int <i>up</i> onto the stack.
12	iadd	$(low + up)$	Add the two topmost int variables after popping them from the stack and push the result.
13	iconst_2	2	Push int 2 onto the stack.
14	idiv	$(low + up)/2$	Divide the two topmost int variables after popping them from the stack and push the result.
15	istore 4	$mid = (low + up)/2$	Pop the topmost int variable from the stack and store it as local variable #4 (<i>mid</i>).
17	iload_0	<i>v</i>	Push int <i>v</i> onto the stack.
18	aload_1	<i>a</i>	Push array <i>a</i> onto the stack.
19	iload 4	<i>mid</i>	Push int <i>mid</i> onto the stack.
21	iaload	<i>a</i> [<i>mid</i>]	Pop the topmost int and array from the stack. Get the element from the array as the position denoted by the int and push it.
22	if_icmpge 0 11	if ($v < a[mid]$)	Jump to offset 33 if ($v < a[mid]$).
25	iload 4	<i>mid</i>	Push int <i>mid</i> onto the stack.
27	iconst_1	-1	Push int -1 onto the stack.
28	isub	$mid - 1$	Subtract the two topmost int variables after popping them from the stack and push the result.
29	istore_3	$up = mid - 1$	Pop the topmost int variable from the stack and store it as local variable #3 (<i>up</i>).
30	goto 0 11	(end of while)	Jump to offset 52.
33	iload_0	<i>v</i>	Push int <i>v</i> onto the stack.
34	aload_1	<i>a</i>	Push array <i>a</i> onto the stack.
35	iload 4	<i>mid</i>	Push int <i>mid</i> onto the stack.
37	iaload	<i>a</i> [<i>mid</i>]	Pop the topmost int and array from the stack. Get the element from the array as the position denoted by the int and push it.
38	if_icmplt 0 11	if ($v > a[mid]$)	Jump to offset 49 if ($v > a[mid]$)
41	iload 4	<i>mid</i>	Push int <i>mid</i> onto the stack.
43	iconst_1	1	Push int 1 onto the stack.
44	iadd	$mid + 1$	Add the two topmost int variables after popping them from the stack and push the result.
45	istore_2	$low = mid + 1$	Pop the topmost int variable from the stack and store it as local variable #2 (<i>low</i>).
46	goto 0 6	(end of while)	Jump to offset 52.
49	iload 4	<i>mid</i>	Push int <i>mid</i> onto the stack.
51	ireturn	return <i>mid</i>	Pop the topmost int from the stack and return it.
52	iload_2	<i>low</i>	Push int <i>up</i> onto the stack.
53	iload_3	<i>up</i>	Push int <i>low</i> onto the stack.
54	if_icmplt 255 212	if $low \leq up$	Jump to offset 10 if $low \leq up$ (while condition).
57	iconst_m1	-1	Push int -1 onto the stack.
58	ireturn	return -1	Pop the topmost int from the stack and return it.

Listing 12: Bytecode for binary search—part 2 (while body)

Muggl employs an *iterative-deepening* depth-first search algorithm. It basically follows the search strategy of depth-first. Upon reaching a pre-defined search depth without encountering a leaf node in the search tree, backtracking is activated. Once all nodes of the search-tree for the given depth have been discovered, execution is restarted with an increased depth. This strategy is very advantageous. In almost any case, a high number of solutions is found within a short time. Most leafs are reached on a path with a small depth. At the same time, the approach is flexible: as long as more execution time is desired by the user, execution continues.

Each node in the search tree has at least two children. The average number of children is near to two, since choice points with more than two branches (such as switches) are rarely used. Consequently, the number of states visited again when restarting execution is low. This even applies to an increment of only 1—typically, the increment would be higher.

For simplification consider a search-tree in that each node has exactly two children; assume an infinite depth. The number of nodes for a given depth n is 2^{n-1} . For search depth x , the number of nodes to discover is $2^x - 1$. Even with an increment of 1, the number of states that have to be processed again is only $2^{n-1} - 1$ which is less than half of the total states to discover.

Besides the search algorithm, Muggl has a module that keeps track of the execution of loops. Consider a loop such as shown in Listing 13. If `limit` has a high value, much time is spent executing the loop. In many cases, such behavior does not lead to additional test cases. Thus, Muggl can be configured to force backtracking after a loop has been executed for a specified amount of time. A value sufficient in practice is 200.

```

1  for (int i = 0; i < limit; i++) {
2      // Do something.
3  }
```

Listing 13: Unpropitious loop for high values of `limit`

5.2 Step-by-Step GUI

Muggl is a prototype. As long as it is not exclusively used in productive environments, it needs to have debugging capabilities. Therefore, it provides a step-by-step GUI in that programs can be executed while the JVM's state is visualized. This GUI is not only helpful to debug Muggl. In fact, it can be used to understand a program's behavior both during normal and symbolic execution. Furthermore, it greatly aids to understand how bytecode is processed. It thereby can be used to debug programs. In particular, specialties in the characteristic behavior of the JVM can be comprehended.

The merits of executing Java bytecode step by step can be motivated by using an example. You can easily solve the “Java memory puzzle” announced on the *Java specialist newsletter*. The puzzle is depicted in Listing 14.⁶ On Oracle's JVM this will yield an `OutOfMemoryError`, which cannot be explained from the source code. The program utilizes more than the available memory. It would be guessed that the memory used for `data` would be freed before `data2` is generated since `data` is encapsulated and brackets to indicate that the variable is not accessed afterwards.

A look at the bytecode for `public void f()` reveals the problem (Listing 15). For both arrays, the same local variable is used. Before the second array is stored to local variable #1 at offset 13, it is created at offset 11. However, the first array is still stored as local variable #1 at this time and thus not freed by garbage collection. To observe this, the step-by-step GUI is a very convenient tool.

⁶The puzzle can be found at <http://www.javaspecialists.eu/archive/Issue173.html>

```

1 public class JavaMemoryPuzzle {
2     private final int dataSize =
3         (int) (Runtime.getRuntime().maxMemory() * 0.6);
4
5     public void f() {
6         {
7             byte[] data = new byte[dataSize];
8         }
9
10        byte[] data2 = new byte[dataSize];
11    }
12
13    public static void main(String[] args) {
14        JavaMemoryPuzzle jmp = new JavaMemoryPuzzle();
15        jmp.f();
16    }
17 }

```

Listing 14: Java memory puzzle

offset	bytecode	Java	explanation
		<i>method invocation</i>	
00	aload_0	<i>this</i>	Push <i>this</i> onto the stack.
01	getfield 0 24	<i>this.dataSize</i>	Push value from field <i>this.dataSize</i> onto the stack.
04	newarray 8	<i>new byte[dataSize]</i>	Push new array of type <i>byte</i> with length <i>dataSize</i> onto the stack.
06	astore_1	<i>data = new byte[dataSize]</i>	Pop the topmost array from the stack and store it as local variable #1.
07	aload_0	<i>this</i>	Push <i>this</i> onto the stack.
08	getfield 0 24	<i>this.dataSize</i>	Push value from field <i>this.dataSize</i> onto the stack.
11	newarray 8	<i>new byte[dataSize]</i>	Pop the topmost array from the stack and store it as local variable #1.
13	astore_1	<i>data2 = new byte[dataSize]</i>	Push array <i>a</i> onto the stack.
14	return	<i>(finish execution)</i>	Return from execution.

Listing 15: Bytecode for JavaMemoryPuzzle

The Java specialist newsletter proposed a solution for a program that circumvented the problem and asked its readers to explain the observed behavior. The modified method `public void f()` is shown in Listing 16. However, the messages written to the console meant to distract readers. Despite the first guess, the loop does not allow garbage collection to kick in. As long as the array is stored as a local variable and the frame is not discarded, the memory will not be freed regardless of the time that elapses. In fact, the loop's variable *i* is stored as local variable #1. Thereby, it makes sure that no reference to *data* exists, and the memory occupied by it is freed *before* the second array is generated.

Consequently, a simpler example can be generated that has the same effect. It is shown in Listing 17. Please note that it is necessary to use the code blocks. Otherwise, the local variable would not be reused.

The step-by-step execution window of Muggl is shown in Figure 9. On the left side, the bytecode instructions are listed. For each instruction the offset, the instruction with additional bytes, and more detailed information are given. The latter includes jump targets, values to be pushed for instructions for which this is not obvious, and resolved method identifiers. Moreover, each instruction offset is colored to reflect the jump behavior of it and to indicate whether it may throw an exception, or not.

```

1 public void f () {
2     {
3         byte [] data = new byte [dataSize];
4     }
5
6     for (int i=0; i<10; i++) {
7         System.out.println ("Please be so kind and release memory");
8     }
9
10    byte [] data2 = new byte [dataSize];
11 }
    
```

Listing 16: Java memory puzzle solution

```

1 public void f () {
2     {
3         byte [] data = new byte [dataSize];
4     }
5
6     {
7         int a = 0;
8     }
9
10    byte [] data2 = new byte [dataSize];
11 }
    
```

