

Bytecode 2010

5th workshop on Bytecode Semantics, Verification, Analysis and Transformation

Preface

This volume contains the proceedings of the Bytecode 2010 workshop, the Fifth Workshop on Bytecode Semantics, Verification, Analysis and Transformation, held in Paphos, Cyprus, on the 27th of March 2010 as part of ETAPS 2010.

Bytecode, such as produced by e.g. Java and .NET compilers, has become an important topic of interest, both for industry and academia. The industrial interest stems from the fact that bytecode is typically used for the Internet and mobile devices (smartcards, phones, etc.), where security is a major issue. Moreover, bytecode is device-independent and allows dynamic loading of classes, which provides an extra challenge for the application of formal methods. In addition, the unstructuredness of the code and the pervasive presence of the operand stack also provide extra challenges for the analysis of bytecode. This workshop focuses on the latest developments in the semantics, verification, analysis, and transformation of bytecode; encompassing both new theoretical results and tool demonstrations. There were 8 submissions. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept 5 papers. The programme also includes 4 invited talks: Francesco Logozzo, Mark Marron, Matthew Parkinson and Fausto Spoto.

As the workshop chair, I would like to thank members of the program committee and all anonymous referees, for their hard work, particularly as much of this had to be done over their Christmas holidays.

David Pichardie

Programme Committee

David Aspinall
Stephen Chong
Alessandro Coglio
Pierre Crégut
Samir Genaim
Rene Rydhof Hansen
Bart Jacobs
Gerwin Klein
Victor Kuncak
Patrick Lam
Francesco Logozzo
Matthew Parkinson
David Pichardie
Fausto Spoto

External Reviewers

Frédéric Dabrowski
Kenneth MacKenzie
Martin Toft
SongTao Xia

Table of Contents

Invited Talks

1. *Francesco Logozzo*. Language-agnostic Contract specification and checking with CodeContracts and Clousot..... 3
2. *Mark Marron*. Spec-tacular: heap assertions for .net bytecode..... 4
3. *Matthew Parkinson*. The design of jStar 9
4. *Fausto Spoto*. Static Analysis of Java: from the Julia Perspective .. 10

Regular Research Papers

5. *Jacek Chrząszcz, Patryk Czarnik and Aleksy Schubert*. A dozen instructions make Java bytecode..... 11
6. *Jaroslav Bauml and Premek Brada*. Reconstruction of Type Information from Java Bytecode for Component Compatibility 26
7. *Philippe Wang, Adrien Jonguet and Emmanuel Chailloux*. Non-Intrusive Structural Coverage for Objective Caml 41
8. *Michael Eichberg and Andreas Sewe*. Encoding the Java Virtual Machine's Instruction Set 56

Tool Demo Papers

9. *Jevgeni Kabanov*. JRebel Tool Demo..... 71

Language-agnostic Contract specification and checking with CodeContracts and Clousot

Francesco Logozzo

Microsoft Research, USA

Abstract

CodeContracts allow specifying contracts (preconditions, postconditions and object invariants) in a language agnostic form. Contracts are expressed as calls to static methods of the homonymous class included in .NET 4.0. They are serialized and persisted as bytecode, achieving language agnosticism. Runtime checking is performed via bytecode rewriting. The static checker, Clousot, directly analyzes the bytecode to validate the user-provided contracts as well as the absence of common runtime errors. Clousot is based on abstract interpretation, and as such it can presents a higher level of automatism (e.g. for the inference of loop invariants, or postconditions) with respect to similar tools. Also, it scales up to large codebases thanks to the use of new numerical abstract domains and adaptive techniques (which I will sketch in the talk).

Spec-tacular: heap assertions for .net bytecode

(Extended Abstract)

Mark Marron¹

IMDEA Software, Spain

1 Background

The widespread adoption of object-oriented and memory managed programming languages has lead to programs that make extensive use of rich pointer structures and that use data (objects) as a central part of organizing the program structure. Thus, understanding the memory state of the program is an increasingly critical part of understanding the behavior of the program (for both the human programmer and automated tools that transform or analyze the program). This increasing difficulty (and importance) of understanding the behavior of a program is occurring at the same time as the growing need to take advantage of parallel hardware (plus complex memory hierarchies), and decreasing tolerance for software defects. This confluence of events is making the development of practical heap analysis tools an increasingly important area of research.

While there is a large body of work on heap analysis techniques, ranging from simple points-to analyses [2,13,23,28] to very powerful shape analysis techniques [1,4,12,24,25], there has been a relatively small amount of work on analysis that fall in this middle ground of this spectrum (producing relatively precise results while remaining computationally tractable). Some of the more notable work in this area include work done on identifying data structures on the heap [5,15,17], and work on pair/set sharing [22,26], heap constancy [6,11,15], nullness [14,27], and escape [10,16,29] information. Our work is focused on developing a rich model for these properties (and a number of others) that can precisely identify data structures on the heap (similar to the work in [15] but with a more precise model) and then track, sharing, shape, nullity, use/mod, and object lifetime information on the set of data structures identified by the region analysis (thus tracking the information with a finer resolution than previous approaches).

In this extended abstract we focus on the description of commonly useful heap properties which we can extract from the shape analysis results and that can be inserted into

¹ Email: mark.marron@imdea.org

the program source as pre/post conditions. Our objective, in the *Spec-tacular* specification tool, is not to produce the most detailed conditions possible but instead to produce a rich set of pre/post annotations that are useful to the developer and other analysis tools without overwhelming the end user with (often un-needed) details inferred by the underlying shape analysis.

2 Spec-tacular Tool

The *Spec-tacular* tool is a static analysis that integrated with Visual StudioTM as an addin. It runs on the .net IL that is produced for a C# program and then inserts select inferred heap assertions back into the source code as method pre/post conditions (and class invariants). The goal of this tool is to export a set of simple but commonly useful properties back into the source code via *Code Contracts* [8] so that they can be used as documentation for the developer and by other analysis tools such as *Clousot* [7]. With this goal in mind we will only export a fraction of the information that can be inferred by the analysis tool to avoid overwhelming the user with information.

We consider a program state to be composed of local and argument variables, static fields, variables on the call stack, heap objects (*Obs*), and references (root references and pointers stored in the heap, *Refs*). In the following definitions we use the notation $r \rightarrow o$ to indicate that the reference parameter r points to object o (and similarly the notation $o \xrightarrow{p} o'$ to indicate that the object o refers to object o' via pointer p). We use the notation $o \leadsto o'$ to denote that there is a non-empty path of references $\langle p_1 \dots p_k \rangle$ in the concrete heap that starts at o and leads to o' . Similarly we use the notation $r \leadsto o'$ to denote that \exists an object o and path s.t. the root reference $r \rightarrow o \wedge o \leadsto o'$.

We note that the set of extracted heap properties illustrate one of the advantages of explicitly modeling the data structures on the heap. For example in the, *Constness* property we can not only determine if a parameter is constant or not as a binary concept but we can give more precise information on partial constness (i.e. the entire heap reachable from the variable may not be constant but the object itself may be constant and we can distinguish these cases, or at even a finer level of detail if desired).

Argument Properties.

The first category of properties covers standard information on sharing, nullity, and constness of the heap based argument variables.

Nullity. Given an argument variable r we generate the following nullity assertions on the argument variable:

- $NonNull(r) \Leftrightarrow \exists o \in Obs \text{ and } r \rightarrow o.$
- $Null(r) \Leftrightarrow r = null.$

Sharing. Given two argument variables r and r' we generate the following assertions on the sharing relations between them:

- $Alias(r, r') \Leftrightarrow may \exists o \in Obs \text{ s.t. } r \rightarrow o \wedge r' \rightarrow o.$
- $Reachable(r, r') \Leftrightarrow may \exists o \in Obs \text{ and a path s.t. } r \leadsto o \wedge r' \rightarrow o.$

Ownership. Given an argument variable r we generate the ownership assertion on the parameter that holds whenever it is *non-null*:

$Owner(r)$ where r points to object $o \Leftrightarrow \nexists$ any other reference r' (argument variable, variable on the call stack, or static field) s.t. $r' \rightarrow o$ or there is a path where $r' \rightsquigarrow o$.

Constness. Given an argument variable r we generate the following constness assertions on the argument variable:

- $NotMod(r) \Leftrightarrow \forall o \in Obs$ s.t. $r \rightarrow o$ or on some path where $r \rightsquigarrow o$ no fields of o are modified during the method call.
- $NotModImm(r) \Leftrightarrow r = null \vee$ for the single $o \in Obs$ s.t. $r \rightarrow o$ no fields of o are modified during the method call.

Container Argument Properties.

The second category is specific to parameters that are container types (Arrays, Lists, and Sets). Often information about the contents of these container arguments is critical to understanding the behavior of a method and so the analysis (as well as the assertion output) treats them specially.

Empty. Given a container argument variable r_c we generate an assertion on the size of that holds whenever r_c is *non-null* and there is a container $o_c \in Obs$ s.t. $r_c \rightarrow o_c$:

- $Empty(r_c) \Leftrightarrow o_c.Count = 0$.
- $NonEmpty(r_c) \Leftrightarrow o_c.Count > 0$.

Sharing. Given a container argument variable r_c we generate an assertion to indicate if the container may have duplicate entries that holds whenever r_c is *non-null* and there is a container $o_c \in Obs$ s.t. $r_c \rightarrow o_c$:

$$UniqueEntries(r_c) \Leftrightarrow \forall p \in o_c \text{ either } p = null \text{ or } |\{o \mid p \in o_c \wedge o_c \xrightarrow{p} o\}| = 1.$$

Null Entry. Given a container argument variable r_c we generate an assertion that holds whenever r_c is *non-null* and there is a container $o_c \in Obs$ s.t. $r_c \rightarrow o_c$, and indicates if the container may contain `null`:

$$EntriesNonNull(r_c) \Leftrightarrow \forall p \in o_c, p \neq null.$$

Return Value Properties.

The final group of assertions that the analysis extracts are related to the return value from a method (if the return value is a heap reference). As with most other approaches we assume that a special name exists, we use v_{ret} , for referring to the return value.

Nullity. Given the return variable we generate a nullity assertion:

- $NonNullRet() \Leftrightarrow \exists$ object $o \in Obs$ and $v_{ret} \rightarrow o$.
- $NullRet() \Leftrightarrow v_{ret} = null$.

Freshness. Given the return variable we generate several assertions to specify if the return value (or some part of the structure it refers to) was freshly allocated in the method call:

- $AllRetFresh() \Leftrightarrow \forall o \in Obs$ s.t. $v_{ret} \rightarrow o$ or on some path $v_{ret} \rightsquigarrow o$, and o was allocated in the method call.
- $MustRetOnlyFresh() \Leftrightarrow r = null$, or for the single $o \in Obs$ where $v_{ret} \rightarrow o$, and o must have been allocated in the method call.
- $MayRetOnlyFresh() \Leftrightarrow r = null$, or for the single $o \in Obs$ where $v_{ret} \rightarrow o$, and o may have been allocated in the method call.

Relation to Parameters. Given the return variable we generate several assertions to specify how it may be connected to the argument parameters:

- $RetAlias(r) \Leftrightarrow may \exists o \in Obs \text{ and argument variable } r \text{ s.t. } r \rightarrow o \wedge v_{ret} \rightarrow o.$
- $RetReachable(r) \Leftrightarrow may \exists o \in Obs \text{ s.t. } (r \rightarrow o \vee r \rightsquigarrow o) \wedge v_{ret} \rightarrow o.$

3 Model and Analysis

As mentioned in the previously, the exported properties (to avoid overwhelming the user with information) are a small subset of the information actually inferred by the model and analysis. As an example of this reduction, the analysis internally computes enough information to output a complete set of access paths, similar to the paths from [9], from the parameters indicating which objects are shared, as well as which are used and modified. While this information may be useful to other analysis tools it is too much for the programmer to wade through.

For complete discussions of what properties are computed by the underlying analysis we provide references to the most relevant publications from our work on the general heap analysis problem. The general decomposition of the heap into related data structures is presented in [18] which describes the heuristics used to identify recursive and composite data structures on the heap. The techniques used to determine sharing between variables, data structures, and within collections is the focus of the work in [20]. Techniques for tracking the use and modification of heap based objects are explored in [21] which describes techniques for tracking object (data structure) identities across method calls and a field sensitive method for determining which fields/objects are used/modified in a given statement or method call. The techniques for tracking nullity and fresh allocation (escape) are currently in the process of publication. The static analysis is currently able to handle the majority of the managed .net bytecode and can analyze programs up to 15KLoc in a few minutes and a 100-200MB of memory (which is exceptionally fast for shape analysis techniques and is sufficient for module level analysis). Our work on interprocedural dataflow analysis to enable the flow-sensitive, context-sensitive interprocedural analysis of large programs for this (and other rich heap domains) is covered in [19].

4 Future Work

Our experience with the prototype indicates that it is able to consistently infer interesting predicates. We believe that these assertions are valuable from the standpoint of program documentation to aid developers and, even in the restricted form described here, can be very useful to other analysis and optimization tools. Based on the positive results with our prototype implementation of the analyzer we are currently in the process of implementing a full featured version of the analysis based on the CCI [3] framework.

References

- [1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O’Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
- [2] M. Berndt, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, 2003.
- [3] Common Compiler Infrastructure. <http://ccimetadata.codeplex.com>.

- [4] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
- [5] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
- [6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.
- [7] Clousot. <http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx>.
- [8] Code Contracts. <http://research.microsoft.com/en-us/projects/contracts/>.
- [9] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *PLDI*, 1994.
- [10] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC*, 2000.
- [11] S. Genaim and F. Spoto. Constancy analysis. In *FT/JP*, 2008.
- [12] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
- [13] M. Hind. Pointer analysis: haven't we solved this problem yet? In *ISSTA*, 2001.
- [14] L. Hubert. A non-null annotation inferencer for Java bytecode. In *PASTE*, 2008.
- [15] C. Lattner and V. Adve. Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.
- [16] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.
- [17] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.
- [18] M. Marron, D. Kapur, and M. Hermenegildo. Identification of logically related heap regions. In *ISMM*, 2009.
- [19] M. Marron, O. Lhoták, and A. Banerjee. Scalable interprocedural heap analysis: A pragmatic approach. In *(In Submission)*, 2010.
- [20] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
- [21] M. Marron, D. Stefanovic, D. Kapur, and M. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *LCPC*, 2008.
- [22] M. Méndez-Lojo and M. V. Hermenegildo. Precise set sharing analysis for Java-style programs. In *VMCAI*, 2008.
- [23] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 2002.
- [24] S. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
- [25] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
- [26] S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, 2005.
- [27] F. Spoto. Nullness analysis in boolean form. In *SEFM*, 2008.
- [28] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
- [29] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, 1999.

The design of jStar

Matthew Parkinson

University of Cambridge, UK

Abstract

jStar is a program verifier for Java that uses separation logic. In this talk I will present an overview and demonstration of the tool. I will then go into some of the design choices that aim to make jStar a platform for developing other research tools.

Static Analysis of Java: from the Julia Perspective

Fausto Spoto

University of Verona, Italy

Abstract

Verification of real software is recognised as an important topic in software development. Automatic software analysis tools for Java play a prominent role there. They are on main street since a long time now and keep improving every year. We discuss here some of their distinctive aspects, such as genericity, need of annotations, modularity, correctness, precision, real-time and scalability, with some focus on the Julia tool for static analysis of Java bytecode. We then discuss their most notable limitations and their perspective, identifying multi-threading as both the nightmare of next generation tools and a promise for better scalability and efficiency.

A dozen instructions make Java bytecode¹

Jacek Chrząszcz² Patryk Czarnik³ Aleksy Schubert⁴

*Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw
Poland*

Abstract

One of the biggest obstacles in the formalisation of the Java bytecode is that the language consists of 200 instructions. However, a rigorous handling of a programming language in the context of program verification and error detection requires a formalism which is compact in size. Therefore, the actual Java bytecode instruction set is never used in the context. Instead, the existing formalisations usually cover a ‘representative’ set of instructions. This paper describes how to reduce the number of instructions in a systematic and rigorous way into a manageable set of more general operations that cover the full functionality of the Java bytecode. The factorisation of the instruction set is based on the use of the runtime structures such as operand stack, heap etc. This is achieved by presentation of a formal semantics for the Java Virtual Machine.

Keywords: bytecode, semantics

1 Introduction

The transfer of programs from one party to the other raises the problem of security of its execution on the receiver’s side. Therefore it is desirable to provide means to guarantee certain computational properties of the code in the form it travels from the developer to the consumer. Java bytecode language (JVML in short) is one of the most popular formats for a code that travels in the Internet and the security of its execution has already caused practical problems (see [5,8]) which go beyond the abilities to control the execution by means of Java sandboxing. One of the possible ways to overcome the problems is to provide a precise mathematical model for the language, then prove properties of the programs using the model and supply the travelling program with additional information that will make it possible to reconstruct the proof efficiently on the code consumer’s side.

¹ This work was partly supported by Polish government grant N N206 493138.

² Email: chrzaszcz@mimuw.edu.pl

³ Email: czarnik@mimuw.edu.pl

⁴ Email: alx@mimuw.edu.pl

Several formal semantics were proposed for the JVMML including the most notable ones: [1,7,11,13,14,15,16]. These formulations suffer from one of two problems—either they provide a formal semantics of (almost) all 200 bytecode instructions⁵ or they choose a subset of the instructions that represents most of the interesting features. The drawback of the former option is that the formalisation in this case is very difficult to operate with as most of the proofs have to be done by induction on the structure of programs. Therefore the latter option is more often followed by researchers, but then the particular choice of instruction representatives is often not related to the actual instructions of the bytecode and is presented with very little discussion on the issue of the correspondence of the actual instructions to the ones in the model. The current paper provides the missing discussion and divides the instructions into groups that follow the same pattern of access to the JVM runtime structures (such as heap, operand stack etc.). For example, all load instructions are grouped together, all jumps, including the subroutine ones (`jsr` and `ret`), but also `*aload`, `getfield`, `checkcast` and `instanceof` form a single group as they all access the heap and (possibly) put something on the operand stack or raise an exception. In this way we obtain a factorisation of the whole set of the JVMML instructions to 12 items. The actual lists of instructions can be found in [3].

We believe that it is crucial to come up with a formalisation that is based on a small number of instructions as then it is much easier to demonstrate the properties of the language itself—many proofs for such a language are done by induction on the structure of possible programs. If the number of instructions is limited then the number of cases to consider in such a proof is small. This is the main reason why ventures such as EML [10], where the number of semantical rules reaches several hundred, failed to develop metatheoretical properties, while such as Coq module system [2] succeeded with this regard. Moreover, it is a standard compiler design technique to establish a small language that makes easy design of optimisation techniques. Examples of such languages for Java and its bytecode include BAF, Jimple and Grimp [17] as well as BIR [6].

Moreover, our rigorous consideration gives the opportunity to present what are the instructions that really cover the whole spectrum of bytecode behaviours. We are aware that for certain properties of the JVMML a slightly different set of instructions would be more convenient (e.g. the proofs for interval static analysis require access to the actual arithmetic operations and then it is desirable to consider them explicitly). However, one still has a path to reach to all the operations in JVMML as their particular behaviour in our semantics is available through access to appropriate tables associated with our generalised operations. We hope that this solution is useful in all meta-proofs for JVMML as it allows to build a common framework for many analyses which is important when a verification platform is to be built for real JVMML programs.

Naturally, this paper does not provide the full semantics for the JVMML as it is very complex. In fact, in a few places we make deliberate simplifications of the semantics in order to stay comprehensive in presentation.

⁵ The number is even greater when one considers wide instructions as separate.