Listing 17: Java memory puzzle solution—our simple version

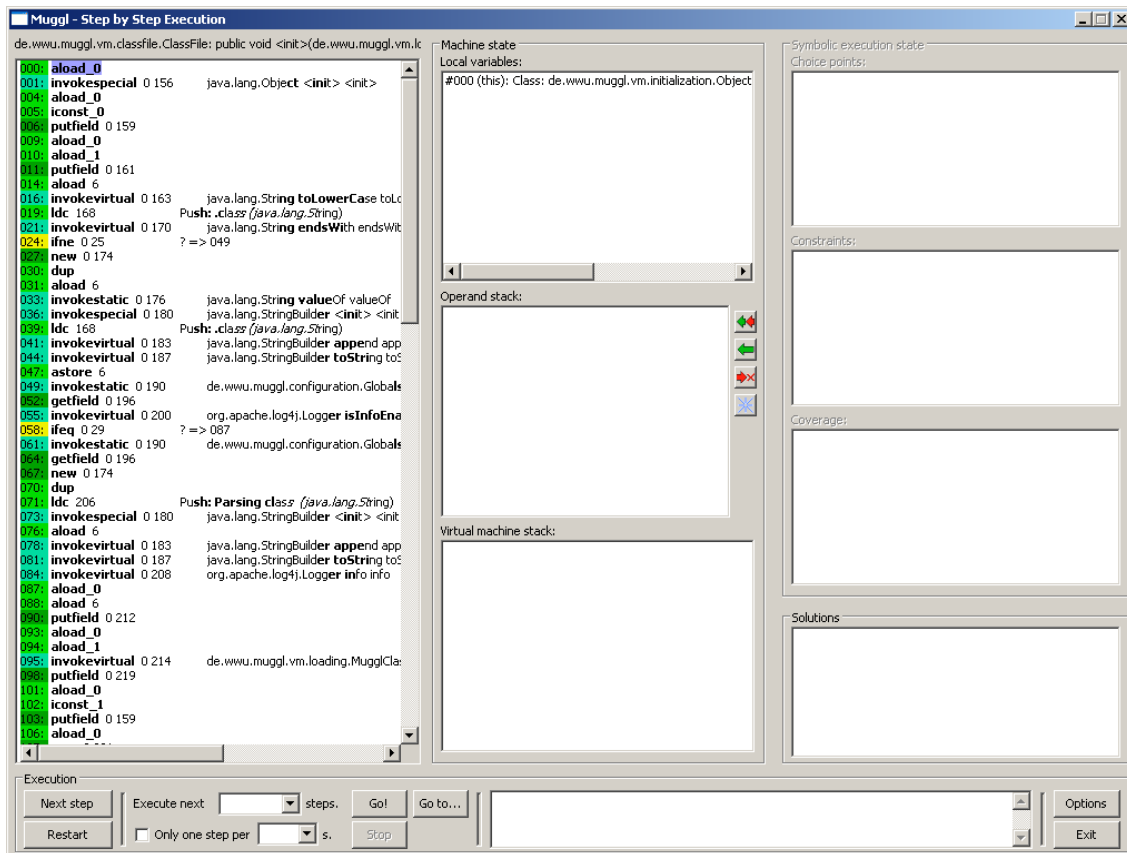


Figure 9: Step-by-step execution window for normal execution

In the middle of the window, already defined local variables are shown by number, variable name, type, and value. Values are directly shown for primitive types; for objects, their `toString()` method is used to get a string representation. The following area shows elements that are on the operand stack. With the help of four buttons it is possible to push and modify stack elements of primitive types and their corresponding wrapper classes, to pop elements from the stack, or to clear it completely. Finally, the third area shows frames that are on the virtual machine stack.

The right side of the window is used for the state of symbolic execution and is unavailable during normal execution. The “solution” area will be used to display a value returned from execution.

In the lower part of the window, execution can be controlled with the help of a number of buttons. It is possible to execute one step or a number of consecutive steps. Stepping can be stopped and the time spent per step can be set. Moreover, it is possible to execute a number of steps without showing each single step; the window is refreshed with the current state once the desired number of instructions have been executed. Moreover, execution can be restarted. Additionally, a small area shows the most recent messages written to the log file. Finally, the options dialogue can be opened or the window closed (and step-by-step execution thereby halted).

If symbolic execution is chosen, the step-by-step window makes use of the three additional areas on the right side (Figure 10). The upper area lists choice points currently active. Their number in descending order is given along with their type. Moreover, the method in which they have been set, the offset (pc of the executed frame), and the instruction that created them are given. The second area lists constraints currently on the constraint stack. They are given in descending order; when backtracking, the topmost one will be negated. Finally, the third area gives information on control-flow and data-flow coverage.

In the “Solutions” area, possible paths through the program are listed by constraints that describe the input parameters and a constraint (or an exception type) that describes the output. No constant values are assigned as this point.

The step-by-step GUI has proven to be a versatile tool. It is particularly helpful if errors occur during execution. Since the execution window shows the number of executed instructions, it is possible to restart execution using the step-by-step window and to then skip execution to an appropriate point. In general, this would be a few instructions before the error halted execution. Using the step-by-step functionality, it can be observed what happened to the JVM state right before the error occurred. The problem can be understood—and fixed—in many cases. Despite some cases in which errors have their roots in an error that occurred ten thousands of instructions executed prior, the step-by-step GUI helped to greatly enhance Muggl.

5.3 Configurability

Muggl is an experimental tool. While a commercial TCG tool should not need much configuration, Muggl is highly configurable. Most of its standard values are suitable for most programs to generate test cases for. Changing some key values has great influence on how successful TCG is. In cases where Muggl does not generate test cases or requires too much time, optimizing additional values can improve execution.

For reasons of performance and encapsulation, Muggl has several *singleton* objects that provide general options and particular settings. All changeable values are read from an XML (Extensible Markup Language) configuration file. Currently, over 60 setting can be changed. Further settings can be modified with regard to programs when running Muggl (such as predefined parameters). Additionally, Muggl provides some hard-coded settings. They require compilation of class `Globals` to be changed. These are values that might be changed in the future (or even provided as configurable settings), but at the moment it would not be reasonable to touch them. Examples are

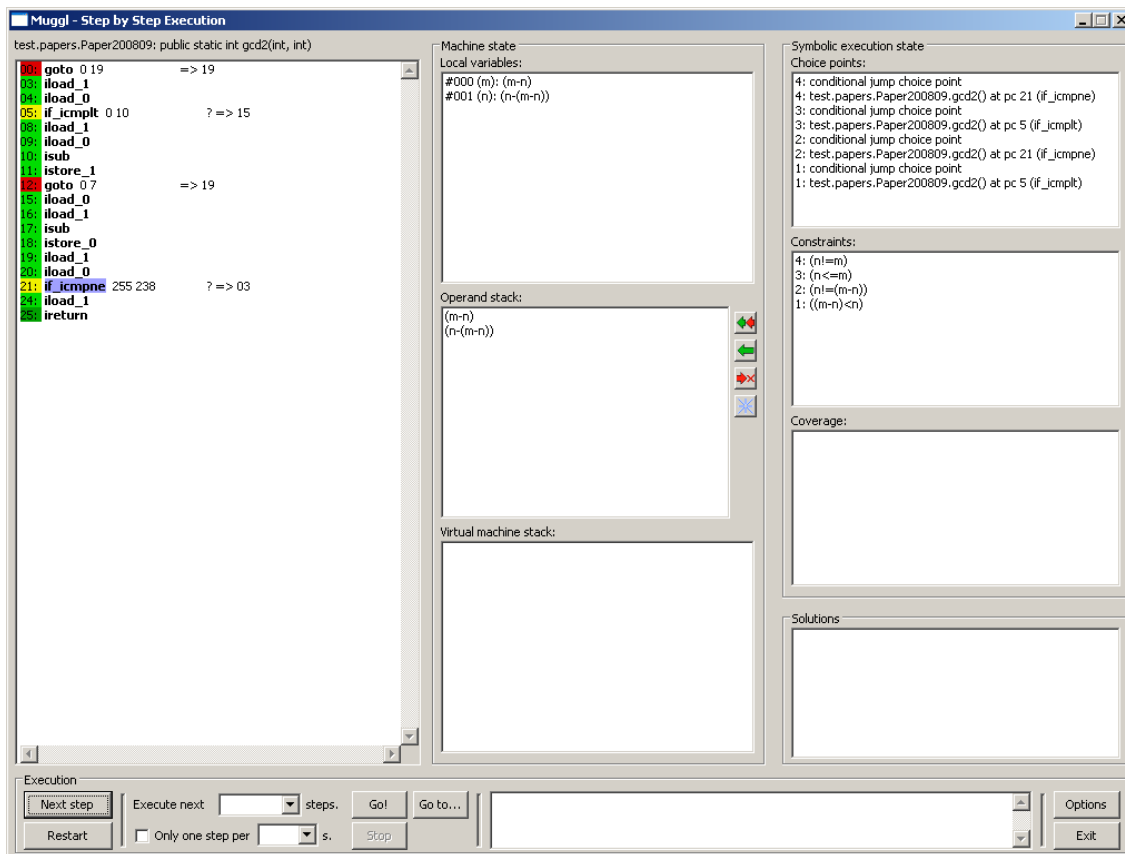


Figure 10: Step-by-step execution window for symbolic execution

the version number of Muggl and a minimum delay in milliseconds used when threads are temporarily stopped. The idea is to never use numerical constant in Muggl's code. Constants are either defined in appropriate classes, or—if they are general constants and cannot be attributed to a class—in the above mentioned `Globals` class.

With the exception of a small number of experimental settings that are subject to be removed in the future, all settings can be changed in Muggl's option dialogue. Options are provided in tabs for a variety of categories (see Figure 11). The following options are provided:

■ *General*

In this tab, the Java Runtime Environment (JRE) can be specified and the used class path edited. Both directories and jar files can be added to the class path (see Figure 12).

■ *File inspection*

This tab allows to customize colors that are used in the class file inspection windows (cf. Section 5.4).

■ *Visual*

GUI options both for the main windows and the step-by-step GUI can be set in this tab. In particular, several levels of visually skipping method invocation in step-by-step mode can be chosen.

■ *Execution*

This tab offers options with regard to the virtual machine's general behavior. The maximum runtime can be set, handling of native methods can be configured (cf. Section 5.7), and dynamic optimization of instructions can be enabled (cf. Section 5.8).

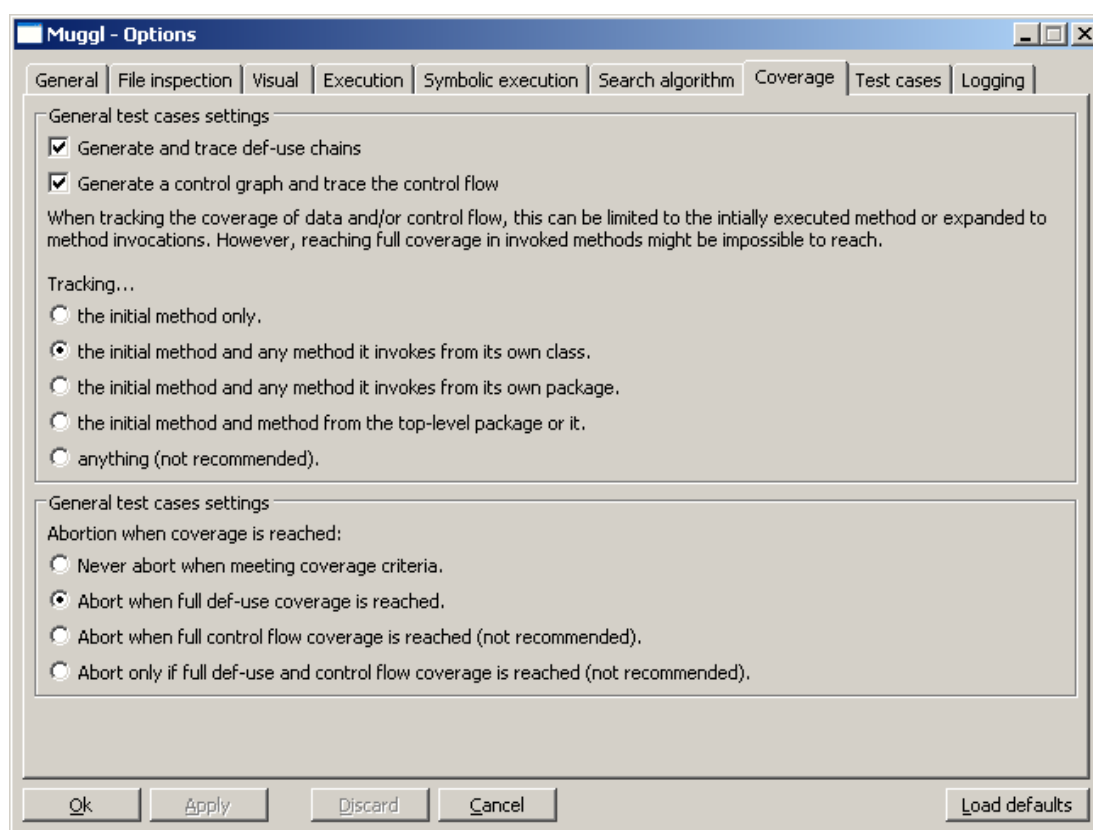


Figure 11: One tab of the options dialogue

■ *Symbolic execution*

In this tab, strategies for array generation can be customized (cf. Section 7.1).

■ *Search algorithm*

This tab offers the customization of a number of settings regarding the search algorithm. First of all, either depth-first or iterative-deepening can be chosen. If the latter is selected, starting depth and incrementation factor can be specified. Moreover, forced backtracking if execution is stuck in a loop can be enabled. Finally, as an additional abortion criterion a number of instructions (or choice points) without finding a *new* solution can be specified after that execution is finally halted.

■ *Coverage*

In this tab, control-flow and data-flow coverage can be customized. More information is provided in Section 5.5.

■ *Test cases*

Directory, package, and naming scheme for test cases are specified in this tab. Furthermore, test case elimination based on coverage information can be configured (cf. Section 5.6).

■ *Logging*

Logging can be switched between HTML and simple text in this tab. Moreover, the maximum number of entries per log file can be specified. If this number is reached, Muggl will continue logging in another, consecutively numbered file. This is specifically important for HTML logging. If logs become too large, Web browser tend to crash while rendering them.

Options can be saved (Muggl then overrides the configuration file) with “Ok” and “Apply”; the

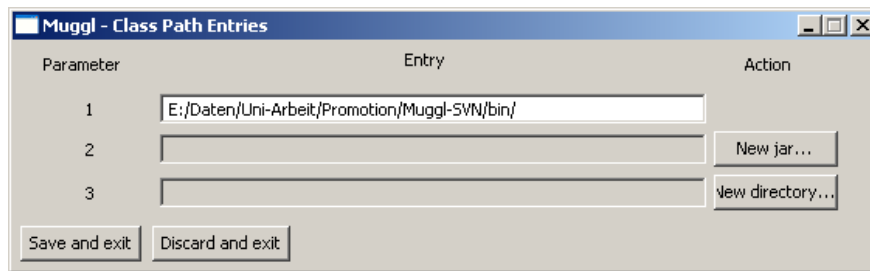


Figure 12: Editing class path entries

first button also closes the window. “Discard” and “Cancel” discard changes; the latter button also closes the window. “Load defaults” resets all values to defaults which are suitable for most programs that test cases should be generated for.

5.4 Class File Inspection

Muggl is meant to be a powerful tool for test case generation. Nevertheless, it is reasonable to provide additional functionality. With supplemental functions, Muggl can also be used for debugging and to better understand Java programs. Providing this function is fostered by the code Muggl already requires for test case generation. For example, Muggl has a parser for `class` files and provides data structures to represent the internal structures of a Java class (see Section 5.10). It is only a short step to visualize them.

The `class` file inspection window (see Figure 13) does exactly that. It displays `class` files in a hierarchical tree structure. `class` files consist of all information required to store an executable Java class along with its methods and all data required by them. This comprises data structures such as the constant pool, exception tables, and many additional attributes. Details are described in the JVM specification [17] and were summarized in Section 3.

With the tree structure, elements of interest can be expanded. This keeps the view easy to comprehend since class files can comprise a high number of fields, methods, and constant pool entries. Inspection of the entry of a method is shown in Figure 14. Some nested entries are expanded. Along with the step-by-step GUI, the class file inspection window is very helpful to better understand how Java programs work and to optimize them.

A distinct feature is Muggl’s capability to write `class` files. Currently, string entries of the constant pool (or their representation as `CONSTANT_Utf8_info` to be precise) (see Figure 15) can be edited. The changed class can then be saved to another file. Changed to constant pool entries directly affect execution in Muggl. Unsaved changes are lost if Muggl is closed or if the class file cache is flushed. For future releases, editing of additional class file constructs could be added.

5.5 Control-Flow and Data-Flow Coverage

Coverage of control-flow and data-flow are important criteria for test case generation. Test cases should cover as many control-flow graph edges and def-use chains as possible—ideally all that are coverable. Muggl tracks control-flow and data-flow and takes it as an abortion criterion. That means, it decides based on coverage information when symbolic execution is finished. Using coverage information is exhaustively discussed in a paper on Muggl already [19]. Therefore, only a short introduction is given here.