2 Semantic domains and notation

We give here a small step semantics for the Java bytecode. The general form of a semantics step is:

$$P \vdash h, ts \rightarrow h', ts' \quad (2.1)$$

where P is a program, h, h' are heaps and ts, ts' are states of the threads. The semantic domains of these values are defined in the following way. First, we provide the description of programs: $\text{Prog} = [\text{Cnames} \rightarrow_{\text{fin}} \text{CDesc}]$. Programs are partial functions with finite domain that associate class descriptions from CDesc with class names from Cnames . The class names are just appropriately defined identifiers, the class descriptions are defined as $\text{CDesc} = [\text{Mnames} \rightarrow_{\text{fin}} \text{MDesc}]$ i.e. partial functions with finite domains that associate method descriptions MDesc with the method names. Again the method names are just appropriate identifiers while the method descriptions are somewhat more complicated and defined as

$$\begin{aligned} \text{MDesc} &= [\text{PC} \rightarrow_{\text{fin}} \text{Instr}] \times \text{ExTable} \\ \text{ExTable} &= [\text{PC} \times \text{Cnames} \rightarrow_{\text{fin}} \text{PC}] \end{aligned}$$

where Instr is the set of JVM instructions and ExTable is an exception table for the method. The intent is that a function in $[\text{PC} \rightarrow_{\text{fin}} \text{Instr}]$ provides a mapping from instruction labels to the instructions under the labels. The ExTable returns the handler address for a given exception origin address and class.

The set of heaps is defined to be the set of

$$\text{Heap} = [\text{Loc} \times \text{ThreadId} \rightarrow_{\text{fin}} (\text{Cnames} \times \text{Monitor} \times [\text{Fnames} \rightarrow_{\text{fin}} \text{Val}])]$$

where Loc is the set of locations (e.g. natural numbers or pointers in the current architecture) with a distinguished location null (the set $\text{Loc} \setminus \{\text{null}\}$ will be denoted by Loc^\bullet), ThreadId is the set of the thread identifiers (e.g. natural numbers), Monitor is the set of monitors which will control the lock counter for the given object, this is defined precisely later. Fnames is the set of field names and Val is the set of expected field values i.e. $\text{Val} = \text{int} \uplus \text{long} \uplus \dots \uplus \text{Loc}$. The ThreadId is an argument of the heap, as each thread has its own view of the heap state. The exact way the different views are synchronised is described by the Java Memory Model [9, Section 17].

The set of *thread states* is defined as the set of all finite sets of thread descriptions, $P_{\text{fin}}(\text{Thread}) \times \text{History}$ combined with a state information History which contains an information needed for the thread scheduler to deterministically select a thread to execute. A thread description is

$$\text{Thread} = \text{ThreadId} \times \text{ThreadStatus} \times \text{EvalState} \times \text{FrameStack}$$

where ThreadStatus represents the current status of the thread i.e. sleeping, blocked, running, terminated etc. At last, the $\text{FrameStack} = \text{MethodFrame}^*$ contains a sequence of the method frames of the form

$$\text{MethodFrame} = \text{Cnames} \times \text{Mnames} \times \text{LVals} \times \text{OpStack} \times \text{PC}$$

where **LVals** is the local variable table defined as $[Vars \rightarrow_{fin} \text{Type} \times \text{Val}]$ with the set of local variable indices $Vars = \mathbb{N}$, **Type** being the type of the value in the given entry and **Val** the value contained in the local variable table; **OpStack** is the operand stack defined as $(\text{StackKind} \times (\text{Val} \uplus \text{PC}))^*$, where **StackKind** represents the type of the value in the current cell of the stack, note that we have to add **PC** type to make sure we can put labels of bytecode instructions used by subroutine commands; the same set **PC** is used as the final compound of **MethodFrame** and the value points to the currently executed bytecode instruction; **EvalState** is a set that represents the information on which exception has been thrown. We may assume that $\text{EvalState} = \text{Loc}$. The special location **null** is used to mark the situation that no exception has been thrown. We also assume that certain exceptions, such as **NullPointerException**, **ClassCastException** etc. are preallocated on the heap. This greatly simplifies the semantics as otherwise a number of semantic rules would be needed to allocate the exception on the stack and call its constructor before actually throwing it. And since the simplification does not concern used defined exceptions we decided not to complicate the semantics.⁶

It is worth mentioning that the semantics we provide here is in the so called *defensive* style i.e. we provide the type identification along with the operand stack and local variables table entries to check if the values stored there have correct type.

We can now define the set of monitors **Monitor** to be the product $\text{ThreadId} \times \mathbb{N}$. A pair from the set represents the identifier of the thread that holds the lock and the number of the times the thread entered the monitor. We assume that the set **ThreadId** contains a distinguished constant **none** which is used to represent the situation when no thread holds the monitor.

A natural operation on the operand stack o is pushing an element e . It is written as $e \cdot o$. The examining the top of the stack is done by pattern matching and $o = e \cdot o'$ means that the stack o contains e at the top followed by the rest in o' .

The data structures which describe the state of the virtual machine are complicated. Therefore we need further notation to retrieve the information from them. First, we have to introduce the scheduler which chooses the particular thread to be executed: $\ast : P_{fin}(\text{Thread}) \times \text{History} \rightarrow \text{Thread}$. We do not provide a particular definition for **History** as this is implementation dependent. We assume only that the scheduler returns any element from its first argument. To make the notation more succinct we write \ast_{ts} to denote $\ast(ts)$. The components of the current thread are denoted as $\ast_{ts} = \langle \text{tid}_{ts}, \text{tstatus}_{ts}, \text{est}_{ts}, \text{tfs}_{ts} \rangle$. As tfs_{ts} is also a composite value, we introduce further notation

$$\text{tfs}_{ts} = \langle \text{cnm}_{ts}, \text{mnm}_{ts}, \text{lv}_{ts}, \text{ostck}_{ts}, \text{pc}_{ts} \rangle \cdot \text{tfs}_{ts}^{\text{tail}} \quad (2.2)$$

where cnm_{ts} is the class name and mnm_{ts} is the method name of the currently executed method, lv_{ts} is the local variables table for the current method, ostck_{ts} is the current operand stack, pc_{ts} is the label of the currently executed instruction. The value $\text{tfs}_{ts}^{\text{tail}}$ denotes the (possibly empty) sequence of remaining method frames on the frame stack.

⁶ In the Bicolano [13] JVM semantics the space on the heap is allocated but the constructor is not called.

2.1 Modification and lookup notation

We frequently modify slightly a given thread state to obtain a new one. The modification is described using the notation $changed_item[replaced_part \leftarrow new_part]$. These can be defined precisely as the construction of a new value where all components but $replaced_part$ are unchanged and the latter is replaced by new_part . For example $tfs[lv \leftarrow lv']$ is a thread state tfs modified so that its local variable table lv_{ts} in the topmost method frame is replaced with a new table lv' .

The lookup of a particular instruction is done using the notation $P@pc.mnm.cnm$ where $P \in \mathbf{Prog}$, $pc \in \mathbf{PC}$, $mnm \in \mathbf{Mnames}$, and $cnm \in \mathbf{Cnames}$. This operation extracts from the program P the class declaration cnm and then it uses the Java method lookup scheme to retrieve the method of the name mnm (we assume the method name is such that it takes into account the signature of the method and therefore uniquely determines the method in the class). Then pc indicates which bytecode instruction from the code of the method should be retrieved.

Similarly, $P@etable.mnm.cnm$ denotes the exception table for the method of the name mnm in the class cnm in P .

For $h \in \mathbf{Heap}$, $s \in \mathbf{Loc}$, and $i \in \mathbf{ThreadId}$ we write $h(s, i)$ to denote the value at the location s visible in the heap h from the thread i . In most cases i is clear from the context so we omit it and write $h(s)$. As $h(s)$ is a compound value, we define

$$\begin{aligned} h(s)@cnm &= \pi_1(h(s)) & h(s)@monitor &= \pi_2(h(s)) & h(s)@obj &= \pi_3(h(s)) \\ h(s)@tid &= \pi_1(\pi_2(h(s))) & h(s)@lcount &= \pi_2(\pi_2(h(s))) \end{aligned}$$

In case $s = \mathbf{null}$ or $s \notin \mathbf{dom}(h)$, the notations above have the value \perp .

2.2 Auxiliary definitions

Throughout the following semantics description we use many minor notations. This section collects the description of their meaning.

The names such as `int` are used here in two meanings, as a name for the set of elements in the Java type of native integers and as a syntactical identifier which is used to refer to the set. The 64-bit values divide into two halves. The notation $\mathbf{long}(m_1, m_2)$ (resp. $\mathbf{double}(m_1, m_2)$) means the 64-bit value of type `long` (resp. `double`) constructed from two 32-bit words m_1 and m_2 . The type of a half with no distinction to which half and for which type (`long` or `double`) for a 64-bit value is denoted as `half`.

The Java Virtual Machine handles the 64-bit types in a special way. Therefore, the Java computational kinds are divided according to [12, Section 3.11.1] in two categories: $\mathbf{Cat1} = \{\mathbf{int}, \mathbf{float}, \mathbf{ref}, \mathbf{returnAddr}\}$ for 32-bit types and $\mathbf{Cat2} = \{\mathbf{long}, \mathbf{double}\}$ for 64-bit types. We will also use the notation $\mathbf{Cat1}^\bullet$ to denote $\mathbf{Cat1} \setminus \{\mathbf{returnAddr}\}$.

As soon as a current thread is chosen we can conclusively determine the currently executed method. This method is denoted $\mathbf{cmthd} \in \mathbf{MDesc}$. We also use a function $\mathbf{next} : \mathbf{MDesc} \times \mathbf{PC} \rightarrow \mathbf{PC}$ to obtain the label of the next instruction in the method using the order of the instruction occurrence there.

2.3 Additional remarks

The semantics we give below is in fact more in the flavour of the interleaving semantics than the actual Java Memory Model one. However, we provide here a way to handle the Java Memory Model as our heap is defined so that it can give a different view of the memory to each thread. Other features of the semantics such as class loading, class initialisation, finalisation, native and synchronized methods etc. are not handled as well. However, slight changes of the definitions above can give the rules below the meaning which can take them into account. Adding reflection would be more problematic as it would require us to change the form of semantic steps.

3 Semantics of instructions

The semantic rules present the evolution of runtime structures caused by the execution of instructions. Most of the rules are directly governed by the current instruction of the current method, but those dealing with exceptions are not.

In the course of the semantic transition the scheduler \ast chooses a particular thread in ts to be executed. The notations we introduced in Section 2.1 all rely on the assumption that a thread is fixed. Therefore, we fix a single choice made by \ast throughout each particular rule. However, the choice may change for different steps of our semantics. We also assume that the state of the heap can change after each rule so that the visibility of its content gets partially synchronised among threads. If we do full synchronisation with every step we obtain the interleaving semantics.

3.1 Instruction load

This instruction generalizes all JVM instructions that read local variables and push the value to the operand stack. Its parameters describe the type and source of the value to be written to the stack, the general form of the instruction is $load(k, n)$ where $k \in \mathbf{Cat1}^\bullet \cup \mathbf{Cat2}$ is a kind, and n is a local variable index.

In the simplest case, when k is a 32-bit kind, $k \in \mathbf{Cat1}^\bullet$, the instruction reads a value from the local variable pointed by the index n and puts the value on the top of the operand stack. It is required that the value is of kind k .

$$\begin{array}{c}
 lv_{ts}(n) = (k, m) \quad ostck' = (k, m) \cdot ostck_{ts} \quad pc' = \text{next}(\text{cmthd}, pc_{ts}) \\
 \hline
 \frac{P@pc_{ts}.mnm_{ts}.cnm_{ts} = load(k, n) \quad k \in \mathbf{Cat1}^\bullet \quad est_{ts} = \text{null}}{P \vdash h, ts \rightarrow h, ts[ostck \leftarrow ostck'][pc \leftarrow pc']} \quad ncat1\text{-load}
 \end{array} \tag{3.1}$$

If k denotes a category-2 kind (**long** or **double**), the value to push on the stack is obtained from the values of two variables, indexed by n and $n + 1$. This is because category-2 values occupy two subsequent cells in the local variables array. We provide an artificial kind **half** for the second variable in such a pair of variables. Following the JVM description [12, Section 3.6.2] we use a single operand stack element for a category-2 value.

$$\begin{array}{c}
 \text{lv}_{ts}(n) = (k, m_1) \quad \text{lv}_{ts}(n+1) = (\text{half}, m_2) \\
 \text{ostck}' = (k, k(m_1, m_2)) \cdot \text{ostck}_{ts} \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \\
 P@_{\text{pc}_{ts}. \text{mnm}_{ts}. \text{cnm}_{ts}} = \text{load}(k, n) \quad k \in \text{Cat2} \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad \text{ncat2-load}
 \end{array} \quad (3.2)$$

3.2 Instruction store

This instruction generalizes all JVM instructions that pop a value from the operand stack and put it in the local variable table. Its arguments are the kind and destination of the popped value, the general form of the instruction is $\text{store}(k, n)$ where $k \in \text{Cat1}^\bullet \cup \text{Cat2}$ is a kind and n is a local variable index.

In case of a category-1 kind, the store instruction pops the topmost value from the operand stack and stores it in a local variable indexed by n .

$$\begin{array}{c}
 \text{lv}' = \text{lv}_{ts}[n \leftarrow (k, m)] \\
 \text{ostck}_{ts} = (k, m) \cdot \text{ostck}' \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \\
 P@_{\text{pc}_{ts}. \text{mnm}_{ts}. \text{cnm}_{ts}} = \text{store}(k, n) \quad k \in \text{Cat1}^\bullet \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'][\text{lv} \leftarrow \text{lv}'] \quad \text{ncat1-store}
 \end{array} \quad (3.3)$$

If $k \in \text{Cat2}$, two subsequent variables, n and $n+1$, are modified. It is required that the first variable is of kind k , and the second one is of kind half .

$$\begin{array}{c}
 \text{lv}' = \text{lv}_{ts}[n \leftarrow (k, m_1)][n+1 \leftarrow (\text{half}, m_2)] \\
 \text{ostck}' = (k, k(m_1, m_2)) \cdot \text{ostck}_{ts} \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \\
 P@_{\text{pc}_{ts}. \text{mnm}_{ts}. \text{cnm}_{ts}} = \text{store}(k, n) \quad k \in \text{Cat2} \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad \text{ncat2-store}
 \end{array} \quad (3.4)$$

3.3 Instruction stackop

Instruction $\text{stackop}(\text{op})$ generalizes all JVM instructions that use only the operand stack. It should be noted, that all such instructions operate on a fixed number of top elements, while the bottom part of the stack is neither read nor modified.

The parameter op denotes the stack operation to perform. The meaning of op is obtained through $\text{kinds}_{\text{stackop}}(\text{op})$, which is a set of triples, each of them consisting of: a list of input kinds l , a function f , and a list of output kinds l' .

The list l defines the requirements of the operation with respect to the operand stack. The number of stack elements must not be less than the length of l , and for all i , the i -th element of the stack must be of kind l_i . This is denoted by $\text{check}(s, l)$.

The function $f : \text{OpStack} \rightarrow \text{OpStack}$ is the actual stack operation. $|l|$ elements are popped from the stack and become the input of f , then the result of f is pushed

on the stack; l' describes guaranteed kinds of the result of f . In a sense $f : l \rightarrow l'$.

$$\begin{array}{c}
 (l, f, l') \in \text{kinds}_{\text{stackop}}(\text{op}) \quad \text{ostck}_{ts} = s \cdot r \\
 \text{check}(s, l) \quad \text{ostck}' = f(s) \cdot r \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \\
 P @ \text{pc}_{ts} . \text{mm}_{ts} . \text{cnm}_{ts} = \text{stackop}(\text{op}) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-stackop}
 \end{array} \quad (3.5)$$

For example, the JVM instruction `iadd` is mapped to $\text{stackop}(\text{iadd})$, and

$$\text{kinds}_{\text{stackop}}(\text{iadd}) = \{([\text{int}, \text{int}], f_{\text{iadd}}, [\text{int}])\}$$

where f_{iadd} performs addition of two 32-bit integers.

Polymorphic instructions, such as `swap` or `dup`, have more than one item in $\text{kinds}_{\text{stackop}}$, for instance $\text{kinds}_{\text{stackop}}(\text{dup2})$ is equal to

$$\{([k_1, k_2], f_{\text{dup2}}, [k_1, k_2, k_1, k_2])\}_{k_1, k_2 \in \text{Cat1}} \cup \{([k], f_{\text{dup}}, [k, k])\}_{k \in \text{Cat2}}$$

3.4 Instruction *cond*

This instruction generalizes all JVM instructions that may affect the program control flow inside the current method, but do not modify the method frame stack, that is all unconditional and conditional jumps including `tableswitch`, `lookupswitch`, `jsr` and `ret`. The instruction reads and modifies the operand stack and the program counter (PC). The general form of the instruction is $\text{cond}(\text{op}, d)$ where op identifies the actual operation on runtime structures and $d \in D_{\text{cond}}$, $D_{\text{cond}} = [\mathbb{N} \rightarrow_{\text{fin}} \text{PC}]$ represents the static arguments of the instruction, which consist of an indexed table of addresses. The form and role of $\text{kinds}_{\text{cond}}(\text{op})$ is analogous to the role of $\text{kinds}_{\text{stackop}}$. The difference here is the type of $f : D_{\text{cond}} \times \text{OpStack} \times \text{PC} \rightarrow \text{OpStack} \times \text{PC}$.

Arguments of f are the table of offsets, the relevant part of the operand stack, and the next PC. The function f returns the new value of the relevant part of the operand stack and the new value of PC. Only one JVM jump instruction, `jsr`, does put some value onto the operand stack: the current PC; `ret` is the only instruction that pops the new value of PC from the operand stack.

$$\begin{array}{c}
 (l, f, l') = \text{kinds}_{\text{cond}}(\text{op}) \quad \text{ostck}_{ts} = s \cdot r \\
 \text{check}(s, l) \quad (s', \text{pc}') = f(d, s, \text{next}(\text{cmthd}, \text{pc}_{ts})) \quad \text{ostck}' = s' \cdot r \\
 P @ \text{pc}_{ts} . \text{mm}_{ts} . \text{cnm}_{ts} = \text{cond}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-cond}
 \end{array} \quad (3.6)$$

For example, the JVM instruction `ifeq(o)`, performing a jump if the value on the top of the stack is the integer 0, is mapped to $\text{cond}(\text{ifeq}, [0 \mapsto \text{pc} + o])$, and $\text{kinds}_{\text{cond}}(\text{ifeq}) = ([\text{int}], f_{\text{ifeq}}, [])$ with $f_{\text{ifeq}}(g, s, \text{pc})$ returning $([], g(0))$ if $s = [(\text{int}, 0)]$ and $([], \text{pc})$ otherwise. For `lookupswitch`, g is a function that maps key values to the corresponding addresses.

3.5 Instruction *iinc*

The opcode *iinc* is the only JVM instruction that uses solely the local variables array. The corresponding instruction in our formalisation is *iinc*(*n*, *c*), where *n* is a local variable index and *c* is an integer value.

If the local variable *n* is of kind *int*, its value is increased by *c*, according to the Java *int* arithmetic.

$$\begin{array}{c}
 \text{lv}_{ts}(n) = (\text{int}, m) \quad \text{lv}' = \text{lv}_{ts}[n \leftarrow (\text{int}, m +_{\text{int}} c)] \\
 \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \quad P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{iinc}(n, c) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{lv} \leftarrow \text{lv}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-iinc}
 \end{array} \tag{3.7}$$

3.6 Instruction *get*

This instruction reads the heap and modifies the operand stack. The general form of the instruction is *get*(*op*, *d*), where *op* is the operator and *d* contains an optional static argument—a qualified field name.

As for the previous rules, *kinds_{get}*(*op*, *d*) provides expected kinds of arguments on the stack, list of kinds of values to be put on the stack, and the function *f* of type $D_{\text{get}} \times \text{OpStack} \times \text{Heap} \rightarrow \text{OpStack} \uplus \text{Loc}^\bullet$. The function *f* attempts to read the indicated object field or array cell from the heap. If it exists, *f* returns the modified part of the stack, which is the value from the heap.

$$\begin{array}{c}
 (l, f, l') = \text{kinds}_{\text{get}}(\text{op}, d) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \\
 s' = f(d, s, h) \quad s' \in \text{OpStack} \quad \text{ostck}' = s' \cdot r \\
 \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \quad P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{get}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-get}
 \end{array} \tag{3.8}$$

If it is impossible to obtain the requested value and an exception must be thrown (e.g. *NullPointerException*), *f* returns the location *e* of the exception in the heap and the resulting evaluation state is the exceptional state.

$$\begin{array}{c}
 (l, f, l') = \text{kinds}_{\text{get}}(\text{op}, d) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \\
 e = f(d, s, h) \quad e \in \text{Loc}^\bullet \quad P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{get}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e] \quad \text{exn-get}
 \end{array} \tag{3.9}$$

3.7 Instruction *put*

This instruction reads and modifies the operand stack and the heap without creating new locations. The general form of the instruction is *put*(*op*, *d*), where *op* is the operator and *d* contains an optional static argument—a qualified field name.

The role of $\text{kinds}_{\text{put}}(\text{op}, d)$ is similar to previous kinds with the function f of type $D_{\text{put}} \times \text{OpStack} \times \text{Heap} \rightarrow \text{Heap} \uplus \text{Loc}^\bullet$. The function f attempts to modify the indicated field or array cell in the heap. If the indicated item exists and may be changed, f returns the modified heap.

Note that the value written by put does not have to be accessible by other threads immediately. In fact, any part of heap may be synchronized with the thread cache at any point of program execution, with Java Memory Model constraints preserved. In particular, the two halves of a category-2 value may be synchronized independently.

$$\begin{array}{c}
 (l, f, l') \in \text{kinds}(\text{op}, d) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \quad \text{ostck}' = r \\
 h' = f(d, s, h) \quad h' \in \text{Heap} \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{put}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h', ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-put}
 \end{array} \quad (3.10)$$

If the requested object does not exist, an exception is thrown.

$$\begin{array}{c}
 (l, f, l') \in \text{kinds}(\text{op}, d) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \quad \text{ostck}' = r \\
 e = f(d, s, h) \quad e \in \text{Loc}^\bullet \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{put}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e] \quad \text{exn-put}
 \end{array} \quad (3.11)$$

3.8 Instruction new

This instruction modifies the operand stack and the heap by creating a new location. The general form of the instruction is $\text{new}(\text{op}, d)$, where op is the operator and d is a list of its arguments (integers and class names).

The precise meaning of the instruction is given by the function f , obtained from $\text{kinds}_{\text{new}}(\text{op}, d)$, together with expected kinds of arguments on the stack and the expected kinds of values to be stored on the operand stack, which is actually always one value of kind **ref**. The function f itself manipulates the heap, allocating the requested structure and returning the location of the allocated structure and the new heap in case of success, and the exception otherwise.

Note that this instruction and its rules are very similar to put . We preferred to keep the two separated as new adds new locations to the heap while put only modifies existing ones.

$$\begin{array}{c}
 (l, f, l') = \text{kinds}_{\text{new}}(\text{op}, d) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \quad \text{ostck}' = s' \cdot r \\
 (s', h') = f(d, s, h) \quad s' \in \text{OpStack} \quad h' \in \text{Heap} \\
 \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \quad P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{new}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h', ts[\text{ostck} \leftarrow \text{ostck}'][\text{pc} \leftarrow \text{pc}'] \quad n\text{-new}
 \end{array} \quad (3.12)$$

$$\begin{array}{c}
 (l, f, l') = \text{kinds}_{new}(\text{op}, d) \\
 \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \quad e = f(d, s, h) \quad e \in \text{Loc}^\bullet \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{new}(\text{op}, d) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e] \quad \text{exn-new}
 \end{array} \quad (3.13)$$

3.9 Instruction monitor

This instruction can modify the state of threads by trying to acquire or release a monitor. The operation itself is done by modifying an object on the heap. The *monitor* instruction expects one location on the operand stack: the object with which the monitor in question is associated. The general form of the instruction is *monitor*(*op*), where *op* is either *enter* or *exit*.

Both variants of the instruction are handled by the same two rules — one for correct operation, one for raising an exception. The rules are governed by a partial function $f : \text{ThreadId} \times \text{Loc} \times \text{ThreadId} \times \mathbb{N} \rightarrow \text{ThreadId} \times \mathbb{N} \cup \text{Loc}$ obtained from $\text{kinds}_{monitor}(\text{op})$. If *op* = *enter*, $f(\text{tid}', s, \text{tid}, c)$ is defined only if $s = \text{null}$ or $\text{tid} = \text{none}$ or $\text{tid} = \text{tid}'$. In the first case f returns a `NullPointerException`, in the second $(\text{tid}', 1)$, and in the third $(\text{tid}', c + 1)$. Since f is not defined when $s \neq \text{null}$ and $\text{tid}' \neq \text{tid} \neq \text{none}$, i.e. the monitor is owned by a different thread, the rule cannot be fired until the monitor is released.

If *op* = *exit*, f returns the exception `IllegalMonitorStateException` if $\text{tid} \neq \text{tid}'$ and otherwise either `NullPointerException` or $(\text{none}, 0)$ or $(\text{tid}, c - 1)$ depending on the values of s and c .

For the lack of space we did not formalize other synchronization operations related to synchronized methods. Note however, that it is quite easy to syntactically transform a synchronized method into one having *monitor*(*enter*) at the beginning and *monitor*(*exit*) at every exit point.

$$\begin{array}{c}
 f = \text{kinds}_{monitor}(\text{op}) \quad \text{ostck}_{ts} = s \cdot r \quad s \in \text{Loc} \\
 (\text{tid}', \text{lcount}') = f(\text{tid}_{ts}, s, h(s)@\text{tid}, h(s)@\text{lcount}) \\
 \text{tid}' \in \text{ThreadId} \quad \text{lcount}' \in \mathbb{N} \quad \text{pc}' = \text{next}(\text{cmthd}, \text{pc}_{ts}) \quad \text{ostck}' = r \\
 h' = h[s \leftarrow h(s)[\text{tid} \leftarrow \text{tid}'][\text{lcount} \leftarrow \text{lcount}']] \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{monitor}(\text{op}) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h', ts[\text{pc} \leftarrow \text{pc}'][\text{ostck} \leftarrow \text{ostck}'] \quad \text{n-monitor}
 \end{array} \quad (3.14)$$

$$\begin{array}{c}
 f = \text{kinds}_{\text{monitor}}(\text{op}) \quad \text{ostck}_{ts} = s \cdot r \quad s \in \text{Loc} \\
 e = f(\text{tid}_{ts}, s, h(s)@\text{tid}, h(s)@\text{lcount}) \quad e \in \text{Loc} \\
 P@\text{pc}_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{monitor}(\text{op}) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e] \quad \text{exn-monitor}
 \end{array} \quad (3.15)$$

3.10 Instruction invoke

This instruction modifies the operand stack, the method frame stack and reads the heap. The general format of the instruction is *invoke(mode, cnm, mnm)*, where *mode* is one of **interface**, **special**, **static** or **virtual**, and *cnm* and *mnm* are class and method name of the method that is supposed to be called.

The principal action of this instruction is to find the method code, prepare the new method frame and pass the execution to the new method instance. To do that the types *l* of expected values on the stack together with the expected types return by the method *l'* are read from *kinds_{invoke}(mode, cnm, mnm)*, which in turn reads them from the method signature. The list *l'* is of length at most 1. Next, the *dispatch* function is executed which checks that the method's flags are not contradictory to the invoke *mode*, that the access rights are preserved (for **private** and **protected** methods) and selects the type of dispatch by returning either the class *cnm* for static dispatch or the class of the first location of *s* in *h* for dynamic dispatch. The *dispatch* function can also return an exception.

The rest of the *n-invoke* rule is devoted to the preparation of the new method frame: the function *initlv* places the arguments from the stack in the local variable table of the new frame after splitting values of type **long** and **double** and performing necessary floating-point value set conversions [12, Section 3.8.3]. Finally, the new method frame is put on the method frame stack with the empty initial operand stack and *pc* = 0.

Synchronized methods are not handled here, but please see the remark at the end of Section 3.9.

$$\begin{array}{c}
 (l, l') = \text{kinds}_{\text{invoke}}(\text{mode}, \text{cnm}, \text{mnm}) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \\
 \text{cnm}' = \text{dispatch}(\text{mode}, \text{cnm}, \text{mnm}, s, h) \quad \text{tfs}' = \text{tfs}_{ts}[\text{ostck} \leftarrow r] \\
 \text{lv}' = \text{initlv}(\text{lvlength}(P@\text{mnm}.\text{cnm}), s) \\
 \text{tfs}'' = \langle \text{cnm}', \text{mnm}, \text{lv}', [], 0 \rangle \cdot \text{tfs}' \\
 P@\text{pc}_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{invoke}(\text{mode}, \text{cnm}, \text{mnm}) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow \text{tfs}''] \quad \text{n-invoke}
 \end{array} \quad (3.16)$$

$$\begin{array}{c}
 (l, l') = \text{kindsof}_{\text{invoke}}(\text{mode}, \text{cnm}, \text{mnm}) \quad \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \\
 e = \text{dispatch}(\text{mode}, \text{cnm}, \text{mnm}, s, h) \quad e \in \text{Loc} \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{invoke}(\text{mode}, \text{cnm}, \text{mnm}) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e] \quad \text{exn-invoke}
 \end{array} \quad (3.17)$$

3.11 Instruction return

This instruction returns from the current method. It reads the operand stack and modifies the method frame stack by removing the current frame and updating the previous frame: moving the pc to the next instructions (usually over an *invoke* instruction) and updating the operand stack by pushing the return value, after the floating-point value set conversion [12, Section 3.8.3]. The general form of the instruction is *return*(*l*) where *l* is a list of kinds of length at most 1.

Even though [12] does not specify this explicitly, we decided to add the rule *n-term-return*, to deal with the termination of the method corresponding to the last frame on the frame stack.

These rules do not handle releasing of monitor when exiting a synchronized method. This can be simulated, however, by putting a *monitor(exit)* instruction before every *return* statement. Please see also the discussion in Section 3.9.

$$\begin{array}{c}
 \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \\
 \text{tfs}_{ts} = f_1 \cdot \langle \text{cnm}', \text{mnm}', \text{lv}', \text{ostck}', \text{pc}' \rangle \cdot \text{tfs}^{\text{tail}} \quad f_1 \in \text{MethodFrame} \\
 \text{tfs}' = \langle \text{cnm}', \text{mnm}', \text{lv}', \text{vsc}(s) \cdot \text{ostck}', \text{next}(P@mnm'.\text{cnm}', \text{pc}') \rangle \cdot \text{tfs}^{\text{tail}} \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{return}(l) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow \text{tfs}'] \quad \text{n-return}
 \end{array} \quad (3.18)$$

$$\begin{array}{c}
 \text{ostck}_{ts} = s \cdot r \quad \text{check}(s, l) \quad \text{tfs}_{ts} = [f] \quad f \in \text{MethodFrame} \\
 \text{tfs}' = [] \quad \text{tstatus}' = \text{TERMINATED} \\
 P@pc_{ts}.\text{mnm}_{ts}.\text{cnm}_{ts} = \text{return}(l) \quad \text{est}_{ts} = \text{null} \\
 \hline
 P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow \text{tfs}'][\text{tstatus} \leftarrow \text{tstatus}'] \quad \text{n-term-return}
 \end{array} \quad (3.19)$$

3.12 Instruction throw

This instruction takes no parameters, it reads and removes the location of the exception from the stack and changes the evaluation state of the current thread (the rule *ex-throw*). The way the exceptions are handled in our semantics is the

following. The evaluation state (*est*) component of each thread says if the execution is in the normal state, when *est* = **null**, or in exception handling state otherwise.

Note that the switch to the latter state can be done not only by executing the *throw* instruction but also by throwing an exception (e.g. `NullPointerException`) by other semantic rules. If *est* = *e* is a location of a valid exception, the remaining rules *ex-in-handle*, *ex-out-handle* or *ex-term-handle* can be fired, depending on the fact whether the exception is handled inside the current method or provokes its abrupt termination. In the latter case, the *ex-term-handle* rule handles the special case where the current method is the last on the method frame stack. This rule does not have a direct correspondence in [12], just like the rule *n-term-return*.

The feature which is not handled is the release of monitor when a synchronized method is abruptly terminated by an exception. Note however that this can be simulated by adding a catch-all exception handler which would execute the instruction *monitor(exit)* and then rethrow the exception. See also the discussion at the end of Section 3.9.

$$\begin{array}{c} \text{ostck}_{ts} = e \cdot r \quad e \in \text{Loc}^\bullet \\ \hline \frac{P@pc_{ts}.mnm_{ts}.cnm_{ts} = \text{throw} \quad \text{est}_{ts} = \text{null}}{P \vdash h, ts \rightarrow h, ts[\text{est} \leftarrow e]} \quad \text{ex-throw} \end{array} \quad (3.20)$$

$$\begin{array}{c} \text{ostck}' = [e] \quad (pc_{ts}, h(e)@cnm) \in \text{dom}(P@etable.mnm_{ts}.cnm_{ts}) \\ pc' = P@etable.mnm_{ts}.cnm_{ts}(pc_{ts}, h(e)@cnm) \quad \text{est}_{ts} = e \in \text{Loc}^\bullet \\ \hline \frac{P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'] [pc \leftarrow pc'] [\text{est} \leftarrow \text{null}]}{P \vdash h, ts \rightarrow h, ts[\text{ostck} \leftarrow \text{ostck}'] [pc \leftarrow pc'] [\text{est} \leftarrow \text{null}]} \quad \text{ex-in-handle} \end{array} \quad (3.21)$$

$$\begin{array}{c} (pc_{ts}, h(e)@cnm) \notin \text{dom}(P@etable.mnm_{ts}.cnm_{ts}) \\ \text{tfs}_{ts} = f_1 \cdot f_2 \cdot \text{tfs}_{ts}^{\text{tail}} \quad f_1, f_2 \in \text{MethodFrame} \quad \text{est}_{ts} = e \in \text{Loc}^\bullet \\ \hline \frac{P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow f_2 \cdot \text{tfs}_{ts}^{\text{tail}}]}{P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow f_2 \cdot \text{tfs}_{ts}^{\text{tail}}]} \quad \text{ex-out-handle} \end{array} \quad (3.22)$$

$$\begin{array}{c} (pc_{ts}, h(e)@cnm) \notin \text{dom}(P@etable.mnm_{ts}.cnm_{ts}) \\ \text{tfs}_{ts} = [f] \quad f \in \text{MethodFrame} \quad \text{est}_{ts} = e \in \text{Loc}^\bullet \\ \hline \frac{P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow []] [\text{tstatus} \leftarrow \text{TERMINATED}]}{P \vdash h, ts \rightarrow h, ts[\text{tfs} \leftarrow []] [\text{tstatus} \leftarrow \text{TERMINATED}]} \quad \text{ex-term-handle} \end{array} \quad (3.23)$$

3.13 Instructions without semantics

The functionality of a few instructions cannot be expressed by semantical transformation of the runtime structures as their meaning is not described in JVM specification [12]. These are `breakpoint`, `impdep1`, `impdep2`, and the instruction with the opcode 186.⁷ Therefore, they are omitted from the paper. The opcode `wide` is taken into account along with the non-wide operations.

⁷ JVM semantics says: ‘For historical reasons, opcode value 186 is not used.’

4 Conclusions

We have presented a concise formalisation of JVMML which turns out to be factorisable into 12 instruction mnemonics. This was possible because we separated generic operation of many instructions and tabularised particular behaviours of individual opcodes. In this way we rigorously reduced the overall complexity of the whole language without significantly sacrificing its features.

References

- [1] Atkey, R., *CoqJVM: An executable specification of the Java Virtual Machine using dependent types*, in: M. Miculan, I. Scagnetto and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, Lecture Notes in Computer Science **4941** (2008), pp. 18–32.
- [2] Chrząszcz, J., *Modules in Coq are and will be correct*, in: S. Berardi, M. Coppo and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, Lecture Notes in Computer Science **3085**, 2004, pp. 130–146.
- [3] Chrząszcz, J., P. Czarnik and A. Schubert, *A dozen instructions make Java bytecode*, available for download at <http://www.mimuw.edu.pl/~chrzaszcz/papers/>.
- [4] Consortium, M., *Deliverable 3.1: Bytecode specification language and program logic* (2006), available online from <http://mobius.inria.fr>.
- [5] Dean, D., E. Felten and D. Wallach, *Java security: From HotJava to Netscape and beyond*, Security and Privacy, IEEE Symposium on (1996), pp. 190–200.
- [6] Demange, D., T. Jensen and D. Pichardie, *A provably correct stackless intermediate representation for Java bytecode*, Technical Report Research Report 7021, INRIA (2009).
- [7] Freund, S. N., “Type systems for object-oriented intermediate languages,” Ph.D. thesis, Stanford University (2000).
- [8] Freund, S. N. and J. C. Mitchell, *The type system for object initialization in the Java bytecode language*, ACM Transaction on Programming Languages and Systems **21** (1999), pp. 1196–1250.
- [9] Gosling, J., B. Joy, G. Steele and G. Bracha, “The Java Language Specification, third edition,” The Java Series, Addison Wesley, 2005.
- [10] Kahrs, S., D. Sannella and A. Tarlecki, *The definition of Extended ML: A gentle introduction*, Theoretical Computer Science **173** (1997), pp. 445–484.
- [11] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine, and compiler*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 619–695.
- [12] Lindholm, T. and F. Yellin, “The Java (TM) Virtual Machine Specification (Second Edition),” Prentice Hall, 1999.
- [13] Pichardie, D., *Bicolano – Byte Code Language in Coq* (2006), <http://mobius.inria.fr/bicolano>. Summary appears in [4].
- [14] Pusch, C., *Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL*, in: R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS ’99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, Lecture Notes in Computer Science **1579** (1999), pp. 89–103.
- [15] Qian, Z., *A formal specification of Java Virtual Machine instructions for objects, methods and subroutines*, in: *Formal Syntax and Semantics of Java* (1999), pp. 271–312.
- [16] Stärk, R. F., J. Schmid and E. Börger, “Java and the Java Virtual Machine: Definition, Verification, Validation,” Springer, 2001.
- [17] Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, *Soot - a Java bytecode optimization framework*, in: *CASCON ’99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), p. 13.

Reconstruction of Type Information from Java Bytecode for Component Compatibility ¹

Jaroslav Bauml, Premek Brada

{jbauml|brada}@kiv.zcu.cz
Department of Computer Science and Engineering
University of West Bohemia
Univerzitni 8, 30614, Pilsen, Czech Republic

Abstract

The Java type system is strictly checked by both the compiler and the runtime bytecode interpreter of the JVM. These mechanisms together guarantee appropriate usage of class instances. Using modern component systems can however circumvent these static checks, because incompatible versions of classes can be bound together during component installation or update. Such problematic bindings result in `ClassCastException` or `NoSuchMethodException` runtime errors. In this paper we describe a representation of Java language types suitable for checking component compatibility. The presented approach applies various bytecode handling techniques to reconstruct a representation of the Java types contained in a component implementation, using different sources of class data. The representation is then used during build- and run-time type system verifications with the aim to prevent these kinds of errors. We have successfully applied this approach to prevent OSGi component incompatibilities.

Keywords: type reconstruction, reflection, bytecode analysis, subtyping, component compatibility

1 Introduction

Statically typed languages have clear advantages for which they are used in the majority of software systems. As Erik Allen describes clearly in [4], static type checking improves robustness through early error detection, increases performance by making the required checks at the best time and supplements the weaknesses of unit testing. Early checks of type coherence done by compiler ensure type safety of the program code and guarantee that types used at runtime are compatible.