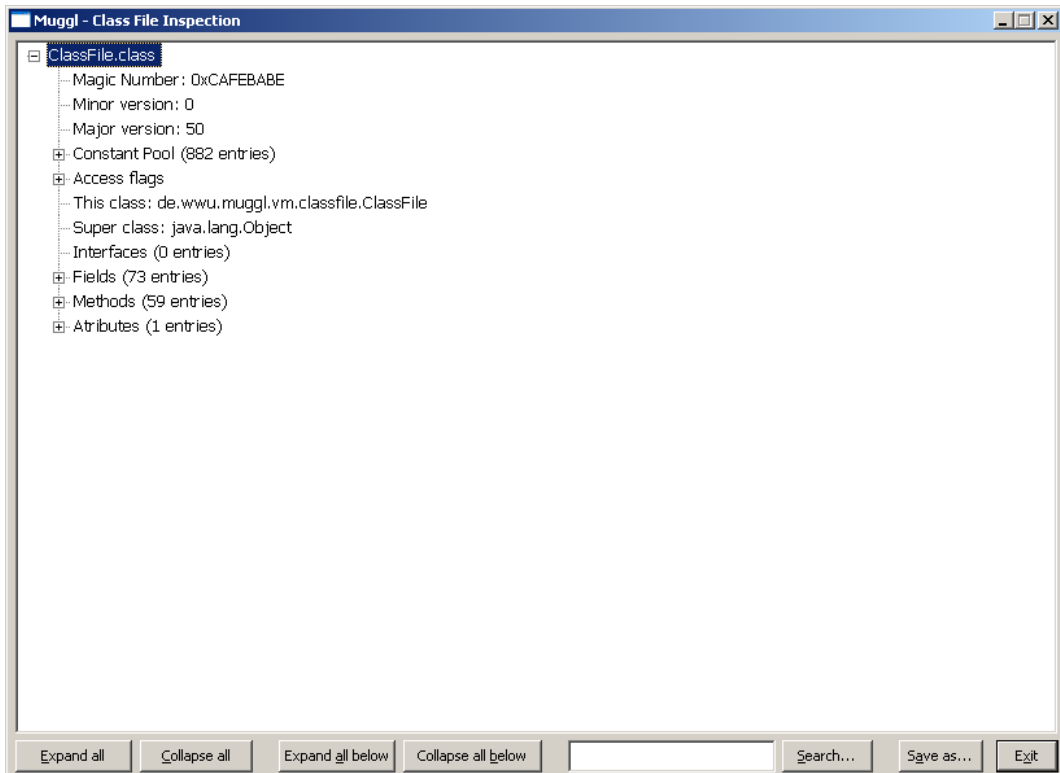


Figure 13: Class file inspection window

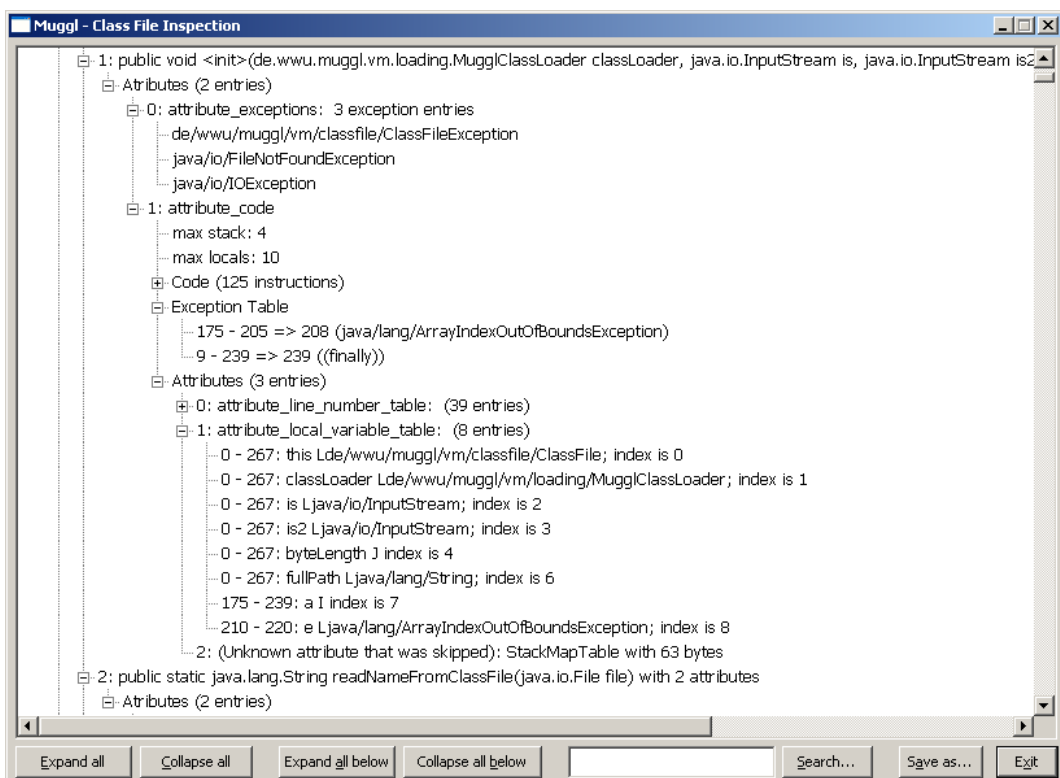


Figure 14: Class file inspection of a method's definition

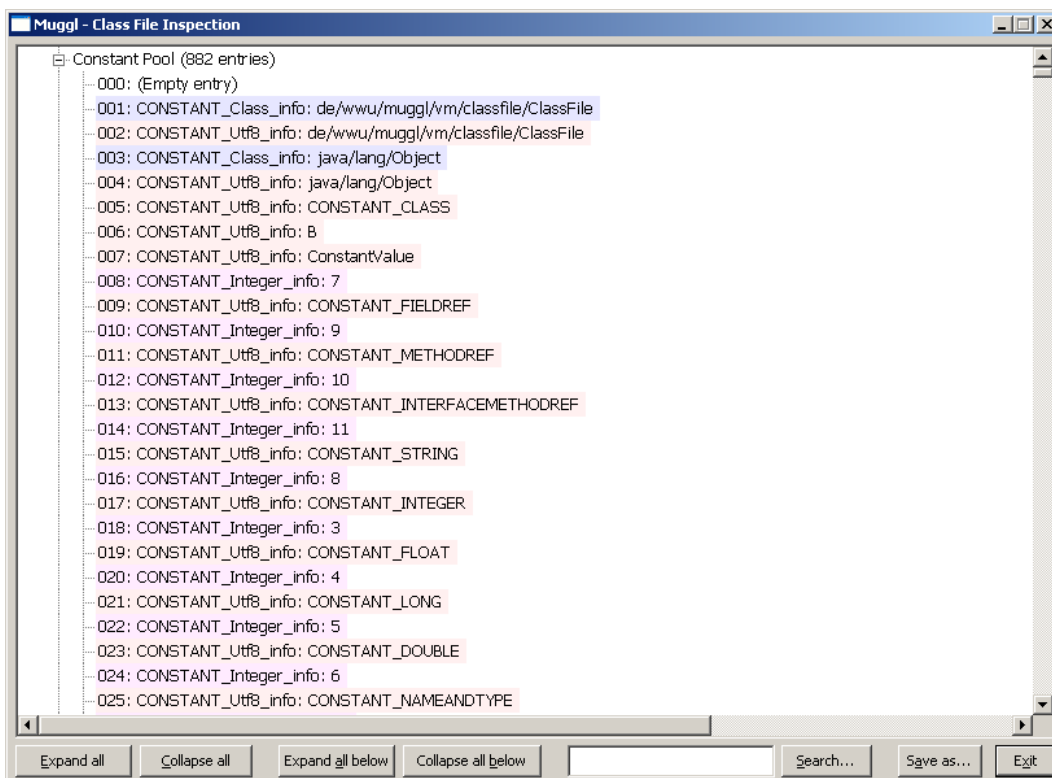


Figure 15: Class file inspection of the constant pool

Muggl is capable of building control-flow graphs and creating def-use chains for any program. Def-use chains are built interprocedurally, which requires a sophisticated approach. They are created statically in order to be able to determine the coverage level reached *during* symbolic execution. Graphs and chains can be determined for the method executed, for any methods of the same class, for any methods of the same package, or for all methods. In general, it is advisable to exclude methods that are known to be free of defects from coverage tracking. This speeds up application. For example, methods from the Java libraries should normally not be included. Configuration is possible using the option dialogue (see Section 5.3).

During symbolic execution, Muggl tracks coverage information. Special care has to be taken of backtracking and restarting execution due to iterative-deepening. Coverage information can be reverted using the trail. Muggl can be configured to abort execution once full coverage of edges and chains has been reached. Theoretically, abortion would also be possible when reaching a threshold value. This is not currently implemented. In general, full coverage is hardly reachable in practice. While def-use chains might be fully covered, some control-flow graph edges remain uncovered in almost any example. They cannot be covered for semantic reasons. For example, by specification on any method invocation, a `java.lang.RuntimeException` or any of its subtypes might be thrown. However, most methods will never throw a `RuntimeException`. Nevertheless, an additional edge for this exception type exists in the statically generated graph.

It has proven to be a good choice to abort execution based on def-use coverage for the initially executed method and any methods from the same package. Abortion based on control-flow coverage has proven impractical; similarly, intraprocedural def-use chains beyond class boundaries become too complicated to be handled efficiently.

Allowing threshold values for coverage in order to abort execution might be a feasible approach. However, means to determine a reasonable threshold have to be found.

5.6 Test Case Elimination

Coverage is not only tracked as a criterion to abort execution. It is also used to eliminate test cases. For each solution, coverage data is saved. When execution is finished and the solution processors is activated, it may invoke test case elimination. Test case elimination can be done based on control-flow or data-flow coverage, or based on both. Since basing it on one criterion only might omit test cases that would detect additional defects, elimination based on both criteria is recommended.

The elimination process is described in [19]. Muggl uses a *greedy* algorithm to select solutions. It selects solutions for inclusion in the set of kept solutions as long as not all coverable control-flow graph edges and def-use chains have been covered by the selected set of solutions. Once such a set is compiled, all other solutions can be discarded. Due to the greedy algorithm, elimination is very fast; even for hundreds of thousands of test cases it only took a few second in test runs. Moreover, for most examples only a very small number of test cases remain—in almost any example we tested it was the optimal number of test cases (despite the heuristic nature of a greedy algorithm).

Up to now, no examples have been found in that non-redundant test cases have been eliminated. Theoretically, there could be solutions that do not further contribute to coverage but still would detect a defect that is not detected by the test cases generated from the selected set of solutions. If such an example is found, another criterion has to be included in the elimination process. More details on test case elimination based on coverage is described in [19].

5.7 Native Wrapper

Realistic (i.e. productively used) Java programs make use of so called native methods. Parts of the Java libraries are not implemented in Java but in platform specific code (usually in C). This code is required to communicate with the operating system and for performance reasons. Typical methods that have a native implementation access files, write messages to the console, or render GUI elements. Since Muggl is implemented in Java and does not have a native interface to C code, it cannot execute native code.

To be able to run programs that invoke native methods, Muggl provides a *native wrapper*. This component detects calls of native methods and handles them. Muggl offers two strategies of dealing with native method calls:

- The first strategy is to invoke *handling code* in Muggl. This is for example done for the often used `System.out` object, which writes messages to the console. Messages are not written to the console but forwarded to Muggl's log file. Another example is `System.arraycopy(Object, int, Object, int, int)`. This methods copies an array or parts of it. It is implemented natively for performance reasons. Muggl forwards calls of this method to a pure Java implementation of `arraycopy`. It is considerably slower than the native implementation. Nevertheless, for all examples yet tested this does not lead to a bottleneck. In particular, symbolic execution requires so many resources that overhead for native execution can be neglected—it is much slower than constant, i.e. normal execution, anyway.
- The second strategy is applied to all native method calls for which no wrapper implementation exist. Muggl converts arguments supplied to native methods from their internal representation into objects compatible with the system's JRE. It then directly invokes the native method using the Java environment it runs on. The result from execution is converted back to a Muggl data structure and used for further execution. This process is convenient and sounds rather simple. However, much code is needed specifically to cover the conversions. Conversion is a time-consuming process.

Therefore, the first strategy is preferable for performance reasons. Nevertheless, in examples yet tested the overhead in symbolic execution mode was negligible. It even was hardly noticeable in normal execution mode but would be a problem for high-performance applications that e.g. throughput streams of data.

Native wrapping currently is only possible for methods from the Java libraries (such as `java.*` and `javax.*`). Theoretically, it could be extended to third-party libraries. The conversion process would not change. However, there has not been a reason for adding such functionality, yet. Many third-party libraries do not consist of native code. Those that require native method calls are infeasible for being tested. An example is the GUI library *Simple Widget Toolkit* (SWT) which extensively uses native methods.