This clear situation is however complicated by component systems. One of the most important contributions of Component-Based Software Engineering (CBSE) [20,5] is the decomposition of applications into smaller parts – components. An application is not built and deployed as one monolithic block but composed from components which encapsulate parts of its functionality, possibly developed by independent vendors. Each component has its own interface which is split into two sides

¹ This work was supported by the Grant Agency of the Czech Republic under grant 201/08/0266.

– the sets of provided and required features. Through these features, components are wired together according to their declared dependencies. Today more and more Java based systems move to this kind of modularized or component-based architecture, supported by systems like OSGi, Netbeans plugins or Android application architecture.

At component deployment time, a problem stemming from type mismatches can occur in case the structure of a type exported by a providing component changes during its evolution. The client components will still be wired to such provider (since the type names in the provided-required feature pairs match) but the provided language type can now be incompatible with the notion of this (referenced) type on the client side.

As we show in [8] and [6], this scenario is realistic in case of independent component evolution. We have therefore proposed a method for deciding on component compatibility based on the subtyping comparison of the real structure of the referenced type and the described client’s notion of that type. Such run-time compatibility checks depend on the complete reconstruction of type description from component binary implementation – during and after deployment, its source code is rarely accessible.

In this paper, we discuss in detail the alternative ways that exist for the reconstruction of Java types by bytecode analysis and run-time introspection. The following Section 2 focuses deeply on the motivating problem with real life examples. The features which are utilized in component dependencies actually depend on the component model used. Since we work mainly with the OSGi component model, we will provide short description of OSGi in subsection 2.3.

The proposed method of Java type reconstruction is described in the next two sections. Section 3 describes a Java type system representation employed by our method. The generality of its design allows the representation to be used in other projects to ensure Java language type compatibility. Section 4 describes the approach to deciding on component compatibility which uses algorithms working on the type system representation. The merits of these methods are discussed in the end of the work.

2 Compatibility in Component Software

In the industry and research worlds there are various component models which differ from each other by complexity, level of abstractness, technical maturity and the purpose of use. But in each of these different component models one implicit requirement is shared – it is component compatibility.

2.1 Component Dependencies

Component compatibility is a crucial requirement because of component life cycle. As shown in Figure 1a, when working with standard monolithic software the dependencies between its parts (let say classes) are created at build time when they are also checked by the compiler to be type compatible. In case of problems found by the compiler, the resulting code is not created; when language types (classes and

interfaces) are compatible, the application can be built and deployed to a production site. When a new version of software is developed, the whole monolithic code package is again moved to the production site for application upgrade.

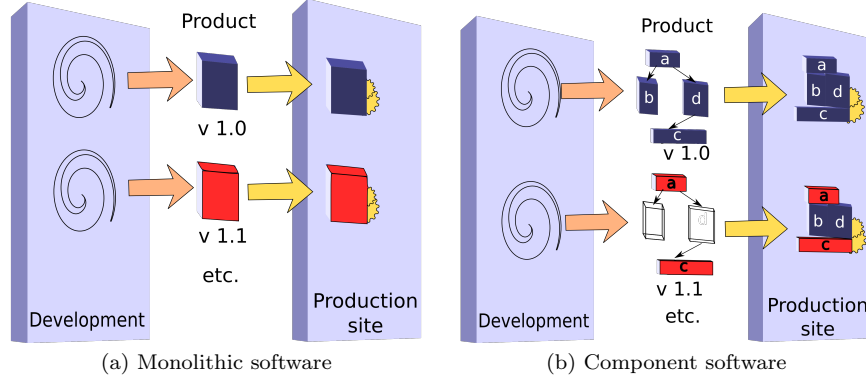


Fig. 1. Software process

When using component software development techniques the situation is similar in general but differs in important details. Each component has its own interface which is composed of two sides – the provided and required one. The provided side consists of features (services, packages, events, ...) which are published by the component so as to be available to component surroundings. Conversely, the required side declares features which the surrounding must supply to the component for its proper function. Through these features, components are wired together, forming dependencies in a way described by the particular component model.

2.2 Type Compatibility in Component Applications

As can be seen in Figure 1b the dependencies between components are also at first checked by compiler at component build time. Unlike the monolithic application scenario however, each component (including those developed by third-party vendors) can then be deployed to the production site separately – without its depended-on suppliers, or to be precise, only with a declaration of these dependencies.

During subsequent update of the component-based application, the wirings among components are re-established at the production site with the component in the new version. When the new version is incompatible, the application will fail with some kind of runtime exception. The probability of the failure is equal to the probability of invoking the type (e.g. class) which exhibits the incompatibility.

In many currently used component models, there is no mechanism to describe the type system of component interface. The types from the provided part of a supplier component are used in a client's source code, creating an implicit "notion" of these types in client's implementation. This binding of the client's code to the supplier's types is logical but creates an invisible static dependency in the client implementation – its notion of the referenced types is based on the structure of the particular supplier's types used during compilation. At compile time, the actual types exported by the supplier and this notion are checked for coherence. However, an analogous mechanism is missing during the component (re)wiring operation in the deployment phase.

This general problem is shared by many Java based component systems. From now on we will present it on a case study of the OSGi platform which is very simple and lightweight and its popularity is growing. For insight to the problems handled further, an elementary knowledge of OSGi is required; if you are familiar with the framework you can skip the next subsection.

2.3 A Brief Overview of OSGi

The Open Services Gateway Initiative (OSGi) platform [18] is an open Java-based framework for service deployment and management. Its uses range from embedded applications to large-scale desktop and enterprise systems. The core of OSGi is the *framework* which creates a runtime environment for managing the deployment and lifecycle of components called *bundles*. A set of standardized basic services, implemented by system bundles, is provided as part of the framework distribution.

A bundle can export (provide) or import (require) Java packages and services, declare native libraries used, and specify dependencies on the execution platform and concrete bundles. The standard Java manifest file holds the specification meta-data. *Packages* are used to access shared types and bundle implementation, and form static bindings between bundles. *Services* are represented by Java interfaces and allow dynamic registration, lookup and (un)binding of functionality using a centralised framework registry.

The onus of service binding is by default on the bundle implementation which brings flexibility in handling runtime changes. If a standardized declarative services module is used, service declaration and binding can be delegated to the framework. On the other hand, dependency resolution for packages is always handled by the framework core, requiring no work on the programmer's side.

2.4 Real Word Problem Example

In this section we describe a concrete example of the problem with component compatibility, showing how the user can be affected by this issue. It is one instance of a set of runtime failures which take hours to track down. Methods which prevent such runtime exceptions can therefore save valuable amounts of development time.



Fig. 2. Real world compatibility problem

To develop a frontend for a research project we decided to extend the Apache Felix Webconsole [1] bundle. This Webconsole is an extendable web-

page for managing a running OSGi framework. It embeds a servlet container which can be extended by registering a service implementing an interface `org.apache.felix.webconsole.AbstractWebConsolePlugin`.

Our plugin bundle, called `subst-verifier`, is very simple. It can verify if a new version of a bundle is compatible with an old version. It has only one HTML form with file input. When the file is uploaded to the plugin, the verification is performed.

When we installed `subst-verifier` and tried to upload a file we got the error message shown in Figure 2. After several hours of problem searching we found the following issue to be its cause: We have used the `org.apache.felix.webconsole` bundle in version 1.2.10 which expects library `commons-fileupload` in version 1.1. This dependency is not handled by metadata description of the component and therefore it was not easy to observe it. Our `subst-verifier` was however compiled to `commons-fileupload` in version 1.2. Because these two versions of a widely used library are not compatible and no compatibility checks are made in OSGi component model, we got a serious runtime crash of our application.

3 Component Type-Level Representation

In order to compare two components and determine the level of their compatibility at runtime (when source files are not accessible), we need a suitable model to represent both components. The representation we describe in this section was designed to capture all syntactic changes of types on the public API of a component. It consists of two layers (see Figure 3). For the layer of the whole component we use a simple metamodel of OSGi called *BundleTypes*. It represents the exported and imported features of a bundle. The second layer describes only the Java types declared and used by the component – it is therefore called *JavaTypes*. Since all OSGi bundle features are implemented as or consist of Java classes, the leaf nodes of the BundleType layer use the Java type representation described by *JavaTypes*.

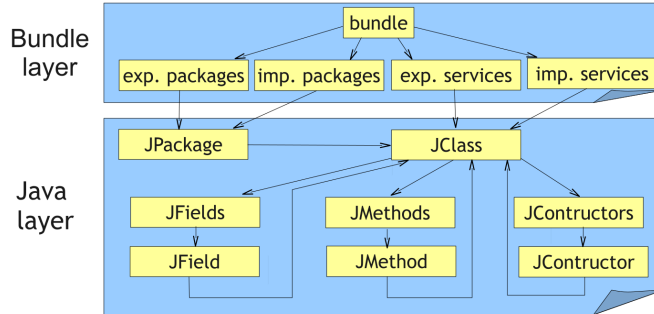


Fig. 3. Component Representation

The *JavaTypes* layer can be used independently of the Bundle representation. We have designed this layer according to the Java Language Specification, Third Edition [14]. The *JavaTypes* layer is very similar to the Java reflection API [13] but is more general because the contents can be obtained from other sources than just reflection. For reasons described below, the unique feature of *JavaTypes* is the ability to create the type representation also from component bytecode or the possibility to create representation of nonexistent classes by manipulation.

There are some quite subtle differences between the Java language type system and the type system which JVM uses when interpreting bytecode. Since our method is focused on reconstructing Java representation and reasoning over Java programming elements, for the rest of the paper we will use the Java type system.

The base interfaces of *JavaTypes* are shown in Figure 4. *JType* is the parent interface of specific types in Java. These specific types are: *JClass*, *JTypeVariable*, *JParameterizedType*, *JWildcardType* and *JGenericArrayType*. *JClass* represents basic types of Java language – a class or an interface. Other children of *JType* represent generic types. In the rest of the paper we will call types represented by *JClass* as basic types and other types as generic types.

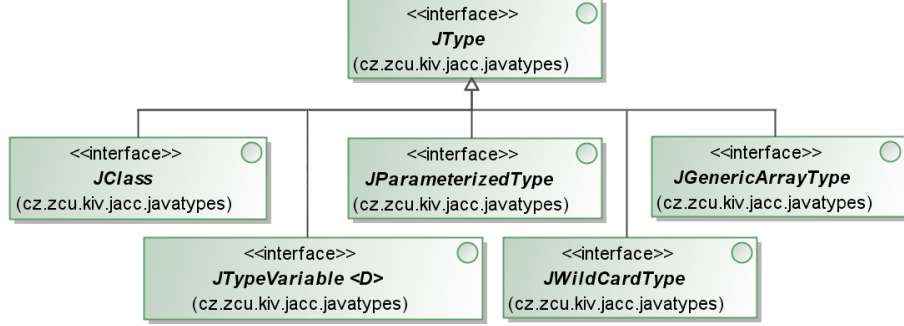


Fig. 4. Base interfaces of JavaTypes representation

To cover generic types, *JTypeVariable* represents a type variable in a class or interface definition, *JParameterizedType* covers the case of an instance with type variable in its definition, and *JWildcardType* represents an instance of type with a wildcard – eg. `List<?>`. Lastly, *JGenericArrayType* represents an array of *JTypeVariable*, *JParameterizedType* or *JWildcardType*. (The situation concerning generics is actually more complicated because there are cases when their representation is not available, e.g. through reconstruction from bytecode which does not contain generics annotations.)

The representation of language features available in Java is summarised in Figure 5. *JClasses* aggregate *JMembers* (*JFields*, *JMethods* or *JConstructors*). For all these elements an *JModifier* can be obtained, expressing their access modifier – *public*, *private*, *static*, etc. All elements which can have an annotation attached have to implement the *JAnnotable* interface which can return an *JAnnotation* object.

3.1 Type Representation Sources

Creating the representation of types referenced by a component’s interface is designed in a way similar to the Java class loading process. The types can be obtained from several sources. Each source has its *JTypeLoader* object that is responsible for reading the types located in the given source. The currently implemented sources for retrieving *JavaTypes* representation are:

- A loaded program – through Java reflection API;

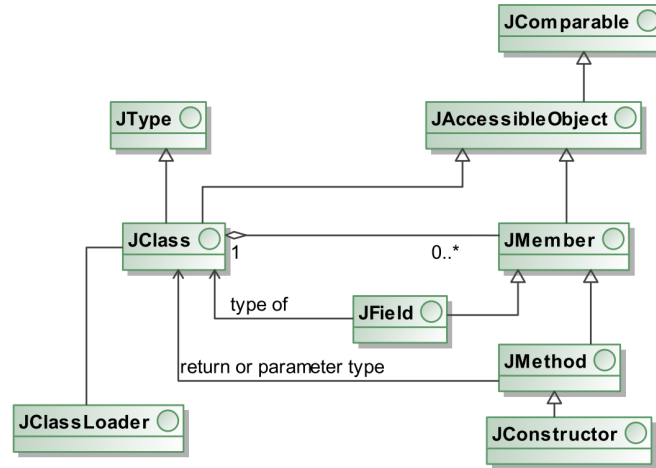


Fig. 5. Diagram of core JavaTypes representation layer

- Compiled but not loaded files – using bytecode inspection;
- Programmatically – custom creation, for cases like testing or stub creation.

Furthermore, each *JType* loader has its parent *JTypeLoader*. The tree organization of loaders and the fact that each loader can create the representation from a different source brings variability into the process of obtaining the component's type representation. For example, when a referenced class is not found inside the component's bytecode we can try to find it through the parent *JType* loader. This can be a reflection loader which creates the class representation in cooperation with a classloader pointed to a classpath (where the class is available). These sources can be arbitrarily combined together.

Retrieving representation using *Java reflection* is quite straightforward because of the intentional similarity between JavaTypes and the reflection API. In fact, JavaTypes implementation for reflection is a *Decorator* design pattern implemented over the reflection API.

The two other options of creating *JavaTypes* – bytecode inspection and custom creation – are more interesting.

When creating the *representation from bytecode*, we use the ASM and BCEL bytecode analysis frameworks. ASM [2] provides a Visitor pattern approach for accessing all parts of class data. We have implemented visitors for the particular JavaTypes classes. BCEL [3] is used for historical reasons (introduced earlier than ASM to the project).

The following example illustrates the creation of a class representation from bytecode. Let us have an *Example* class with one method:

```
public class Example {
    public void callMe(int i, String s) { ... }
}
```

The bytecode method descriptor for the method is:

```
(ILjava/lang/String;)V
```


This bytecode data are read by ASM into our *JMethodVisitor* implementation in the *JTypeLoader*. When called, the visitor creates a *JMethod* instance which the type loader adds to a *JClass* object, created earlier in a similar way.

Custom creation of JavaTypes can be useful in three cases. The simplest one is testing purposes, when we need to create artificial types to perform tests on the representation. Our second use case is programmatic creation of nonexistent types – this will be described in detail below. Lastly we can use the custom creation principle for *stubbing* purposes. Stubbing is a technique for obtaining the replacement of a class we cannot or do not want to load. Such a class is replaced by a dummy one called *stub*.

The creation of stub *JClass* objects in the custom JType loader is parametrized by a classname mask which defines the set of stubbed classes (e.g. `java.lang.*` for core Java classes). In case of loading via reflection, stubs are created for all classes available on system classpath – the assumption is that those classes are shared by all components in the system and therefore do not influence component substitutability. Stubs are also created in all cases when a class is not available inside the component and we can safely assume that its source is not changed by component update (i.e. that the old and new version of the component will reference the same class code) – this is the case of library classes or imported packages.

3.2 Obtaining Complete Bundle Type Representation

To be able to compare bundles, we have to create the representation of those bundles. This is performed in three steps. The first one is reading the bundle metadata information, in the second and third steps we follow the pointers from this metadata and go to bundle implementation to get the Java layer representation.

3.2.1 Component Metadata – First Step

Bundle manifest file acts as the point of first contact where the names of packages and other features are found. Bundle layer representation is built from this information. This step is trivial, because it means parsing a well specified text file (see example below).

```
Bundle-Name: LogService
Bundle-Version: 2.3.2
Export-Package: cz.zcu.logging;version="1.3.0"
Import-Package: org.osgi.framework
```

The next steps are more interesting, because the JavaType representation must be loaded from bytecode saved in the bundle jar file.

3.2.2 Exported Side of Component – Second Step

The classes for all exported features of a component must be naturally included in the component package itself. We can therefore construct their representation directly from the bytecode of the corresponding types.

The type reconstruction starts at the bundle level. For each exported package and service we create the corresponding *JPackage* or *JService* objects. Then, their

JClass contents needs to be filled in. The situation is trivial for the service case when only one class (the service interface) is referenced. For packages, the list of all contained classes is first obtained by querying the classloader and then expanded by creating *JClasses* using the reflection type loader.

Next we have to create the representation of all types referenced by public methods or fields of these classes because they will be used in the type-based bundle comparison. This process is bootstrapped by adding the *JClasses* from exported packages and services to the *knownTypesList* queue. Then an iterative algorithm for creating the whole transitive closure of interface types starts. All unprocessed types from *knownTypesList* are handled consecutively. For each type *T* from *knownTypesList*, the *JTypes* referenced by its members are retrieved. For each type *R* from these referenced types one of these possibilities is true:

- *R* is contained in the component and not in *knownTypesList* – add *R* to *knownTypesList*.
- *R* is contained in the component and already in *knownTypesList* – no action.
- *R* is not inside the component and its namespace is listed in an `import-package` header – a stub is created.
- *R* is not inside the component and its namespace is not in an `import-package` header – exceptional state.

When all referenced types of type *T* are processed, *T* is marked as unfolded and next type in the *knownTypesList* is processed with the same algorithm. The exceptional state is handled by throwing the appropriate exception, to indicate that the analysed bundle is invalid (referencing a type in code without corresponding imported package declaration means the bundle would not be resolved and started by the OSGi framework anyway).

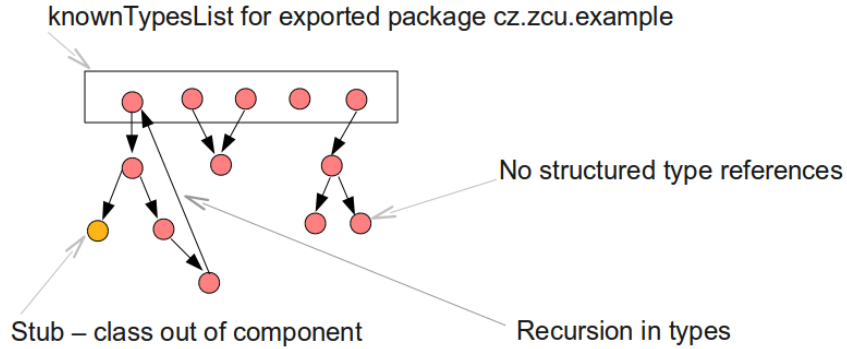


Fig. 6. Exported package – example

Because there is a danger of recursion in type dependencies, the algorithm must include a recursion detection instrument. For this purpose an additional stack data structure is used which saves the currently unfolded type branch sequence. On its bottom there is the *JClass* directly referenced from an `export-package` component

header. The stack contains the path from the currently processed node to this root. When a new referenced type is found, the stack is checked for containment of this type. If found, recursion was detected and the type is not expanded but the reference is pointed to the stacked instance.

In this way the whole tree of all *JTypes* is expanded. The tree is created because each *JClass* can reference another *JClass* as its field, method parameter, or method return type. The leaf nodes of this tree are of the following three kinds:

- Primitive types. In this trivial case there is no need to create children *JClasses*.
- Stub class. It means this type is contained in another bundle or library, and an empty stub class is created in its place.
- Reference to a recursively defined type. In this case this node is not a leaf, but the expansion of types ends.

3.2.3 Imported Side of Component – Third Step

The situation with the imported side of component is trickier. While for the features on the exported side a concrete bytecode of their types is available within the component package, the situation is the exact opposite for the imported side. This is an obvious consequence of the component-based decomposition of application functionality, as discussed in Section 2.