It also has to be noted that native wrapping using the second strategy fails for symbolic variables. While constants used in symbolic execution can be converted, conversion fails for variables. This can only be circumvented by implementing the native methods as wrappers in Muggl. However, not many examples have been found yet for that this limitation applies. Many native methods are called with constant arguments only or do not accept arguments. Therefore, more wraps will be added when needed.

5.8 Dynamic Optimization

While Java code was said to be executed slowly on the first releases of (by then) Sun's JVM, today's virtual machines employ advanced optimization techniques. Due to dynamic optimization, code can even be executed faster than native code written in languages such as C or even assembly language. This becomes possible by taking into account information that only is available at runtime; the resulting optimization lead to faster code execution despite intrinsic overhead of a virtual machine in comparison with directly executed machine code. Therefore, a dynamically optimizing virtual machine is a cheap and convenient alternative to manual optimization of assembly code.

Generating test cases within a reasonable amount of time is essential. Besides optimizing symbolic execution, the JVM of Muggl can be improved. For this, dynamic optimization has been included as an experimental feature. Java bytecode comprises several instructions that are very computationally expensive. In particular, all instructions that fetch elements from the constant pool and method invocations take much longer to execute than instructions such as arithmetic operations or conditional jumps. Most of the computationally intensive operations can be optimized.

The reasons some operations are expensive either is their handling of computationally expensive objects (such as strings), or the need to resolve fields or methods. However, fetching a `String` from the constant pool, or resolving fields or method is done in the same way each time the instruction is executed.⁷ If such instructions are nested in loops, or if they belong to methods that are executed multiple times, execution time can be saved. On their n th execution, the results from computationally expensive actions that have the same result on each execution, are stored. The next time the instruction is executed, it does not need to compute the data but can directly provide it or use it for further computation. With a minimal increase in memory footprint, much execution time can be saved.

In practice, a suitable value for n has to be found. Saving the value creates overhead. Therefore, it would be a waste of execution time and memory to save data if the instruction is not executed another time. However, executing it twice or more before storing the data means that on each call until n , the possible saving of execution time is not realized. From early experiments, values of 2 or even 1 result in the best performance.

⁷This statement holds for most but not all instructions. Methods invocation might change dynamically with an object under consideration. Special care has to be taken to cope with such cases.

Dynamic optimization is implemented by providing additional bytecode instructions that replace instructions suitable for dynamic optimization. On the n th call, they store the calculated value in another instruction and replace themselves with it. In subsequent calls, the optimized instruction with predefined data is executed. While replacing instructions sounds like much effort, it is computationally cheap. In particular, Muggl offers interfaces that can be used to easily write optimized counterparts for existing instructions.

Currently, dynamically optimized instructions exist for `ldc_w`, `ldc2_w`, and `invokestatic`. The first two instructions fetch strings from the constant pool; the third one invokes `static` methods. Additional instructions will be added after further testing.

5.9 Array Generation

Muggl's solver does not have means to symbolically represent arrays. While it is possible to have arrays with symbolic elements, arrays that are symbolic themselves would have a variable length and arbitrary variable or constant elements. This limitation does not hinder execution if arrays are initialized with constant value during execution. However, if arrays are initialized with a logic variable given for the length or if arrays are expected as input parameters, symbolic executing would fail. Therefore, Muggl uses a strategy to provide arrays for symbolic execution.

The basic idea is to execute the program under consideration with arrays of various fix lengths. For this purpose, an *array generation choice point* is set at `xaload` instructions that get values from arrays provided as arguments and at instructions that generate arrays but encounter a variable specifying the length. These choice points initialize a so called *array generator*. It provides an initial array and execution is continued with this array. If backtracking reaches the *array generation choice point*, it provided another array. The elements of non-empty arrays are logic variables if their type is either primitive or a primitive wrapper class.

Array generators follow a distinct strategy. In fact, Muggl has four built-in array generators which can be selected and configured in the options dialogue (cf. Section 5.3). Thus, they are seamlessly integrated. During our tests, we found that in almost any case there is no need to configure a generator to provide a high number of arrays. This merely prolongs execution. Usually, one of the following two attempts is successful:

- Provide several short arrays starting with 1 and with up to about 32 elements.
- Provide several arrays with quickly increasing length. Thus, both very short and very long arrays are tested.

We cannot think of any but of artificial algorithms that would require 2^{31} different arrays to be tested. $2^{31} - 1$ is the maximum size of an array permitted by Java.⁸ To provide arrays for various needs, the total number $n \in \mathbb{N}$ of arrays to be provided can be set. Muggl will backtrack this often to the choice point and request another array from the generator. Backtracking will of course only be done as long as abortion criteria have not been met.

Besides the number of runs, a starting length of $l \in \mathbb{N}$ can be specified. Typically, l will be 1. There might be algorithms that by contract only process array length $l' \in \mathbb{N}^+$ with $l \geq l'$ which can be satisfied by changing l . Furthermore, the generator can be configured to provide one array of zero length and the `null` reference instead of an actual array. According to the JVM specification, arrays of zero length (i.e. empty arrays) are not equal to `null` [17]. Many algorithms will fail if an empty array is provided, and even more cannot handle `null` instead of an array. Therefore, providing both usually yields additional test cases that could identify additional defects.

⁸Arrays's length is given by an `int` value. The maximum positive `int` is $2^{31} - 1$. Counting the zero length array, there are 2^{31} arrays of distinct length.

There are four basic strategies for array generation that determine the length of the generated arrays. Regardless of the strategy, a set $A = \{a_1, \dots, a_n\}$ with arrays $a_i, i \in \mathbb{N}^+$ being the i th array is requested. All strategies provide arrays in order of ascending lengths. No reason could be identified to provide them in another order. The strategies can be described as follows:

Linear

Starting with a length of l , subsequent arrays are created for n times. A linear increment $j \in \mathbb{N}^+$ is used to increase the array length each time an array is requested. Hence, the length of the generated array is $l + (i - 1) * j$ with $l > 0$. Typical sequences of length include 1, 2, 3, ..., 10, 20, 30, ... or 111, 222, 333, This strategy is ideal for testing an algorithm with a couple of short arrays.

Fibonacci

Arrays are generated with a length of f_{l+i} with $l \geq 1$ according to the fibonacci sequence with $f_n = f_{n-1} + f_{n-2}$, $f_0 = 0$ and $f_1 = 1$. If the length determined would cause a 32 bit Java **int** to overflow, a length of `java.lang.Integer.MAX_VALUE` i.e. $2^{31} - 1$ is used. This for example applies to $l = 1 \wedge i > 46$. A sequence with $l = 1$ provides arrays of lengths $\in \{1, 2, 3, 5, 8, 13, 21, 34, \dots\}$.

Exponential

The length of the generated array is $l + 2^{i-1}$ with $l \geq 0$. Again, integer overflow is prevented by generating an array of length $2^{31} - 1$ for $l + 2^{i-1} > 2^{31} - 1$. A typical sequence of length is 1, 2, 4, 8, 16, 32, 64, 128, 256, Due to the nature of exponentiality, this strategy is ideal to generate array to first test an algorithm with a couple of shorter arrays but to also examine its behavior when confronted with very large arrays. The maximum number of arrays generated is 32 (again due to exponentiality).

Ten to the power of x^9

This strategy generates arrays with a length of $l + 10^{i-1}$ with $l \geq 0$. Thus, the maximum number of calls is 10 with any further call generating an array of length $2^{31} - 1$ to prevent an integer overflow. The sequence of length 11 with $l = 0$ is 1, 10, 100, 1 000, 10 000, 100 000, 1 000 000, 10 000 000, 100 000 000, 1 000 000 000, 2 147 483 647.

If the generator has been configured to provide arrays of zero length or `null`, they will be provided first before arrays of non-zero length will be generated. Consequently, the number of total arrays tested is $n + 1$ or $n + 2$ whereas n is the number of arrays provided by the generation strategy.

5.10 Java Class File Representation by Java Classes

The package `de.wvu.muggl.vm.classfile` provides structure to represent Java `class` files. Each Java `class` file is represented by a class `ClassFile` and by a number of classes that implement the interface `ClassFileStructure`. Thereby, class file can be completely represented by Java classes. They are very convenient to use. Moreover, these data structures can as well be used by other applications.

A diagram of the package, basic classes and its subpackages is given in Figure 16. Subpackages are depicted in Figures 17, 18, and 19. As high level views, no method definitions or public class members are given. Muggl's JavaDoc documentation described the Application Programming Interface (API) provided by these classes.

⁹This strategy might be dropped since it is hardly needed with the exponential strategy available.