Our solution is to use the following approach to reconstruct the imported side. Each language type *a* imported by the component *C* which is really needed by its functionality is used by at least one type *r* inside *C*'s implementation. The compiler leaves an imprint of *a* in the bytecode of *r* containing the type signatures of the class as well as its members (fields and methods) used by *r*. When we create a union of all these signatures in *C*'s implementation, we get the complete structure of the *a* type as needed by the whole component.

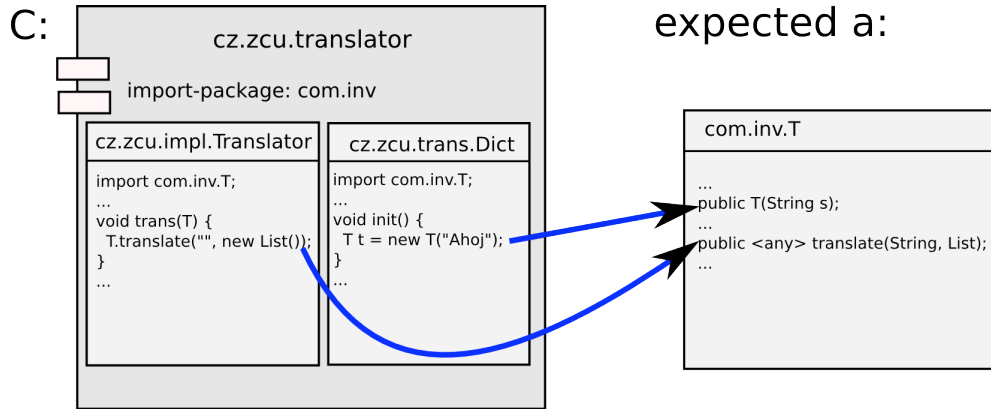


Fig. 7. Imported package – example

This idea can be used to create the representation of imported packages and services, because they are compounds of types. The principle is illustrated by Figure 7. In this example, the component *C* is *cz.zcu.translator* which imports the package *com.inv*. This package contains a referenced class *com.inv.T*, which is used by two types (*Translator* and *Dict*) in component *C*. The structure of the type *T* reconstructed from component's implementation consists of the two methods

deduced from the code snippets shown in the figure. The symbol $\langle \text{any} \rangle$ denotes any object type, primitive type or void; it acts as a supertype for all types (the calling convention in the bytecode does not contain enough information to reconstruct the precise return type of the operation’s signature).

Using this bytecode analysis technique, a similar structure as for exported packages is created. However, the root classes of the representation are created by custom creation (stubbing) described in the previous section.

In certain scenarios it is possible to use an alternative approach to reconstructing imported types. When the bytecode of bundle’s imported packages (e.g. their `.jar` files) is available during bundle analysis, we can reconstruct the representation from its “classpath”. For instance, when the bundle imports the package `cz.zcu.example` we will include all classes from this package in the representation of bundle’s imported side. In section 4 below we show that the assumption of available “classpath” is fulfilled for a large class of situations.

The first approach to obtaining imported types representation is more laborious but exactly matches the component’s real requirements. As such it actually provides more precise information than the representation created by analysing the imported packages themselves (the second approach). This advantage is used by the contextual compatibility evaluation [7] proposed earlier by one of the authors.

4 Component Compatibility Determination

The method of determining component compatibility we propose is based on evaluating the subtype relation between two components. Briefly, if type A can be used in all possible contexts of another type, then A is a subtype of the other one.

In this section we describe the algorithm of component type-based comparison. Because the focus of this paper is on type representation reconstruction, only the main principle will be illustrated through an example. We first describe a function used to compare type structures, then show some typical use cases for this method.

4.1 Component Type Differences

The result of comparing two types a and b can be described by the character of changes between them. Let us define the function $Diff(a, b) : Type \times Type \rightarrow Differences$ which computes the difference between types a and b . The returned value is one of:

- *None*: No change between a and b .
- *Spec*: Specialization – b is subtype of a .
- *Gen*: Generalization – a is subtype of b .
- *Mut*: Mutation – there is no subtype relation between a and b .

The value of $Diff()$ function for structured types is computed by combining the differences of their constituent parts. The exact algorithm of $Diff()$ value determination was published in [6] and is explained in Figure 8 where the package `cz.zcu.logging` is in two versions.

In the second version one method (`void setSize(int)`) in interface `Logger` was deleted and at the same time one method (`void flushAllLoggers()`) of another interface `LogService` was added. Whereas the `Logger` was generalized the `LogService` was specialized. These two changes in the same package are contravariant, so that the resulting difference is a Mutation of the type.

4.2 Differences and Compatibility

When we retrieve the value of $Diff(a, b)$ function we can use it to make a decision about a to b compatibility. The following table (1) shows the rules:

$Diff(a, b)$	<i>None</i>	<i>Specialization</i>	<i>Generalization</i>	<i>Mutation</i>
Type a is compatible to b	Yes	No	Yes	No
Type b is compatible to a	Yes	Yes	No	No

Table 1
Mapping of Difference values to Compatibility

4.3 Use Cases of The Method

The method described above is general – it “only” defines how to create representation of Java language types and how to use it in subtyping comparison to determine the level of type and component compatibility. In this section we provide a list of use cases in which the method is used now.

Automated Versioning

As described in detail in previous work [6], the bundle comparison method can be used for automated versioning of components. This process can simplify the error-prone task of assigning version identifiers to components and their features. The type differences described above can be used as an input to an automatic creation of version identifiers describing the real evolution of component interface. In the case of OSGi for example, the version numbering scheme is governed by rules which nicely map to the difference values. When using such automated versioning

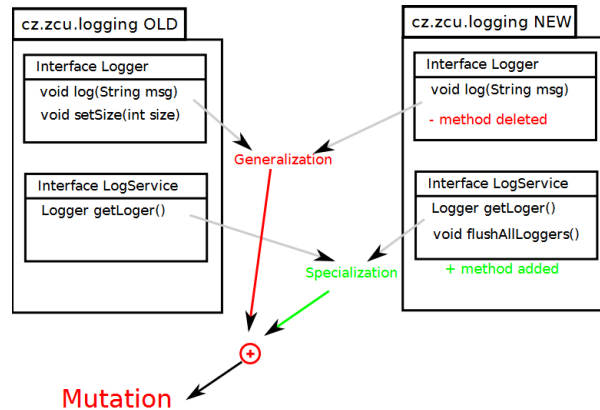


Fig. 8. Subtyping example

in a component system, its administrators can rely on the reliability of the version identifiers.

This use case applies the method at the bundle release time, when bundle type representations obtained by bytecode inspection of two last component revisions are compared. The first bundle in the comparison is the last previously released component with version identifier. The second bundle is the next release candidate for which we want to determine the version identifier.

With this approach released bundles carry version identifiers which describe not only the piece of software itself but also the changes it has undergone.

Safe Update

Another use case of the method is applicable at the deployment time of components. In this case we can use the subtyping comparison to ensure that the new version of a bundle is compatible with the previous one, regardless of the version numbers assigned to both (taking the conservative stand that their reliability is low and that a robust method is needed to ensure application type consistency). Alternatively, the method can be similarly applied to comparing a new version to the actual context of its deployment.

In this use case the method is applied at component deployment time. The representation of the old bundle version is obtained from reflection, including the imported side (resolved to existing package exporters). The representation of the new bundle version is obtained via bytecode analysis.

When using Safe Updater as an updating tool we can prevent the situation from introductory example in Section 2.4. The Safe Updater searches the interfaces of providers and importers recursively and verifies if types referenced by these two sides of the contract are compatible. The subtyping rules applied recursively guarantee those errors preclusion.

In this concrete use case we prevent nearly the same set of errors as would be found during JVM linking and verification processes. Contrary to them we can do these checks on demand without loading the bytecode to JVM.

Component Dependency Resolving with Checks

The last application scenario we mention is the process of resolving the components with additional subtyping checks. Resolving, similar to the linking stage of a compilation process, is used to bind imported packages to corresponding exporters. These bindings are made only on matching names of imports and exports.

The additional checks ensure that all mutual component interfaces in a component system are compatible with each other. Here, our method is applied at the start time of the component framework, after component installation or update. Both compared interface sides are loaded via reflection.

5 Related Work

In both research and industry world bytecode analysis and manipulation techniques are common. The ASM library [11] can be used to modify existing classes or dy-

namically generate classes and it is focused on simplicity of use and performance. Other frameworks with similar functionality are JMangler [15] or JavaAssist [10]. Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular. This approach is less efficient than ASM visitor design pattern, thus ASM is the best choice for dynamic systems.

A distinct class of frameworks uses XML for bytecode representation and manipulation [17,19]. They are comparable in features to the above approaches and support advanced operations including crosscompilation. The latter work discusses the need to transform or wrap API calls embedded in the bytecode.

Unlike JavaTypes, none of these approaches deals with the problems of reconstructing the referenced types not found in the analysed bytecode, which is a key need in the CBSE context. Another advantage of JavaTypes is the ability to compare the reconstructed types by subtyping rules. On the other hand, JavaTypes is not intended for bytecode manipulation and intentionally supports only a limited subset of Java language features.

Concerning the evaluation of component compatibility, there are two general methods. Dynamic assessment determines compatibility by running a regression test suite [12]. More closely related to our approach, McCamant et al [16] define compatibility based on observed (not declared) behaviour while Chaki et al [9] verify that global correctness properties are preserved through component updates, applying model checking on abstractions of component’s source code. These approaches are certainly more precise than compatibility based on type reconstruction used in our work. On the other hand it is much more difficult to obtain the required behavioural representations of a component. Our method could be used as a first check prior to expensive model checking is performed.

6 Conclusion

The ability to perform type-based compatibility checks is important for enhanced robustness of component applications. In this paper, we have described a supporting representation of the Java language types which constitute the interface of components together with a set of methods for obtaining this representation. Our system allows to use a mixed set of sources in these methods, including the Reflection API and bytecode analysis using the ASM tool.

Among the main challenges which our approach addresses are (i) the need to cover various stages of component development lifecycle – build, deployment, as well as runtime checks; (ii) limited access to some of the classes referenced by component’s interface types; (iii) faithful reconstruction of types of the imported (required) features from a standalone component package. The key contribution presented is the method for obtaining the real structure of the imported-side types from their parts referenced by the component’s bytecode implementation.

The methods described in this paper have been successfully used in several applications dealing with component representation and compatibility. They form a base for automated component versioning as well as a type-safe update mechanism implemented for the popular OSGi framework.

References

- [1] *Apache felix web console*.
URL <http://felix.apache.org/site/apache-felix-web-console.html>
- [2] *Asm website*.
URL <http://asm.ow2.org/>
- [3] *Bcel website*.
URL <http://jakarta.apache.org/bcel/>
- [4] Allen, E., *Diagnosing java code: The case for static types* (2002).
URL <http://www.ibm.com/developerworks/java/library/j-diag0625.html>
- [5] Bachmann, F. et al., *Volume II: Technical concepts of component-based software engineering*, Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University (2000).
- [6] Bauml, J. and P. Brada, *Automated versioning in OSGi: a mechanism for component software consistency guarantee*, in: *Proceedings of Euromicro SEAA* (2009).
- [7] Brada, P., “Specification-Based Component Substitutability and Revision Identification,” Ph.D. thesis, Charles University in Prague (2003).
- [8] Brada, P., *Enhanced OSGi bundle updates to prevent runtime exceptions*, in: *Proceedings of the 34th Euromicro SEAA conference* (2008).
- [9] Chaki, S., E. Clarke, N. Sharygina and N. Sinha, *Verification of evolving software via component substitutability analysis*, *Formal Methods in System Design* **32** (2008).
- [10] Chiba, S. and M. Nishizawa, *An easy-to-use toolkit for efficient java bytecode translators*, in: *Proceedings of the 2nd international conference on Generative programming and component engineering*, Springer-Verlag, New York, NY, USA, 2003, pp. 364–376.
- [11] E. Bruneton, R. Lenglet and T. Coupaye, *Asm: a code manipulation tool to implement adaptable systems*, in: *Adaptable and extensible component systems*, Grenoble, France, 2002.
- [12] Flores, A. and M. Polo, *Testing-based process for evaluating component replaceability*, in: *Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008)*, 2009, pp. 101 – 115, *Electronic Notes in Theoretical Computer Science*, vol. 236.
- [13] Forman, I. R., N. Forman, D. J. V. Ibm, I. R. Forman and N. Forman, *Java reflection in action* (2004).
- [14] James Gosling, G. S., Bill Joy and G. Bracha, *Java language specification, third edition* (2005).
- [15] Kniesel, G., P. Costanza and M. Austermann, *Jmangler-a framework for load-time transformation of java class files*, in: *IEEE International Workshop on Source Code Analysis and Manipulation* (2001).
- [16] McCamant, S. and M. D. Ernst, *Formalizing lightweight verification of software component composition*, in: *Proceedings of SAVCBS 2004: Specification and Verification of Component-Based Systems*, Newport Beach, CA, USA, 2004, pp. 47–54.
- [17] NoUnit Team, *NoUnit* (2006), accessed 12/2009.
URL <http://nunit.sourceforge.net/>
- [18] The OSGi Alliance, “OSGi Service Platform, Release 4,” (2005), available at <http://www.osgi.org/>.
- [19] Puder, A. and J. Lee, *Towards an XML-based bytecode level transformation framework*, in: E. Albert and S. Genaim, editors, *Preproceedings of 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, York, UK, 2009.
- [20] Szyperski, C., “Component Software, Second Edition,” ACM Press, Addison-Wesley, 2002.

Non-Intrusive Structural Coverage for Objective Caml

Philippe Wang¹, Adrien Jonquet², Emmanuel Chailloux³

*Équipe APR
Laboratoire d'Informatique de Paris 6 (CNRS UMR 7606)
Université Pierre et Marie Curie (Paris 6)
4 place Jussieu, 75005 Paris, France*

Abstract

This paper presents a non-intrusive method for Objective Caml code coverage analysis. While classic methods rewrite the source code to an instrumented version that will produce traces at runtime, our approach chooses not to rewrite the source code. Instead, we use a virtual machine to monitor instructions execution and produce traces. These low-level traces are used to create a machine code coverage report. Combined with control-flow debug information, they can be analyzed to produce a source code coverage report. The purpose of this approach is to make available a method to generate code coverage analysis with the same binary for testing and for production. Our customized virtual machine respects the same semantics as the original virtual machine; one of its original aspects is that it is implemented in the Objective Caml, the very language we build the tool for. This work is part of the *Coverage* project, which aims to develop open source tools for safety-critical embedded applications and their code generators.

Keywords: Certification Tools Design, Code Coverage, Objective Caml Virtual Machine

1 Introduction

One of the most demanding development process for safety-critical software was defined a couple of decades ago by the civil avionics authorities as the DO-178B standard [17]. This standard notably contains all constraints ruling aircraft software development. A very precise development process is imposed, and its preponderant activity is independent verification of each development step. Product specifications are written by successive refinements, from high-level requirements to design and then to implementation. Each step owns an independent verification activity, which must provide a complete traceability of the requirements appearing at this stage.

¹ Email: Philippe.Wang@lip6.fr

² Email: Adrien.Jonquet@lip6.fr

³ Email: Emmanuel.Chailloux@lip6.fr

The certification process, required before actually using such a software, mainly consists in making a specification and testing process document reporting that software specifications and implementation are tested by another entity to show that its behaviour conforms to its specifications.

Code coverage reports are part of the documents required by the certification process. They are generated from the program’s source code, and the execution traces of the compiled program. The classic approach to obtain the latter is to add instructions in the source code to produce them, while keeping the same semantics otherwise. For instance, the Esterel Technologies company developed such a tool for Objective Caml [14], a multiparadigm programming language [12], which is not widely used in the safety-critical domain. However it has already been successfully used for safety-critical development tools, e.g., a code generator written in Objective Caml is used in Esterel Technologies’ SCADE Suite [15].

A different approach consists of keeping the same program without adding instructions but instead to run it in a modified execution context. This approach means that the code coverage tools do not instrument the original program, so that the binary executed for code coverage testing purpose can be the exact same binary as for the final product. This is the core of the Coverage project ⁴ which interests on the QEMU virtualizer and the ZINC machine [11], the Objective Caml virtual machine, called ZAM ⁵ here after.

Both approaches should produce the same reports, but the non-intrusive way should shorten the traceability process because the exact same code can be executed for both functional testing and coverage testing.

In this paper, we focus on this Objective Caml multiparadigm programming language, which is distributed in an open source package that contains – among other things – a compiler and a virtual machine. One major motivation for using Objective Caml is that it has already been used with success in a certification framework.

We present ZAMCOV, a new ZAM implementation in Objective Caml, which produces traces at runtime for future code coverage analysis. This work will be compared to Esterel Technologies’ approach.

This paper is organized as follows: section 2 describes a past experiment on using Objective Caml in the safety-critical software domain; section 3 details the project in which our work takes part; section 4 presents the ZAM machine and our implementation, first step in non-intrusive code coverage process; section 5 shows non-intrusive code coverage process at the machine code level and how to go from machine code coverage to source code coverage with our tool available at <http://www.algo-prog.info/zamcov> and section 6 describes related work and announces our future work in this project.

⁴ This project is supported in part by the SYSTEM@TIC PARIS-REGION Cluster in the Free and Open Source Software thematic group (<http://www.projet-couverture.com/>). Two companies are involved in the development: AdaCore and OpenWide, together with two academic partners: Telecom ParisTech and University Pierre et Marie Curie (Paris 6).

⁵ ZAM stands for ZINC Abstract Machine

2 Structural Coverage in Objective Caml by Esterel Technologies

The French company Esterel Technologies ⁶ decided in 2006 to base its new SCADE SUITE 6™ ⁷ [4,5] certifiable code generator on Objective Caml. Esterel Technologies markets SCADE SUITE 6™, a model-based development environment dedicated to safety-critical embedded software. The code generator (KCG ⁸) of this suite that translates models into embedded C code is DO-178B compliant and allows to shorten the certification process of avionics projects which use it.

The DO-178B standard applies to embedded code development tools with the same criteria as the code itself. This means that the tool development must follow its own coding standard. The certification standard originally targeted only embedded software, so its application for a development tool must be adapted. For instance, for a code generator it is accepted to use dynamic allocation and have recursive functions. The specificity of the certification process for tools is under discussion to be explicitly addressed by the forthcoming DO-178C standard that should be effective soon.

2.1 Code Coverage and MC/DC (Modified Condition/Decision Coverage)

Among the numerous testing activities, one is making reports on code coverage. This activity has a set of constraints other than just showing whether some code is alive or dead: for instance, if a result is a complex Boolean expression, it is not enough to show that it has been evaluated (to any value). Neither is it enough to show it has taken both true and false values. Indeed, a complex Boolean expression is composed with sub Boolean expressions, and these also have to have taken both true and false values. Plus, if two subexpressions always return the same value, it is suspicious: are they duplicated?

As any activity during a DO-178B compliant development process, the verification activities are evaluated. Some criteria must be reached to decide that the task has been completed. One of these criteria is the activation of any part of the code during a functional test. On this particular point, more than a complete structural exploration of the code, the DO-178B standard requires that a complete exploration of the control flow has to be achieved following the MC/DC measurement that we explain below.

- A *decision* is the Boolean expression evaluated in a test instruction to determine the branch to be executed. It is covered if there exist tests in which it is evaluated to **true** and **false**.
- A *condition* is an atomic subexpression of a decision. It is covered if there exist tests in which it is evaluated to **true** and **false**.
- *MC/DC* requires that, for each condition c of a decision, there exist two tests which must change the decision value while keeping the same valuations for all

⁶ <http://www.esterel-technologies.com>

⁷ SCADE stands for *Safety Critical Application Development Environment*; *Scade* is the programming language provided by SCADE SUITE 6™.

⁸ KCG stands for *qualifiable Code Generator*.

conditions but c . It ensures that each condition can affect the outcome of the decision and that all contribute to the implemented function (no dead code is wanted).