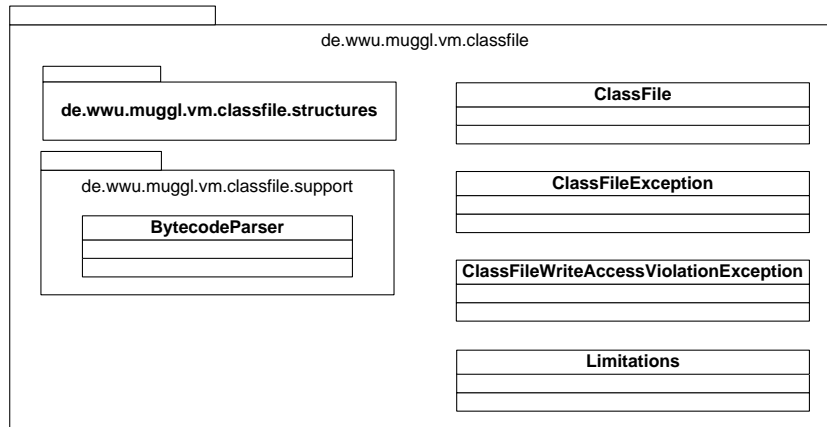


Figure 16: UML class diagram for `de.wwu.muggl.vm.classfile`

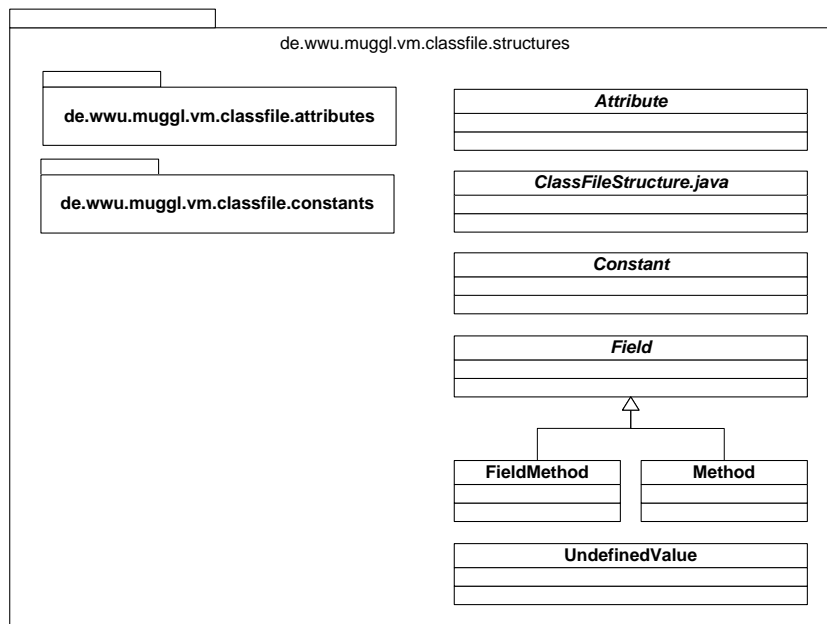


Figure 17: UML class diagram for `de.wwu.muggl.vm.classfile.structures`

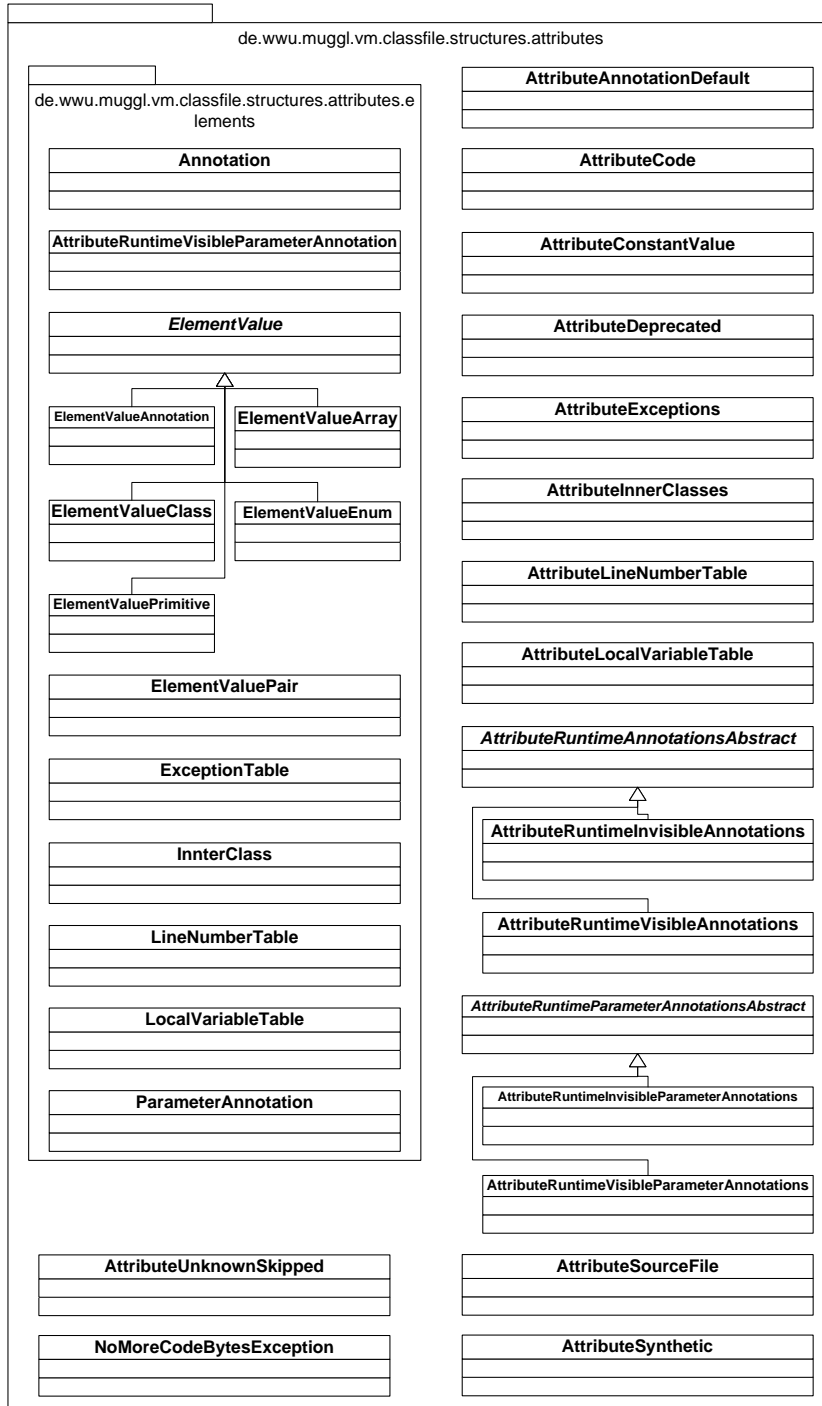


Figure 18: UML class diagram for de.wvu.muggl.vm.classfile.structures.attributes

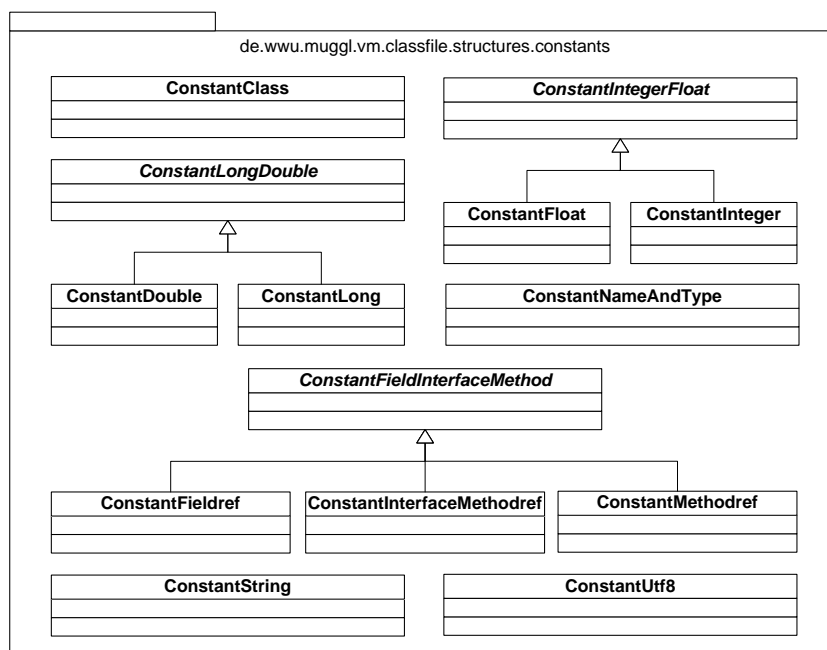


Figure 19: UML class diagram for `de.wwu.muggl.vm.classfile.structures.constants`

6 Experimental Evaluation

In the following three sections, a short summary of experimentation with Muggl is given. Both qualitative and quantitative results are discussed.

6.1 Tested Examples

Muggl has been tested for a variety of examples. It was successfully able to generate test cases for many of them. In particular, it was successful for the following examples:

- A myriad of small examples have been tested. During the development of Muggl, roughly 25 test classes have been created. Each contains up to 20 methods. Whereas the first class contains simplest methods that test the basic symbolic execution, classes become subsequently more complicated. However, all classes contain artificial examples, which were written to evaluate Muggl. Nevertheless, these classes ensured that instructions are implemented correctly and that all basic principles of symbolic execution behave as expected. They underline the general feasibility of the approach
- Several rather simple algorithmic examples have been tested. Algorithms include *binary search*, calculating *fibonacci* numbers, *greatest common divisor*, *mergesort*, *quicksort*, and *approximation of square roots* by Newton's method.
- Muggl has been used in combination with the e-assessment tool EASy [32, 13] in a basic course on programming. It was used for a realistic example. Muggl successfully generated test cases for an *algorithm that blurs images*. Students were asked to implement the blur algorithm without knowing our exemplary solution. They could then request an automatically generated test case for it. In most cases, generation was successful despite differences in implementation. In cases in that generation failed, it could mostly be attributed to the complexity of the student's

solutions. To provide immediate results, a maximum execution time of 5 seconds was chosen. If a solution was too complex, test case generation did not finish within this time.

Some of the examples we tested do not lead to satisfying results, yet. This particularly applies to more complex algorithms or small programs that extensively use sub classes. Tested examples are the *price calculation of Deutsche Bahn*, *binary trees*, and *Kruskal's algorithm*.

A number of examples need further validation since they have only recently been added to the pool of considered programs. This for example includes *swapsort*, calculation of the *dot product of two vectors*, calculation of classes for a *histogram*, *fast Fourier transformation*, and *heapsort*.

Finally, there are some examples for that test case generation currently fails. This specifically applies to some of the student's solutions. Out of approximately 60 solutions uploaded, three were identified for which test case generation failed for no apparent reason and independent of running time. These solutions have to be examined further.