MC/DC is properly defined on an abstract Boolean data flow language [10] with a classical automata point of view. The measure is extended to imperative programming languages, especially the C language, and is implemented in verification tools able to compute this measure.

2.2 *MLcov: an Objective Caml Code Coverage Tool*

MLcov [1] is an open source code coverage measurement tool for Objective Caml developed by Esterel Technologies. MLcov only treats the functional and imperative features of Objective Caml, which correspond to the subset allowed by the coding rules of the Scade-to-C compiler. This subset remains quite large, for instance, it is sufficient to compile the standard library of the Objective Caml distribution.

Coverage is measured by instrumenting the source code of the program. With respect to Objective Caml, we state that an expression is covered as soon as its evaluation ends. The main idea of the instrumentation algorithm is to replace each expression `expr` with `(let aux = expr in mark(); aux)`, where variable `aux` is not free in `expr`, and `mark()` is a side-effect allowing to record that this point of the program has been reached.

A program is structurally covered when every call to `mark()` in the instrumented source code has been reached. This instrumentation algorithm, detailed in [14] and consisting in adding a side-effect after each expression, systematically breaks tail calls, thus forbids this optimization.

2.3 *New certified KCG*

The new developed-in-Objective-Caml KCG is certified with respect to IEC 61508 and EN 50128 norms. It is used in several civil avionics DO-178B projects (e.g., for the A380 Airbus plane) and will be qualified simultaneously to the project qualifications (with DO-178B, the tools are not qualified by themselves, but by their usage in a project).

3 **Code Coverage with Non-Intrusive Tools: The Coverage Project**

The Coverage project, which started a year ago, aims at providing non-intrusive coverage tools in a free software/open source context for safety-critical applications.

In the Coverage project, the main idea is not to instrument the code directly but instead to instrument the runtime environment which executes the code as shown in figure 1. This execution produces some traces which can be analysed offline (i.e. after the execution) and mapped back to the original program source. In this case the final machine code will be executed in a special runtime context.

Two (language \times target machine) approaches are studied: the first is (Ada language \times PowerPC processors family), the second is (Objective Caml language \times

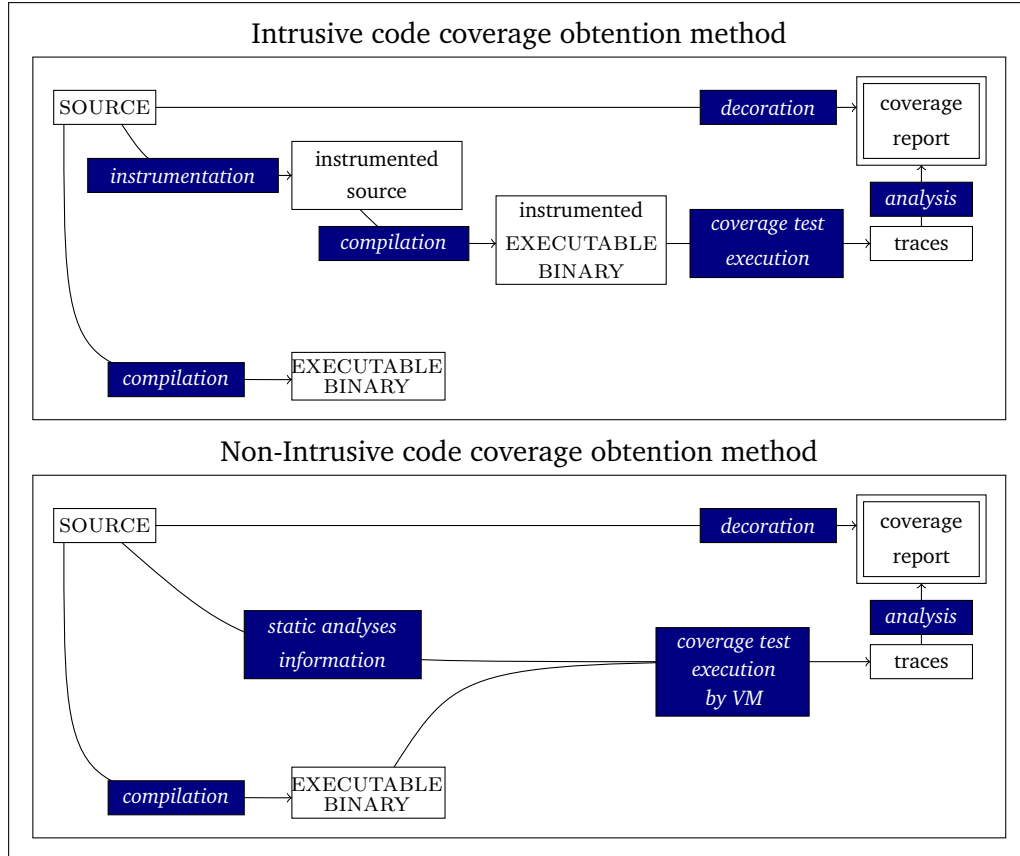


Figure 1. Code Coverage Obtaining Methods Comparison

ZAM its own virtual machine). These two couples are used for safety-critical embedded applications (Ada) and code generators (Objective Caml), including avionics projects using the DO-178B standard.

For common traditional processors, QEMU [2] is used as a free-software emulator (Power-PC, ...) which can generate traces. This allows non-intrusive analysis on final target code with emulators running on development hosts. In this part, the Adacore⁹ company develops tools which are independent from the language, like Ada or C, and from the compiler, by using source DWARF [3] debug info. This independence implies additional yet several restrictions for MC/DC.

For virtual machines, we have studied Objective Caml virtual machine to produce traces. These traces are analyzed after the execution to produce a structural coverage report for machine code and source code. To make the link between machine code and source code, we use debug information added by the Objective Caml compiler in the debug mode. This information, called events, indicates the beginning or the end of an expression. With these events, the control flow graph of a program can be rebuilt during or after an execution. For MC/DC analysis more information is needed, so the original Objective Caml compiler has to be modified. This will be discussed in section 6.

At mid-term of the project, we can present our progression in the two following

⁹ <http://www.adacore.com>

sections on machine code and source code coverage.

4 An Objective Caml Virtual Machine in Objective Caml

Generating a machine code coverage report first means to know the compiled program's binary format and semantics to interpret it, this work is done by a virtual machine. Then, during interpretation, one needs to keep the execution flow for further analysis; this work is done by a component plugged into the virtual machine. In our case, we implemented an Objective Caml virtual machine such that it is easy to extend with pluggable components. For this, we chose Objective Caml as the implementation language, for several reasons:

- it is interesting to implement a virtual machine in the very language it is designed for;
- it brings the bootstrap challenge, so the resulting tool can be used for itself;
- Objective Caml is strongly statically typed, so the interpreter does not use unsafe type casts
- and it permits to build applications that are naturally robust to components plugins.

4.1 *The Objective Caml Virtual Machine (ZAM)*

ZAM is a stack-based virtual machine for a functional-based multiparadigm programming language. It only uses 7 registers: an accumulator to store a value, a code pointer (next instruction to interpret), a stack pointer, another stack pointer for the highest exception handler, an extra arguments counter, an environment (a value array) and a global data (a value array). ZAM interprets 146 different instructions, about 60% of which are shortcuts for several instructions combinations. 18 instructions are for arithmetic and Boolean operations.

Values are uniformly represented, it makes exploring a value easy, notably for the garbage collection system. A value is either a integer encoded on $32-1=31$ bits or $64-1=63$ bits depending on the architecture, or a block value whose header encodes a block tag on one byte (e.g., closure tag, string tag, double tag or variants) and a block size. This integer representation is actually an optimization: since it is often sufficient to have 31 or 63-bit (signed) integers, and since the weakest bit of an address is never set to 1, it is possible to use 32 or 64-bit integers to represent immediate integers. As a consequence, the compiler will automatically convert some int values: e.g., 0 becomes 1, 1 becomes 3, n becomes $n \times 2 + 1$, and arithmetic operations are modified in consequence. Since increasing arithmetic operations has a lower cost than having a pointer dereferencing, the loss of performance is acceptable. An instruction is dedicated to recognizing an integer from a block: `IS_INT`.

Objective Caml has functional values that are encoded as closures (functions \times environment), some specific instructions handle to them (e.g., `APPLY`, `CLOSURE`, `CLOSUREREC`, `GRAB`, `OFFSETCLOSURE`).

4.2 Existing Objective Caml Virtual Machine Implementations

The Objective Caml virtual machine has several implementations, the original one is in C, one is in Java and another is in JavaScript. They are quickly described as follows.

- **In C code:** The INRIA standard distribution provides a virtual machine implemented in C code, it is likely the most efficient implementation available. Its written-in-C runtime library is the same as the one used with hardware machine programs that are produced by the native compiler.
- **In Java code:** The Java implementation, called Cadmium [7], allows an Objective Caml program to be executed on any machine that has a Java Virtual Machine, without having to install the whole Objective Caml system. For instance, this can be used to easily run Objective Caml programs on a web page. Parts of its runtime library rely on Java runtime library such as garbage collection, the other part is in Java.
- **In JavaScript:** The JavaScript implementation, called O'Browser [6], gives the possibility to write dynamic web page components (that are usually written in JavaScript) in Objective Caml. As it is not relevant to have exactly the same runtime library as the original distribution, an alternative version provides an interface with web page related functions.

4.3 Our New Implementation in Objective Caml itself

It is important to note that whereas Objective Caml is strongly statically typed, its virtual machine is untyped. This design was motivated by the guarantee that static type checking process frees the runtime process from making any type checks. Writing an Objective Caml virtual machine in Objective Caml implies writing an untyped virtual machine for a strongly typed programming language in a strongly typed programming language. It is analog with the runtime library: Objective Caml runtime library has two parts: the low-level part is a set of C functions that may access low-level data representations, and the high-level part is a set of Objective Caml functions that may use functions implemented in C code.

We chose to implement ZAMCOV values type as follows:

```

type tag =
  | Structured_tag of int
  | Closure_tag
  | Object_tag
  | Abstract_tag
  | ...

type value =
  | Int of int
  | Float of float
  | String of string
  | Block of block
  | ...

and block = { tag : tag; data : value array; ... }

```

From a bytecode binary, ZAMCOV initializes a virtual machine data structure with the instructions section, the global data section and the set of external functions. Then the interpreter function is linked to this data structure to make it able to run instruction by instruction: it is easy to plug a component into the interpreter.

The original standard library is a set of Objective Caml functions, some of which call some C code. For instance, operations on files are implemented with C functions encapsulated for Objective Caml. When compiling an Objective Caml function to bytecode instructions, there are two cases:

- the underlying functionality is in C code: the bytecode will contain `C_CALL` instructions ¹⁰;
- it is fully in “pure” Objective Caml: bytecode instructions do not contain `C_CALL` instructions.

The first case is not trivial with an alternative bytecode interpreter. In our case, with ZAMCOV, `C_CALL` instructions will be interpreted by an Objective Caml function, and will mean calling an Objective Caml function. Thus, for instance, when calling a I/O operation (or any operation that cannot be directly represented by some bytecode instructions), an indirection is added. The source code in Objective Caml is compiled to bytecode, which is then interpreted by an Objective Caml program.

For instance, to call a C function `foo` from the original virtual machine, the `C_CALL` instruction is used with `"foo"` as first argument and it will call the C function. This C function cannot be called directly at the interpretation of a `C_CALL` instruction, because our value representation is different.

Implementing the runtime library is a weird constraint: the original runtime library is implemented in low-level C code, and this is a behaviour that has to be reproduced in our Objective Caml runtime library. For instance, comparison functions (which compare data structures in depth) are based on the comparison function (`val compare : 'a -> 'a -> int`) which is implemented in C code as part of the runtime library, and since our data representation is not exactly the same, we cannot use it directly when the machine code invokes function `compare` as a C call. This means we had to implement in Objective Caml the comparison function for our data representation that emulate the original data representation.

¹⁰ There is a set of “C call” instructions (`C_CALL[1-5]`, `C_CALL_N`) that allows the bytecode to call external functions (i.e. functions that are not to be compiled to bytecode).

Indeed, one constraint was not to break the type checker because otherwise implementing first draft of the virtual machine would have been quicker but its debugging would have been a nightmare.

5 ZAMCOV's code Coverage Tools

5.1 Execution trace generation

The first component plugs itself in the instructions interpreter and keeps trace of which instructions are executed. Thanks to the design of our virtual machine implementation, it is quite easy to write and plug a component into it. The trace is an array whose length equals the code section's length of the bytecode. When an instruction of the code section is executed, the trace's element whose index is the address of the instruction in the bytecode is marked as "covered". The Objective Caml array, which contains the execution trace, is serialized in an external file when ZAMCOV ends the interpretation of the program.

5.2 Machine Code Coverage

Traces are analyzed after the execution to generate an instruction coverage report (in HTML format). The first report is a machine code coverage report. This report represents a list of all non-covered (never executed) bytecode instructions.

Here is an example of the machine code coverage report of a simple factorial program:

```
let rec fact x =
  if x = 0 then 1
  else if x = 1 then 1
  else x * (fact (x-1));;

fact 5;;
```

This program is a simple function computing the factorial. Application fact 5 does not allow the first test ($x = 0$) to become true, therefore the first branch, which returns 1, is not taken.

```
ocamlc fact.ml -o fact
zamcov-run -trace fact.trace fact
zamcov-cover fact.trace fact
```

First we need to compile the fact.ml file with standard ocamlc compiler. Then, with zamcov-run we interpret and generate the execution trace of the program fact and we build the coverage report with zamcov-cover.

00000002	ACC0
00000003	BNEQ
00000006	CONST1
00000007	RETURN
00000009	ACC0
00000010	BNEQ
00000013	CONST1
00000014	RETURN

The code sample on the left shows part of the machine code coverage result for program factorial. There are some boxed instructions: they are never executed and correspond to the code which returns the constant 1, which indeed is not executed in the factorial example.

Then one question was to know whether there was an equivalence between machine code coverage and source code coverage, as such an equivalence would

remove the need for source code coverage. We will see in the rest of the paper that they are not equivalent.

5.3 Source Code Coverage

The central role of ZAMCOV’s virtual machine is to interpret Objective Caml programs compiled to bytecode by standard distribution’s compiler `ocamlc`. This section presents a ZAMCOV component that generates the execution trace (representing all executed bytecode) instructions during the interpretation.

This trace only contains information about bytecode instructions, such as instructions names or their addresses in the bytecode. Hence, we need a mean to link this information with the source code.

Using Debug Events to Generate a Code Coverage Report

Debug events are debug information added by the compiler when using debug option “-g”. They are used by `ocamldebug`, the Objective Caml debugger. These events are not in program’s code section. There are not differences in this section for a program compiled with or without the debug option. This is important because ZAMCOV is a non-intrusive code coverage tool, so it is not supposed to modify the source code or the bytecode of the program. Debug events are located in an independent section of the program binary, not in the code to be interpreted, so they could be in a separate file only visible by the virtual machine if needed.

A debug event is a data structure linked with an Objective Caml expression during the compilation with the standard compiler (`ocamlc`). Debug events are located strategically in an Objective Caml program as shown in figure 2.

A debug event contains a lot of information about its expression:

- the location of the expression in the source code;
- the first bytecode instruction address corresponding to the Objective Caml expression.

The address in the bytecode recorded by the debug event is the missing link with the execution trace.

Coverage of Objective Caml expressions

ZAMCOV is also a source code coverage measurement tool. First coverage level is statement coverage. Objective Caml is a functional-based language (a program is an expression evaluation), so every “statement” is actually an “expression”. In Objective Caml, it is more appropriate to report an “expression coverage”. This kind of source code coverage checks if all Objective Caml expressions written in the source code are evaluated at least once. An expression coverage tool must show in coverage reports which expressions are not evaluated in the source code. These expressions are called “non-covered expressions” (or dead code), and evaluated expressions are called “covered expressions”.

ZAMCOV uses debug events and execution traces to check which expressions in the source code are covered or non-covered. Debug events contain information to associate bytecode instructions addresses to their corresponding source code. So,

We define F and G recursively as the functions that respectively place debug events in expressions and pattern-matching branches.

- P represents a pattern-matching branches set, unfolded as $p_i [\text{when } c_i] \rightarrow e_i$ which is a shortcut for $p_0 [\text{when } c_0] \rightarrow e_0 \mid \dots \mid p_n [\text{when } c_n] \rightarrow e_n$;
- p or p_n represent a pattern, which is a variable identifier or a structural accessor;
- e, e_n or c_n represent an expression;
- and $\$$ represents a debug event.

$$F(\text{atom}) = \text{atom} \text{ (constant value or identifier)}$$

$$F(e_0 \ e_1) = (F(e_0)) \ \$ \ (F(e_1)) \ \$$$

$$F(\text{let } [\text{rec}] \ p = e_0 \text{ in } e_1) = \text{let } [\text{rec}] \ p = F(e_0) \text{ in } F(e_1)$$

$$F(\text{fun } p_0 \dots p_n \rightarrow e) = \text{fun } p_0 \dots p_n \rightarrow \$ \ F(e)$$

$$F(\text{function } P) = \text{function } G(P)$$

$$F(\text{match } e \text{ with } P) = \text{match } F(e) \text{ with } G(P)$$

$$F(\text{try } e \text{ with } P) = \text{try } F(e) \text{ with } G(P)$$

$$F(e_0; e_1) = (F(e_0); \$ \ F(e_1))$$

$$F(\text{if } e_0 \text{ then } e_1 [\text{else } e_2]) = \text{if } F(e_0) \text{ then } \$ \ F(e_1) [\text{else } \$ \ F(e_2)]$$

$$F(\text{while } e_0 \text{ do } e_1 \text{ done}) = \text{while } F(e_0) \text{ do } \$ \ F(e_1) \text{ done}$$

$$F(\text{for } i = e_0 \text{ to } e_1 \text{ do } e_2 \text{ done}) = \text{for } i = F(e_0) \text{ to } F(e_1) \text{ do } \$ \ F(e_2) \text{ done}$$

$$F(e_0 \# m) = F(e_0) \# m \ \$$$

where m represents the name of a method

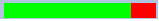
$$G(P) = p_i [\text{when } F(c_i)] \rightarrow \$ \ F(e_i)$$

Figure 2. Debug Events Placement in Expressions

to report source code coverage, for each debug event, if its associated bytecode instructions have been activated according to the execution traces, then its associated source code is covered. If a debug event cannot be related with the execution trace, it means that the associated expression is non-covered.

```
ocamlc -g fact.ml -o fact
zamcov-run -trace fact.trace fact
zamcov-cover fact.trace fact
```

This time, we compile the .ml file with option “-g” given to the standard ocamlc compiler to generate debug events for zamcov-cover to produce the source code coverage. Then, as for machine code coverage, with zamcov-run we interpret and generate the execution trace of the program fact and we build the coverage report with zamcov-cover.

Coverage report	
Trace Filename	Program Name
fact.trace	fact
Source Filename	Expression coverage
fact.ml	85% 6/ 7
	
Object code coverage report	
fact	

There is on the left an expression coverage report generated by ZAMCOV of a simple Objective Caml program (factorial). This report shows the source code files list of the coverage-measured program. Each file has a expression coverage rate which shows the difference between the number of non-covered debug events and the total number of debug events. For each file, there is a link to its source code coverage. There is also an HTML link to the machine code coverage of the program.

ZamCov: Expression Coverage (fact.ml)
<pre> let rec fact x = if x = 0 then 1 else if x = 1 then 1 else x * (fact (x-1));; fact 5;; </pre>

Link “fact.ml” allows to obtain its source code coverage page. In the screenshot on the left, text highlighted in green (light gray) is executed code (covered), and in red (dark gray) is code that has never been executed (non-covered).

ZamCov: Expression Coverage (fact.ml)
<pre> let rec fact x = if x = 0 then 1 else if x = 1 then 1 else x * (fact (x-1));; fact 5;; fact 0;; </pre>

$x = 0$ is never executed when calling fact 5, that is why the first 1 is non-covered. The structural coverage of this example is not complete. For function fact to be fully covered (expression coverage), we need to add more tests as shown in the screenshot on the left.