6.2 Qualitative Results

Continuous experimental evaluation is an essential part of Muggl's development. It thereby has led to various findings:

- Muggl's general approach is feasible. Symbolic execution is possible for all bytecode instructions. It can be executed quickly, typically with rates of several thousand instructions per second.
- For many small examples, test cases are found very quickly.
- A low search-depths and restrictive options are feasible in most cases. The number of test cases found by Muggl is much lower than for high search depths and non-restrictive settings. However, after elimination the same number of unique test cases remains.
- For large programs, the state space becomes too large to be explored with the current means.
- Muggl's solver is flexible and reliable but fails in some special situations. For example, the above mentioned price calculation example fails because the solver runs into an infinity loop. Such problems have to be addressed in future versions of Muggl.

6.3 Quantitative Results

A number on quantitative results on Muggl has been published in [19]. An extended overview of these results is given in Table 3.

In general, the number of generated test case is strongly influenced by the settings chosen when running Muggl. The results in Table 3 reflect this. The number of test cases generated is artificially high in some cases. Some of the tested algorithms have been chosen to benchmark test case elimination rather than to benchmark Muggl in general. All examples were run on a system with an Intel Core 2 Duo E6600 CPU at 2,4 GHz and 3 gigabytes of PC2-6400 memory. The operation system was MS Windows XP with Service Pack 3. The Java SDK 1.6.0_07 was used. VM heap size was limited to 512 megabytes with an initial value of 256 megabytes. However, memory was no limitation in either of the examples.

Algorithm	Runtime	Aborted by	Test cases			Elimination time
			found	eliminated	left	
binary search	0.14s	<i>coverage</i>	60	57	3	< 0.05s
greatest common divisor	10.0s	time	62	59	3	< 0.05s
greatest common divisor	30.0s	time	85	82	3	< 0.05s
fibonacci	10.0s	time	2	1	1	< 0.05s
fibonacci	180.0s	time	2	1	1	< 0.05s
mergesort	10.0s	time	201	199	2	< 0.05s
quicksort	10.0s	time	139	136	3	< 0.05s
quicksort	180.0s	time	488	485	3	0.18s
Newton's method	0.19s	<i>coverage</i>	29	24	5	< 0.05s
Deutsche Bahn pricing	10.0s	time	28	24	4	< 0.05s
Deutsche Bahn pricing	180.0s	time	28	24	4	< 0.05s
Deutsche Bahn pricing	300.0s	time	28	24	4	< 0.05s

Table 3: Experimental results for test case generation and elimination.

The following settings have been used:

- Muggl's native wrapper was enabled (cf. Section 5.7).
- Instructions were dynamically reflected by optimized ones (cf. Section 5.8).
- Depth-first iterative-deepening with starting depth of 5 and an increment of 3 was used. *Soft abortion* was set to 200, i.e. the maximum number of iterations per loop was 200 (cf. Section 5.1). This has proven to be reasonable in our tests.
- Coverage was tracked for the initial method and any method it invokes from its own package. Full coverage of control-flow edges and def-use chains immediately aborted execution.
- Test cases were eliminated based on their contribution to both control-flow and to the data-flow coverage.

The result compiled in Table 3 underline the feasibility of our approach. After elimination, the optimal number of test cases remain for almost all examples. Please note that the number is bound to the implementation of our examples; other implementations might need more or less test cases. Even with structural equivalence of the main algorithm, this can e.g. be caused by explicitly throwing exceptions of varying type.

For exemplary illustration, consider *binary search*. Execution was halted after 0.14s since full coverage of control-flow graph edges and def-use chains could be reached. 60 test cases were found of which 57 could be eliminated. Elimination time was below the threshold for a meaningful result—it was performed without noticeable delay and results did not stabilize but always were below $\frac{1}{20}$ of a second. Of the three test cases, two represent usage of the binary search while one represents an exception. In the first one, both the left and the right side of the array are searched through before the position of the searched value is returned. In the second test case, search is unsuccessful. Instead of an array, `null` was supplied in the third case. Consequently, a `NullPointerException` is thrown.

As discussed earlier, abortion criteria are a problem. Only in two examples, execution was aborted due to full coverage. In all other examples, the time limit was reached. Increasing the time limit shows that more test cases are found, but the same number of test cases remains after elimination. Manually inspecting them reveals that they are the expected test cases. In fact, no test cases are missing. Therefore, coverage was not reached for semantic reasons and test case

Algorithm	time limit	Test cases			Eliminated time
		found	eliminated	left	
bin. search	10.0s	13.530	13.526	4	0.13s
bin. search	60.0s	79.676	79.672	4	0.5s
bin. search	600.0s	374.883	374.879	4	3.56s
quicksort	180.0s	1.158	1.154	4	<0.05s

Table 4: Additional experimental results

generation could have been aborted earlier. Nevertheless, it has to be noted that Muggl always found suitable test cases in a very short time.

Two more notable observations can be made. Firstly, no additional test cases are found for calculating *fibonacci* numbers. Two test cases are found of which one is sufficient for testing. This is almost independent of execution time—both test cases are found within fractions of the first second of execution. Obviously, full coverage cannot be met but the algorithm has an unpropitious loop. Consequently, it is impossible to discover all states and thus finish execution. Secondly, the *price calculating* for Deutsche Bahn fails. Only four test cases remain after elimination whereas at least 16, but probably even many more would be expected. Calculation at the time of implementing the example was based on 16 parameters. Regardless of elimination time, only 28 examples are found. Deleting 12 of them is justified—these test cases are redundant and not eliminated by error. After finding 28 solutions, Muggl's solver runs into an infinite loop. This effectively precludes finding more test cases. Optimizing the solver is an ongoing task.

For additional tests, array generation was used (cf. Section 5.9). Deactivating all abortion criteria but the time limit, this led to an extremely high number of test cases which could be used to assess the elimination algorithm's performance. Results are shown in Table 4.¹⁰

These results show that deleting redundant test cases based on their contribution to the global coverage is very efficient. Taking only a few seconds to eliminate a six-digit figure of test cases is sufficient for almost any practical problem. Additionally, the greedy approach well approximates the minimum number of test cases. Subsequential testing asserts this result.

Additional quantitative results are in preparation. They will be presented in future papers along with a comprehensive analysis.

7 Ongoing Research

Research on Muggl is not finished. In fact, the technical base described in this paper suggests that there are many possibilities for extension. A number of features is already in preparation.

7.1 Features in Preparation

Current work on Muggl deals with three additional features, which will be explained in the following. Besides, further examples are being tested and improvements to details are continuously made.

¹⁰The table has been reproduced from [19].

7.1.1 Combination of Logic and Object-Oriented programming paradigms

The dominating programming paradigm is object-orientation. Concepts such as inheritance and encapsulation of structures and behavior [18] provide advantages with respect to maintainability and adaptability [27]. Nevertheless, other programming paradigms are better suited for distinct application domains. For example (constraint) logic languages such as *Prolog* [34] are used for search-problems. Prolog provides a built-in search mechanism [29]. In some cases, other paradigms are even used for developing business software. The functional language *Erlang* [1] is successfully used especially in the telecommunication industry [26, 25, 35]. This observation encourages developing new problem-adequate languages.

Currently, we prepare an approach that combines object-oriented programming in Java with the search capabilities of logic programming concepts such as logic variables, constraint solving, and backtracking. With the symbolic JVM described in this article, we have all means already at hand. What we need to provide is an interface to use them explicitly. We intend to do this in form of a novel programming language.

Our language will preserve the syntax of *Java* and use *Java annotations* to add logic programming concepts. If executed on a *normal* JVM, annotations will have no effect and the program will perform without search. However, if executed on the SJVM, it will use the same concepts we use for generating test cases. Of course, results are not written as a test case file but available as a list during execution.

To implement the programming language, only little changes to the SJVM will be needed. Extensive work has to be spent on the annotation framework. It does not need to provide a high number of annotations but be conceptionally sound.

7.1.2 Data Structure Generators

Array generators, as described in Section 5.9 are a necessity since it is impossible to symbolically represent arrays of variable length in Muggl. It is possible to represent object structures of arbitrary complexity. Since only variables of primitive type or their wrapper classes need to be symbolical, building object structures and using them for symbolic execution in theory does not cause problems. However, it often becomes impractical.

Consider an algorithm for inserting and deleting entries from a tree (say, an AVL tree) that is to be tested. Due to Muggl's precise operation, it will rarely encounter valid AVL trees during symbolic execution. In fact, most entries will be varying object structures with symbolic values that are *not* AVL trees. Even a simple tree usually consists of a two digit number of objects which reference each other. Without question, both `insert` and `delete` should be tested on invalid structures as this might yield additional test cases. Keeping in mind the infinite number of possible object structures, the algorithm will hardly be tested with valid trees and thus it will take a very long time until test cases that actually check the insertion and deletion functionality. For practical usage, this is very inefficient and thus unacceptable.

Similarly to using array generators, data structure generators could be used. Whenever a complex object structure as an input parameter is expected, an appropriate choice point is set. It initializes an object structure generator which provides objects on subsequent calls. These objects should comprise both invalid ones and valid ones. In contrast to the *normal* approach, there should be a high number of valid object structures provided.

There are three challenges with regard to data structures generators:

- An appropriate framework has to be implemented and built into Muggl. It needs to be reflected in the GUI. Admittedly, it can be similarly implemented as the array generation framework.
- Adequate generators have to be found. It would be possible to provide generators for the most common object-oriented data structures but ideally a way of having parameterizable generators should be found.
- User support is needed. There have to be guidelines on how to use generators and when to apply them. Probably, Muggl could also suggest generator usage when applicable.

Work on the generator framework is advanced; implementing actual generators is at an initial state. A conference paper on generation is upcoming.