6 Related and Future Work

6.1 Other coverage tools in Objective Caml

The official Objective Caml distribution provides two profiler tools. The first one – for bytecode programs – instruments original source and counts each computed expression. The second one modifies the native code generator to produce information which can be used by gprof [9].

Ocamlviz [16] is a new graphical tool for real-time profiling in Objective Caml. It uses alarms to collect and send data. These data can be processed by a graphical interface during execution.

These different tools cannot produce MC/DC report, the only MC/DC coverage tool for Objective Caml is MLCOV that we described in section 2.2. All these tools are intrusive.

6.2 Other coverage tools for different virtual machines

For more classical virtual machines, as the Java Virtual Machine (JVM) or the Common Language Runtime (CLR) of the .NET environment, we find a lot of libraries to build debug tools. They offer a set of services which exposes runtime events that occur during the execution. In Java, JVTI (Java Virtual Machine Tool Interface) ¹¹ allows to write agents which can be notified of interesting occurrences through events. In .NET, the CLR Profiling API can provide notification of many activities within the CLR and managed code.

A good overview describing different ways to instrument Java code is presented in [8]. This bibliographical study compares different static and dynamic instrumentation techniques at source or bytecode level, including hybrid combinations, for Java. Examples using a specialized virtual machine are scarce, mainly for portability and efficiency criteria which can be important for monitoring or optimizing tools.

In our case, portability is guaranteed because we use the same runtime with and without execution traces. The loss of performance efficiency with ZAMCOV is acceptable for this kind of tools.

6.3 MC/DC for ZAMCOV

ZamCov: Expression Coverage (abs.ml)

```
let abs x =
  let y = ref 0 in
    if x < !y then y := -x;
    !y;;
abs (-5);;
```

The next objective of ZAMCOV is to offer a Decision, Condition and MC/DC measurement tool. ZAMCOV will need to identify Boolean expressions evaluation at run-time. Notably, a complete *statement* coverage is not equivalent to a decision coverage. In the example on the left, all the statements are covered but the decision only takes value false.

ZAMCOV needs to analyse these values to generate a report that shows in the source code which decisions satisfy MC/DC and which don't. The operation needs to go back to the source.

The main issue is that we need to recognize Boolean expressions in the machine code, and this is not possible without specific source code analysis information. Indeed, as the machine code is untyped, it is not possible to know the difference between an integer and a Boolean value, it is neither possible to know in all cases if a branch is introduced by a conditional expression or by a Boolean operator, or even a pattern-matching filter.

Adding new debug events requires the modification of Objective Caml's bytecode compiler. Indeed, we need to be able to identify `&&`, `||` and `not` Boolean operations in the source code and link them with the machine code to produce Boolean vectors at run-time.

¹¹ JVTI has replaced the JMVPI (JVM Profiler interface) [13] and JVMDI (JVM Debug Interface).

7 Conclusion

In this paper, we have presented a new approach for structural code coverage analysis without code instrumentation but only runtime environment instrumentation. This approach has been used to build ZAMCOV, a tool dedicated to Objective Caml's virtual machine. Our criteria of success is to produce the same reports as MLCOV, the open source code coverage measurement tool developed by Esterel Technologies for their certifiable code generator (KCG) written in Objective Caml. It will be reached for expression coverage (statement coverage) without any change in the Objective Caml compiler. But for MC/DC coverage, it will be mandatory to add new debug information systematically around Boolean expressions to check the condition/decision coverage to produce traces which can be analysed to measure the MC/DC coverage.

This approach can be used for any compiler that generates ZAM code with the appropriate events, if need be. This indicates a strong link between the compiler schemes and the debug events to map back to the original source.

It can be surprising to associate Objective Caml and bytecode for safety-critical software development tool. But the Esterel experiment has opened this way by using Objective Caml in a complete certification process. The introduction of virtual machine to build certifiable development tool is interesting for its non-intrusive approach: real code is analysed and not an equivalent but instrumented code.

This work takes place in the Coverage Project which studies non-intrusive coverage tools for Ada (to Power-PC) and Objective Caml (to ZAM). In the first case the QEMU emulator is used and in the second the ZAM virtual machine. But the compiler information needed by the modified runtime environment for the MC/DC measurement are similar for both languages.

Finally this work makes the link between two communities: DO-178B world and free open source software, by building the first part of a non-intrusive structural coverage tool. It joins the effort for openDO ¹², towards a cooperative and open framework for the development of certifiable software.

References

- [1] *MLcov*, <http://www.algo-prog.info/mlcov>.
- [2] *Qemu documentations*, <http://www.qemu.org>.
- [3] *The DWARF Debugging Standard* (2007), <http://dwarfstd.org>.
- [4] Berry, G., *The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems*, Technical report, Esterel-Technologies (2003).
- [5] Camus, J.-L. and B. Dion, *Efficient Development of Airborne Software with SCADE SuiteTM*, Technical report, Esterel-Technologies (2003).
- [6] Canou, B., V. Balat and E. Chailloux, *O'Browser : Objective Caml on Browsers*, in: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML The 2008 ACM SIGPLAN Workshop on ML*, 2008, pp. 69–78, <http://www.pps.jussieu.fr/~canou/obrowser/tutorial/>.
- [7] Clerc, X., *Cadmium* (2007), <http://cadmium.x9c.fr>.

¹² <http://www.open-do.org/>

- [8] Delahaye, M., *Instrumentation of Java code : bibliographical study (in French)* (2007), [ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2007/biblio.delahaye.mickael.pdf](http://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2007/biblio.delahaye.mickael.pdf).
- [9] Graham, S. L., P. B. Kessler and M. K. McKusick, *gprof: a call graph execution profiler* (1982).
- [10] Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski and L. K. Rierson, *A Practical Tutorial on Modified Condition/Decision Coverage*, Technical report, NASA/TM-2001-210876 (2001).
- [11] Leroy, X., *The ZINC Experiment : an Economical Implementation of the ML Language*, Technical Report 117, INRIA (1990).
- [12] Leroy, X., *The Objective Caml system release 3.10 : Documentation and user's manual*, Technical report, Inria (2008), <http://caml.inria.fr>.
- [13] O'Hair, K., *The JVMPI Transition to JVMTI* (2004), <http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition>.
- [14] Pagano, B., O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot and P. Wang, *Certified Development Tools Implementation in Objective Caml*, in: P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages (PADL 08)*, Lecture Notes in Computer Science **4902** (2008), pp. 2–17.
- [15] Pagano, B., O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury and J.-L. Colaço, *Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework*, in: *International Conference of Functional Programming (ICFP 09)*, 2009.
- [16] Robert, J. and G. V. Tokauski, *Ocamlviz : reference manual* (2009), <http://ocamlviz.forge.ocamlcore.org>.
- [17] RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* (1992), Radio Technical Commission for Aeronautics RTCA.

Encoding the Java Virtual Machine's Instruction Set

Michael Eichberg¹ Andreas Sewe²

*Department of Computer Science
Technische Universität Darmstadt
Germany*

Abstract

New toolkits that parse, analyze, and transform Java Bytecode are frequently developed from scratch to obtain a representation suitable for a particular purpose. But, while the functionality implemented by these toolkits to read in class files and do basic control- and data-flow analyses is comparable, it is implemented over and over again. Differences manifest themselves mainly in minor technical issues. To avoid the repetitive development of similar functionality, we have developed an XML-based language for specifying bytecode-based instruction sets. Using this language, we have encoded the instruction set of the Java Virtual Machine such that it can directly be used, e.g., to generate the skeleton of bytecode-based tools. The XML format hereby specifies both the format of the instructions and their effect on the stack and the local registers upon execution. This enables developers of static analyses to generate generic control- and data-flow analyses, e.g., an analysis that transforms Java Bytecode into static single assignment form. To assess the usefulness of our approach, we have used the encoding of the Java Virtual Machine's instruction set to develop a framework for the analysis and transformation of Java class files. The evaluation shows that using the specification significantly reduces the development effort when compared to manual development.

Keywords: Java Bytecode, Java Virtual Machine Specification, XML

1 Introduction

The development of programs that parse and analyze Java Bytecode [9] has a long history and new programs are still developed [2,3,4,7,13]. When developing such tools, however, a lot of effort is spent to develop a parser for the bytecode and for (re-)developing standard control- and data-flow analyses which calculate, e.g., the control-flow graph or the data-dependency graph.

To reduce these efforts, we have developed a specification language (OPAL SPL) for encoding the instructions of stack-based intermediate languages. The idea is that—once the instruction set is completely specified using OPAL SPL—generating both bytecode parsers and standard analyses is much easier than their manual development. To support this goal, OPAL SPL supports the specification of both

¹ E-mail: eichberg@informatik.tu-darmstadt.de

² E-mail: sewe@st.informatik.tu-darmstadt.de

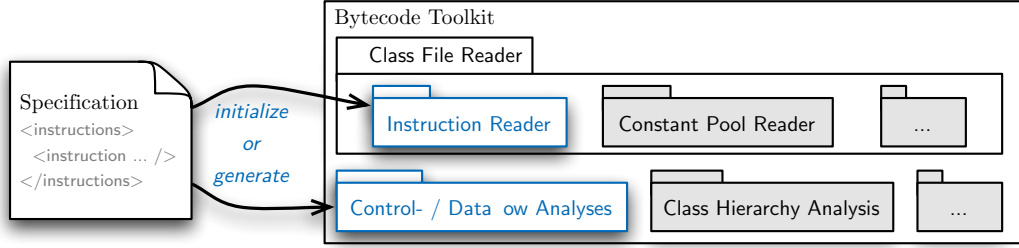


Fig. 1. Use Cases for OPAL SPL Specifications

the format of bytecode instructions and the effect on the stack and registers these instructions have when executed. An alternative use of an OPAL SPL specification is as input to a generic parser or to generic analyses as illustrated by Fig. 1.

Though the language was designed with Java Bytecode specifically in mind and is used to encode the complete instruction set of the Java Virtual Machine (JVM),³ we have striven for a Java-independent specification language. In particular, OPAL SPL focuses on specifying the instruction set rather than the complete class file format, not only because the former’s structure is much more regular than the latter’s, but also because a specification of the instruction set promises to be most beneficial. Given the primary focus of OPAL SPL—generating parsers and facilitating basic analyses—we explicitly designed the language such that it is possible to group related instructions. This makes specifications more concise and allows analyses to treat similar instructions in nearly the same way. For example, the JVM’s `iload.5` instruction, which loads the integer value stored in register `#5`, is a special case of the generic `iload` instruction where the instruction’s operand is `5`. We also designed OPAL SPL in such a way that specifications do not prescribe how a framework represents or processes information; i.e., OPAL SPL is representation agnostic.

The next section describes the specification language. In Section 3 we reason about the language’s design by discussing the specification of selected JVM instructions. In Section 4 the validation of specifications is discussed. The evaluation of the approach is presented in Section 5. The paper ends with a discussion of related work and a conclusion.

2 Specifying Bytecode Instructions

The language for specifying bytecode instructions (OPAL SPL) was primarily designed to enable a concise specification of the JVM’s instruction set. OPAL SPL supports the specification of both an instruction’s format and its effect on the stack and local variables (registers) when the instruction is executed. It is thus possible to specify which kind of values are popped from and pushed onto the stack as well as which local variables are read or written. Given a specification of the complete instruction set the information required by standard control- and data-flow analyses is then available.

However, OPAL SPL is not particularly tied to Java as it abstracts from the particularities of the JVM Specification. For example, the JVM’s type system is

³ The complete specification is available at <http://www.michael-eichberg.de/opal>.

part of an OPAL SPL specification rather than an integral part of the OPAL SPL language itself.

Next, we first give an overview of the language before we discuss its semantics.

2.1 Syntax

```

1. INSTRUCTIONS ::= instructions < TYPES EXCEPTIONS FUNCTIONS INSTRUCTION+ >
2. TYPES ::= types < TYPEDEF > // a common root type is required
3. TYPEDEF ::= type @name @pc? < TYPEDEF* >
4. EXCEPTIONS ::= exceptions < (exception @type)+ >
5. FUNCTIONS ::=
    functions (< function @name < signature < (param @type)* (returns @type) > > >)*
6. INSTRUCTION ::=
    instruction @mnemonic @deprecated? @transfers_control?
    < appinfo < /APPLICATIONSPECIFICCONTENT* >
    format < SEQUENCE+ >
    ( stack < (form < before < BEFOREOP+ > after < AFTEROP+ > >)+ > )?
    ( registers < LOAD? STORE? > )?
    ( exceptions < (exception @type)+ > )? >
7. SEQUENCE ::= sequence
    ( SEQUELEM | padding_bytes @alignment | list @count @var < SEQUELEM+ > |
      (implicit @var < /VALUEEXPRESSION > ) | (implicit_type @type < /TYPEEXPRESSION > ) )+
8. SEQUELEM ::= { u1 | u2 | u4 | i1 | i2 | i4 } @type? @var? < /EXPECTEDVALUE? >
9. BEFOREOP ::= ((operand @type @var?) | (list @loop_var? @count < BEFOREOP > ) ) * rest?
10. AFTEROP ::= (operand @type < /VALUEEXPRESSION? > ) rest?
11. LOAD ::= load @type @index @var
12. STORE ::= store @type @index < /VALUEEXPRESSION >

```

Fig. 2. Grammar of the OPAL Specification Language (OPAL SPL)

The OPAL Specification Language (OPAL SPL) is an XML-based language. Its grammar is depicted in Fig. 2 using an EBNF-like format. Non-terminals are written in capital letters (INSTRUCTIONS, TYPES, etc.), the names of XML-elements are written in small letters (types, stack, etc.) and the names of XML-attributes start with “@” (@type, @var, etc.). We refer to the content of an XML-element using symbols that start with “/” (/VALUEEXPRESSION, /EXPECTEDVALUE, etc.). “<>” is used to specify nesting of elements. “(),?,+,*,{},|” have the usual semantics. For example, `exceptions < (exception @type)+ >` specifies that the XML-element `exceptions` has one or more `exception` child elements that always have the attribute `type`.

2.2 Semantics

Format Specification

Each specification written in OPAL SPL consists of four major parts (line 1 in Fig. 2). The first part (`types`, lines 2–3) specifies the type system that is used by the underlying virtual machine. The second part (`exceptions`, line 4) declares the exceptions that may be thrown when instructions are executed. The third part (`functions`, line 5) declares the functions that are used in instruction specifications. The fourth part is the specification of the instructions themselves (lines 6–12), each of which may resort to the declared functions to access information not simply stored along with the instruction. For example, `invoke` instructions do not store the signature and declaring class of the called methods. Instead, a reference to an entry in the so-called constant pool is stored. Only this constant pool entry has all information about the method. To obtain, e.g., the return type of the called method, an abstract function `TYPE methodref.return_type(methodref)` is declared that takes a reference

to the entry as input and returns the method’s return type. Using abstract function declarations, we abstract—in the specification of the instructions—from the concrete representation of such information by the enclosing bytecode toolkit.

The specification of an instruction consists of up to four parts: the instruction’s format (lines 7–8), a description of the effect the instruction has on the stack when executed (lines 9–10), a descriptions of the registers it affects upon execution (lines 11–12), and information about the exceptions that may be thrown during execution (end of line 6). An instruction’s format is specified by sequences which describe how an instruction is stored. The `u1`, `u2` and `u4` elements (line 8) of each format sequence specify that the current value is an unsigned integer value with 1, 2 and 4 bytes, respectively. Similarly, the `i1`, `i2` and `i4` elements (line 8) are used to specify that the current value is a (1, 2 or 4 byte) signed integer value. The values can be bound to variables using the `var` attribute and can be given a second semantics using the `type` attribute. For example, `<i2 type="short" var="value" />` is a two-byte signed integer value that is bound to the variable `value` and has type `short` with respect to the instruction set’s type system. Additionally, it is possible to specify expected values (line 8). This enables the selection of the format sequence to be used for reading in the instruction. E.g., `<sequence><u1 var="opcode">171</u1>...` specifies that this sequence matches if the value of the first byte is 171. A sequence’s list element is used to specify that a variable number of values need to be read. The concrete number of elements is determined by the `count` attribute. The attribute’s value is an expression that can use values that were previously assigned to a variable. The sequence elements `implicit` and `implicit_type` are used to bind implicit value and type information to variables that can later on be used in type or value expressions (line 7, 10 and 11). To make it possible to aggregate related bytecode instructions to one logical instruction, several format sequences can be defined. The effect on the stack is determined by the number and type of stack operands that are popped (line 9) and pushed (line 10). If multiple stack layouts are specified, the effect on the stack is determined by the first *before-execution* stack layout that matches; i.e., to determine the effect on the stack a data-flow analysis is necessary.

Unique Prefix Rule

One constraint placed upon specifications written in OPAL SPL is that a format sequence can be identified unambiguously by only parsing a prefix of the instruction; no lookahead is necessary. In other words, if each format sequence is considered a production and each `u1`, `u2`, etc. is considered a terminal, then OPAL SPL requires the format sequences to constitute an LR(0) grammar.⁴ This unique prefix rule is checked automatically (cf. Sec. 4); furthermore, this rule facilitates generating fast parsers from the specification, e.g., using nested **switch** statements.

Type System

OPAL SPL does not have a hard-coded type hierarchy. Instead, each specification written in SPL contains a description of the type system used by the bytecode

⁴ Note that OPAL SPL does not have a notion of non-terminal; thus, the grammars are actually weaker than LR(0). Also, the `list` element (cf. Sec. 3.8) is allowed only *following* a unique prefix.

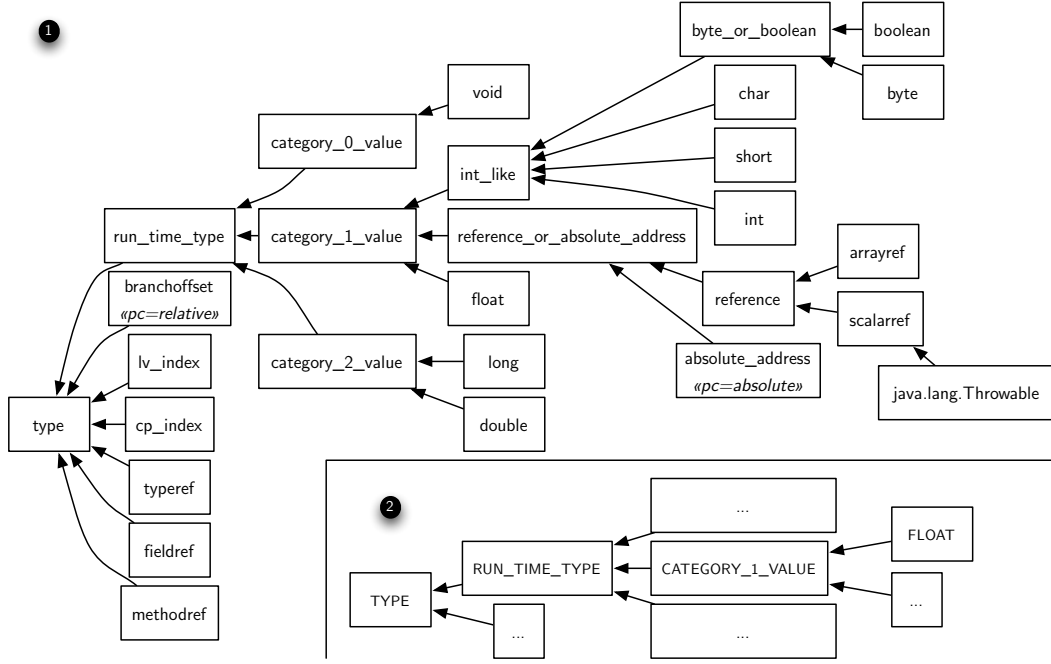


Fig. 3. OPAL SPL Type System

language being described. The only restriction is that all types have to be arranged in a single, strict hierarchy.