7.1.3 Solver Improvements

Experimentation revealed that despite the general feasibility of Muggl's solver, in some cases it fails to successfully finish solving (cf. Section 6). Therefore, ongoing research also includes improvements to the solver. There are several ideas for this:

- Currently, solving has to be repeated if constraints are added or removed. Ideally, the solver would keep track of partial solutions and only need to calculate what modifications have to be made to solutions if constraints are added or removed.
- Additionally, it can be tried to allow incremental processing in the solver. Especially backtracking on level of the solver will be a challenge but might increase execution speed.
- To better support the two ideas mentioned above, a normalization algorithm could be applied to the constraint system. It would remove redundant or unnecessary constraints. However, it has to keep track of adding and removing constraint. A constraint considered redundant might be required again after backtracking.
- The solver could be parallelized in order to solve constraint in parallel. This requires to find ways to disjointly treat subsets of the currently active constraints.
- Currently, it is hardly possible to monitor the solver's state. Muggl reports the total time required for solving but there are no key figures generated on substeps of solving. Logging either is disabled or leads to very verbose outputs. Thus, more convenient ways of keeping track of what the solver does have to be found.

Work on improving the solver has already started.

7.2 Future Work

Future work on Muggl comprises work on the three features described in the previous section along with a number of smaller tasks. In particular, more examples have to be implemented and tested. We intend to both test well-known algorithms and an increasing number of examples from other programs. With each example tested, either Muggl's effectivity is proven or hints for needs of further improvements are found. This circle of testing Muggl and improving it with regard to the deficits found will be continued until it is capable of generating test cases for programs from software development companies.

There is a couple of long-term goals. For example, the SJVM could be parallelized. It would then

process more than one path through a program's search tree at once. Even with an parallelized solver, this would increase execution speed for programs that are not solving-intensive but that have a very high number of states to be discovered. Dynamic optimization should be extended to comprise as many instructions as possible. Probably, further means of analyzing execution and dynamically applying optimizing techniques can be added. Furthermore, Muggl could be enabled to write test cases for other programming languages that are compiled to bytecode for the JVM. Finally, adding threading and in particular finding a way of dealing with multiple threads during symbolic execution is a task that might be addressed in future.

Future versions of Muggl should better support users. As long as Muggl is an experimental prototype, manually finding the optimum configuration for each example is unproblematic. In fact, it even helps to learn about Muggl's behavior and to improve it. However, for commercial use Muggl should be equipped with heuristically reasonable assumptions for settings. It probably could rely on a set of standard setting and adjust them after analyzing a `class` file. Moreover, it could detect if settings yield unsatisfiable results and propose (or even automatically change to) more adequate settings.

An additional and not less important future task is to assess the abortion criteria Muggl uses. Experimentation has shown that abortion is a problem. Ideally, Muggl should halt the SJVM once all test cases required to test a program have been found. However, this has not yet been possible—neither with “scientific” i.e. precise criteria such as full control-flow edge coverage nor with impromptu criteria such as maximum running time. A mixture of existing criteria, thresholds for the criteria currently used, and completely new criteria might be adequate. However, finding a satisfying solution remains a future task. Even if maximum runtime remains the ultimate criterion, Muggl should be able to approximate how long it will take to find a *suitable* set of test cases.

8 Conclusion

In this working paper, a comprehensive overview of Muggl has been given. Muggl is a tool for automated test case generation based on symbolic execution. The paper sketched related work; there is only a small number of comparable approached. Then, the Java virtual machine specification was briefly explained. In Section 4, the basic working principles of Muggl were sketched and illustrated by examples. In particular, execution principles along with constraint solving and test case generation were shown, and the architecture of Muggl highlighted.

Following the basics, Muggl's distinct features were explained in detail. For this paper, nine characteristics and functions of Muggl have been chosen that are specifically notable. Afterwards, results from experimental evaluation were discussed. Even though the number of formally documented experiments is low, the results are promising. They align with further experimentation that continuously accompanies the development of Muggl. Finally, ongoing research was sketched. There are three upcoming extensions, namely a logic object-oriented programming language, data structure generators, and improvements to the solver. Besides that, much more future work is planned.

With its immense base of code and the features embedded in it, Muggl is perfectly fit to be further extended. However, we will not only extend it but also improve existing functions. The aim is to have a versatile yet easy-to-use tool for TCG. The status of Muggl described in this paper can be seen as a leap towards this aim.

References

- [1] J. Armstrong. The development of Erlang. In *Proceedings ICFP '97*, pages 196–203, New York, USA, 1997. ACM.
- [2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.
- [3] M. Bramer. *Logic Programming with Prolog*. Springer, Secaucus, NJ, USA, 2005.
- [4] L. de Moura and N. Björner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *LNCS*, pages 337–340. Springer Berlin, 2008.
- [5] K. Doets. *From Logic to Logic Programming*. MIT Press, Cambridge, MA, USA, 1994.
- [6] J. Doyle and C. Meudec. IBIS: an Interactive Bytecode Inspection System, using symbolic execution and constraint logic programming. In *PPPJ '03: Proceedings*, pages 55–58, New York, 2003.
- [7] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, 1999.
- [8] R. A. Eyre-Todd. The detection of dangling references in C++ programs. *ACM Lett. Program. Lang. Syst.*, 2(1-4):127–134, March 1993.
- [9] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and practice of declarative programming*, pages 63–74, New York, NY, USA, 2007. ACM.
- [10] S. Fischer and H. Kuchen. Data-flow testing of declarative programs. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional programming*, pages 201–212, New York, NY, USA, 2008. ACM.
- [11] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.
- [12] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Software Engineering Notes*, 23(2):53–62, 1998.
- [13] S. Gruttmann, D. Böhm, and H. Kuchen. E-assessment of Mathematical Proofs: Chances and Challenges for Students and Tutors. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 05, CSSE '08*, pages 612–615, Washington, DC, USA, 2008. IEEE Computer Society.
- [14] S. Gupta. *Pro Apache Log4j*. Apress, Berkely, CA, USA, 2nd edition, 2005.
- [15] C. Lembeck, R. Caballero, R. A. Mueller, and H. Kuchen. Constraint solving for generating glass-box test cases. In *Proceedings WFLP '04*, pages 19–32, 2004.
- [16] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum-Akademischer Verlag, Berlin, 2nd edition, 2009.

- [17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.
- [18] K. C. Loudon. *Programming Languages: Principles and Practice*. Wadsworth, Belmont, CA, USA, 1993.
- [19] T. A. Majchrzak and H. Kuchen. Automated Test Case Generation based on Coverage Analysis. In *TASE '09: Proceedings of the 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 259–266. IEEE Computer Society, 2009.
- [20] T. A. Majchrzak and H. Kuchen. Automatische Testfallerzeugung auf Basis der Überdeckungsanalyse. In M. Hanus and B. Brassel, editors, *Technischer Bericht des Instituts für Informatik Nr. 0915: 26. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, pages 14–25, Bad Honnef, Germany, May 2009. Christian-Albrechts-Universität Kiel.
- [21] T. A. Majchrzak and H. Kuchen. IHK-Projekt Softwaretests: Auswertung. In *Working Papers*, number 2 in Working Papers. Förderkreis der Angewandten Informatik an der Westfälischen Wilhelms-Universität Münster e.V., 2010.
- [22] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2):81–96, 2001.
- [23] R. A. Mueller, C. Lembeck, and H. Kuchen. GlassTT – a Symbolic Java Virtual Machine Using Constraint Solving Techniques for Glass-Box Test Case Generation. *Arbeitsbericht Universitaet Muenster, Dept. of Information Systems*, 102, 2003.
- [24] R. A. Mueller, C. Lembeck, and H. Kuchen. Generating Glass-Box Test cases using a symbolic virtual machine. In *Proceedings IASTED SE 2004*, 2004.
- [25] T. Nagy and A. Nagyné Víg. Erlang testing and tools survey. In *Proceedings ERLANG '08*, pages 21–28, New York, NY, USA, 2008. ACM.
- [26] J. H. Nyström. Productivity gains with Erlang. In *Proceedings CUFPP '07*, New York, NY, USA, 2007. ACM.
- [27] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.
- [28] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarajan. A Survey on Automatic Test Case Generation. *Academic Open Internet journal*, 15, 2005.
- [29] P. H. Salus. *Functional and Logic Programming Languages*. Sams, Indianapolis, IN, USA, 1998.
- [30] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In *2nd International Conference on Tests and Proofs*, pages 134–153, April 2008.
- [31] N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software*, 23(4):38–47, July 2006.
- [32] C. A. Usener, S. Gruttmann, T. A. Majchrzak, and H. Kuchen. Computer-Supported Assessment of Software Verification Proofs – Towards High-Quality E-Assessments in Computer Science Education. In *Proceedings of the 2010 International Conference on Educational and Information Technology (ICEIT)*, pages 115–121. IEEE Computer Society, 2010.

- [33] E. Wallmüller. *Software-Qualitätsmanagement in der Praxis*. Hanser, München, 2nd edition, 2001.
- [34] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog – the language and its implementation compared with Lisp. In *Proceedings Symposium on AI and progr. languages*, pages 109–115, New York, NY, USA, 1977. ACM.
- [35] U. Wiger. 20 years of industrial functional programming. In *Proceedings ICFP '04*, pages 162–162, New York, NY, USA, 2004. ACM.
- [36] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *11th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 365–381. Springer, 2005.

Working Papers, ERCIS

- Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004. October 2004.
- Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. March 2005.
- Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. June 2005.
- Nr. 4 Vossen, G.; Hagemann, S.: From Version 1.0 to Version 2.0: A Brief History Of the Web. January 2007.
- Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. July 2007.
- Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.
- Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.
- Nr. 9 Majchrzak, T. A.; Jakubiec, A.; Lablans, M.; Ückert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration



ERCIS – European Research Center for Information Systems
Westfälische Wilhelms-Universität Münster
Leonardo-Campus 3 ■ 48149 Münster ■ Germany
Tel: +49 (0)251 83-38100 ■ Fax: +49 (0)251 83-38109
info@ercis.org ■ <http://www.ercis.org/>