The Java Virtual Machine Specification [9]’s type hierarchy is shown in Fig. 3 (1). It captures all runtime types known to the Java virtual machine, as well as those types that are used only at link- or compile-time, e.g., `branchoffset`, `fieldref` and `methodref`. The hierarchy is a result of the peculiarities of the JVM’s instruction set. The `byte_or_boolean` type, e.g., is required to model the **baload** and **bastore** instructions, which operate on arrays of **byte** or **boolean** alike.

OPAL SPL’s type system implicitly defines a second type hierarchy ((2) in Fig. 3). The declared hierarchy of types (1) is mirrored by a hierarchy of kinds (2); for every (lower-case) type there automatically exists an (upper-case) kind. This convention ensures their consistency and keeps the specification itself brief. The values of kind `INT_LIKE` are `int`, `short`, etc., just as the values of type `int_like` are 1, 2, etc. Kinds enable parameterizing logical instructions like **areturn** with types, thus making a concise specification of related instructions (e.g., **freturn**, **ireturn**, and **areturn**) possible (cf. Sec. 3.12).

Information Flow

In OPAL SPL, the flow of information (values, types, register IDs) is modeled by means of named variables and expressions using the variables. In general, the flow of information is subject to the constraints illustrated by Fig. 4. For example, variables defined within a specific format sequence can only be referred to by later elements within the same format sequence; a variable cannot be referred to across format sequences. If the same variable is bound by all format sequences, i.e., it is common to all format sequences, then the variable can be used to identify register IDs, the values pushed onto the stack, etc. Similarly, if an instruction defines multiple stack

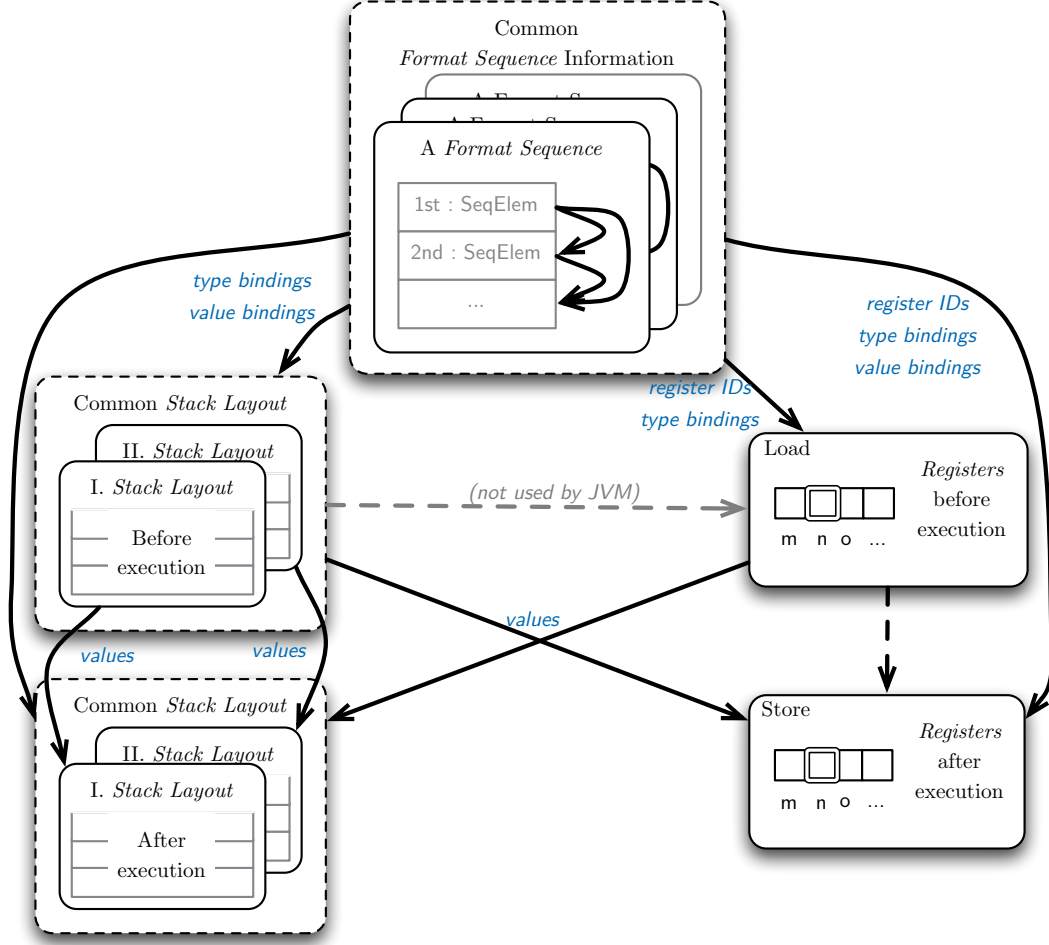


Fig. 4. Flow of information when parsing an instruction

layouts, then a value can only flow from the i -th stack layout before execution to the i -th stack layout after execution and only information that is common to all stack layouts before execution may be stored in a register.

3 Design Discussion

The design of the OPAL specification language (OPAL SPL) is influenced by the peculiarities of the JVM's instruction set [9, Chapter 6]. In the following, we discuss those instructions that had a major influence on the design.

3.1 Modeling the Stack Bottom (**athrow**)

All JVM instructions—with the exception of **athrow**—specify only the number and types of operands popped from and pushed onto the stack; they do not determine the layout of the complete stack. In case of the **athrow** instruction, however, the stack layout after its execution is completely determined (Fig. 5, line 6); the single element on the stack is the thrown exception. This necessitates explicit modeling of the stack's contents beyond the operands that are pushed and popped by a

particular instruction. The explicit modeling of the rest of the stack (line 5) hereby allows for the (implicit) modeling of stacks of a fixed size (line 6).

```

1 <instruction mnemonic="athrow" transfers_control="always">
2   ...
3   <stack> <form>
4     <before> <operand type="java.lang.Throwable" var="exception" />
5     <rest /> </before>
6     <after> <operand type="java.lang.Throwable">exception</operand> </after>
7   </form> </stack>
8   ...
9 </instruction>

```

Fig. 5. OPAL SPL specification of the **athrow** instruction

3.2 Pure Register Instructions (**iinc**)

The flow of information for instructions that do not affect the stack—e.g., the JVM’s **iinc** instruction—is depicted in Fig. 7 and adheres to the general scheme of information flow (cf. Fig. 4). After parsing the instruction according to the format sequence (Fig. 6, lines 3–5, the two variables `lvIndex` and `increment` are initialized.⁵ The value of the former variable is then used to identify the register whose value is to be incremented. The register’s value is thus bound to the variable `value`, which is incremented and stored back into the same register.

```

1 <instruction mnemonic="iinc">
2   <format>
3     <sequence> <u1 var="opcode">132</u1>
4               <u1 type="lv_index" var="lvIndex"/>
5               <i1 type="byte" var="increment"/> </sequence>
6     ...
7   </format>
8   <stack> <form>
9     <before> <rest/> </before>
10    <after> <rest/> </after>
11  </form> </stack>
12  <registers>
13    <load type="int" var="value" index="lvIndex"/>
14    <store type="int" index="lvIndex">add(value, increment)</store>
15  </registers>
16 </instruction>

```

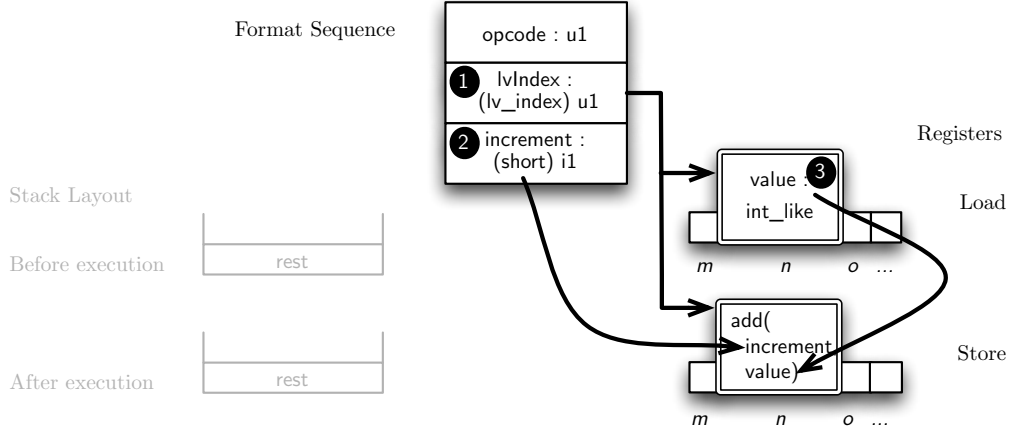
Fig. 6. OPAL SPL specification of the **iinc** instruction

This encoding illustrates OPAL SPL’s capability to model instruction sets of register-based VMs; their instructions simply do not affect the stack (lines 9–10), but only the registers (lines 13–14).

3.3 Interpretation of Arithmetic Instructions (**iinc**, **add**, **sub**, etc.)

The specification of **iinc** (Fig. 6) also illustrates OPAL SPL’s ability to model computed values, e.g., `add(value, increment)`. This information can subsequently be used, e.g., by static analyses to determine data dependencies or to perform abstract interpretations.

⁵ Note that **iinc** also supports a second, **wide** format sequence which binds the same two values.

Fig. 7. Flow of information of the **iinc** instruction

3.4 Constant Pool Handling (**ldc**)

The Java class file format achieves its compactness in part through the use of a constant pool. Hereby, immediate operands of an instruction are replaced by an index into the (global) pool. For example, in case of the load constant instruction **ldc**, the operand needs to be programmatically retrieved from the constant pool (Fig. 8, line 5). To obtain the value's type, one uses the reflective `type_of` function that the enclosing toolkitx has to provide (line 14).⁶

```

1 | <instruction mnemonic="push">
2 |   <format>
3 |     <sequence> <u1 var="opcode">18</u1> <!-- ldc -->
4 |               <u1 type="cp_index" var="cp_index" />
5 |               <implicit var="value">constant_pool.value(cp_index)</implicit> </sequence>
6 |     <sequence> <u1 var="opcode">3</u1> <!-- iconst_0 -->
7 |               <implicit var="value">0</implicit> </sequence>
8 |     <sequence> <u1 var="opcode">17</u1> <!-- sipush -->
9 |               <i2 type="short" var="value" /> </sequence>
10 |   ...
11 | </format>
12 | <stack> <form>
13 |   <before> <rest/> </before>
14 |   <after> <operand type="type_of(value)">value</operand>
15 |   <rest/> </after>
16 | </form> </stack>
17 | </instruction>

```

Fig. 8. OPAL SPL specification of the **ldc**, **iconst_0**, and **sipush** instructions

3.5 Multiple Format Sequences, Single Logical Instruction

An instruction such as **ldc**, which may refer to an integer value in the constant pool, is conceptually similar to instructions such as **iconst_0** or **sipush**; all of them push a constant value onto the operand stack. The primary difference between the format sequences of **ldc** (Fig. 8, lines 3–5) and **iconst_0** (lines 6–7) is that the former's operand resides in the constant pool. In contrast, **sipush** encodes its operand explicitly in the bytecode stream as an immediate value (line 9).

⁶ In this case `type_of` could be supplanted by `implicit_type` (cf. Sec. 3.12) in conjunction with the `constant_pool.type` function. However, `type_of` allows for a clearer specification.

To facilitate standard control- and data-flow analyses, OPAL SPL abstracts away from such details, so that similar instructions can be subsumed by more generic instructions using explicit or implicit type and value bindings. A generic **push** instruction (Fig. 8), e.g., subsumes all JVM instructions that just push a constant value onto the stack. In this case the pushed value is either a computed value (line 5), an implicit value (line 7), or an immediate operand (line 9).

3.6 Variable Operand Counts (**invokevirtual**, **invokespecial**, etc.)

Some instructions pop a variable number of operands, e.g., the four invoke instructions **invokevirtual**, **invokespecial**, **invokeinterface**, and **invokestatic**. In their case the number of popped operands directly depends on the number of arguments of the method. To support instructions that pop a variable number of operands, OPAL SPL provides the `list` element (Fig. 9, line 8). Using the `list` element’s `count` attribute, it is possible to specify a function that determines the number of operands actually popped from the stack. It is furthermore possible, by using the `loop_var` attribute, to specify a variable iterating over these operands. The loop variable (`i`) can then be used inside the `list` element to specify the expected operands (line 10). This enables specification of both the expected number and type of operands, i.e., of the method arguments (lines 8–10).

```

1 <instruction mnemonic="invokevirtual">
2   <format>
3     <sequence> <u1 var="opcode">182</u1>
4               <u2 type="cp_index" var="cplIndex"/>
5               <implicit var="methodRef">constant_pool_methodref(cplIndex)</implicit> </sequence>
6   </format>
7   <stack> <form>
8     <before> <list loop_var="i" count="methodref_arg_count(methodref)">
9               <operand type="methodref_arg_type(i,methodref)"/>
10              </list>
11              <operand type="methodref_receiver_type(methodref)"/>
12              <rest/> </before>
13     <after> <operand type="methodref_return_type(methodref)"/>
14             <rest/> </after>
15   </form> </stack>
16   <exceptions> <exception type="java.lang.NullPointerException"/> ... </exceptions>
17 </instruction>

```

Fig. 9. OPAL SPL specification of the **invokevirtual** instruction

Using functions (`methodref_arg_count`, `methodref_arg_type`, ...) offloads the intricate handling of the constant pool to externally supplied code (cf. Sec. 3.4)—the enclosing toolkit; the OPAL specification language itself remains independent of how the framework or toolkit under development stores such information.

3.7 Exceptions

The specification of **invokevirtual** (Fig. 9) also makes explicit which exceptions the instruction may throw (line 16). This information is required by control-flow analyses and thus needs to be present in specifications. To identify the instructions which may handle the exception the function (`caught_by`) needs to be defined by the toolkit. This functions computes, given both the instruction’s address and the type of the exception, the addresses of all instructions in the same method that

handle the exception. Similar to the handling of the constant pool, OPAL SPL thus offloads the handling of the exceptions attribute.

3.8 Variable-length Instructions (**tableswitch**, **lookupswitch**)

The support for variable-length instructions (**tableswitch**, **lookupswitch**) is similar to the support for instructions with a variable stack size (cf. Sec. 3.6). In this case, an `elements` element can be used to specify how many times (Fig. 10, line 7) which kind of values (lines 8–9) need to be read. Hereby, the `elements` construct can accommodate multiple sequence elements (lines 7–10).

```

1 <instruction mnemonic="lookupswitch" transfers_control="always">
2   <format>
3     <sequence> <u1 var="opcode">171</u1>
4               <padding_bytes alignment="4"/>
5               <i4 type="branchoffset" var="defaultOffset"/>
6               <i4 type="int" var="npairsCount"/>
7               <elements count="npairsCount">
8                 <i4 type="int" var="matchValue"/>
9                 <i4 type="branchoffset" var="branchoffset"/>
10              </elements> </sequence>
11   </format>
12   ...
13 </instruction>

```

Fig. 10. OPAL SPL specification of the **lookupswitch** instruction

The variable number of cases is, however, just one reason why **tableswitch** and **lookupswitch** are classified as variable-length instructions; the JVM Specification mandates that up to three padding bytes are inserted, to align the following format elements on a four-byte boundary (line 4).

3.9 Single Instruction, Multiple Operand Stacks (**dup2**)

The JVM specification defines several instructions that operate on the stack independent of their operands' types or—if we change the perspective—that behave differently depending on the type of the operands present on the stack prior to their execution. For example, the **dup2** instruction (Fig. 11) duplicates the contents of two one-word stack slots.

Instructions such as **dup2** and **dup2_x1** distinguish their operands by their computational type (category 1 or 2) rather than by their actual type (`int`, `reference`, etc.). This makes it possible to compactly encode instructions such as **dup2** and motivates the corresponding level in the type hierarchy (cf. Sec. 2.2). Additionally, this requires that OPAL SPL supports multiple stack layouts.

In OPAL SPL, the stack is modeled as a list of operands, not as a list of slots as discussed in the JVM specification. While the effect of an instruction such as **dup2** is more easily expressed in terms of stack slots, the vast majority of instructions naturally refers to operands. In particular, the decision to base the stack model on operands rather than slots avoids explicit modeling of the higher and lower halves of category-2-values, e.g., the high and low word of a 64 bit **long** operand.

```

1 <instruction mnemonic="dup2">
2   ...
3   <stack>
4     <form>
5       <before> <operand type="category_2.value" var="value" />
6               <rest /> </before>
7       <after>  <operand type="category_2.value">value</operand>
8               <operand type="category_2.value">value</operand>
9               <rest /> </after>
10    </form>
11    <form>
12      <before> <operand type="category_1.value" var="value1" />
13              <operand type="category_1.value" var="value2" />
14              <rest /> </before>
15      <after>  <operand type="category_1.value">value1</operand>
16              <operand type="category_1.value">value2</operand>
17              <operand type="category_1.value">value1</operand>
18              <operand type="category_1.value">value2</operand>
19              <rest /> </after>
20    </form>
21  </stack>
22 </instruction>

```

Fig. 11. OPAL SPL specification of the **dup2** instruction

3.10 (Conditional) Control Transfer Instructions (**if**, **goto**, **jsr**, **ret**)

To perform control-flow analyses it is necessary to identify those instructions that may transfer control, either by directly manipulating the program counter or terminating the current method. This information is specified using the `instruction` element's optional `transfers_control` attribute (Fig. 12, line 1). It specifies if control is transferred conditionally or always. The target instruction to which control is transferred is identified by the values of type `branchoffset` or `absolute_address`. For these two types the type system contains the meta-information (cf. Fig. 3) that the values have to be interpreted either as relative or absolute program counters.

```

1 <instruction mnemonic="ifgt" transfers_control="conditionally">
2   <format>
3     <sequence> <u1 var="opcode">157</u1>
4               <u2 type="branchoffset" var="branchoffset" /> </sequence>
5   </format>
6   ...
7 </instruction>

```

Fig. 12. Specification of the **ifgt** instruction

3.11 Multibyte Opcodes and Modifiers (**wide** instructions, **newarray**)

The JVM instruction set consists mostly of instructions whose opcode is a single byte, although a few instructions have longer opcode sequences. In most cases this is due to the **wide** modifier, a single byte prefix to the instruction. In case of the **newarray** instruction, however, a suffix is used to determine its precise effect. As can be seen in Fig. 13, the parser needs to examine two bytes to determine the correct format sequence.

3.12 Implicit Types and Type Constructors

The specification of **newarray** (Fig. 13) also illustrates the specification of implied types and type constructors. As the JVM instruction set is a typed assembly lan-

```

1 | <instruction mnemonic="newarray">
2 |   <format>
3 |     <sequence> <u1 var="opcode">188</u1>
4 |               <u1 var="atype">4</u1>
5 |               <implicit_type var="T">boolean</implicit_type> </sequence>
6 |     <sequence> <u1 var="opcode">188</u1>
7 |               <u1 var="atype">5</u1>
8 |               <implicit_type var="T">char</implicit_type> </sequence>
9 |     ...
10 |   </format>
11 |   <stack> <form>
12 |     <before> <operand type="int"/>
13 |             <rest/> </before>
14 |     <after> <operand type="array(1, T)"/>
15 |            <rest/> </after>
16 |   </form> </stack>
17 |   ...
18 | </instruction>

```

Fig. 13. OPAL SPL specification of the **newarray** instruction

guage, many instructions exist in a variety of formats, e.g., as **iadd**, **ladd**, **fadd**, and **dadd**. The `implicit_type` construct is designed to eliminate this kind of redundancy in the specification, resulting in a single, logical instruction: **add**. Similarly, **newarray** makes use of type bindings (lines 5, 8).

But, to precisely model the effect of **newarray** on the operand stack, an additional function that constructs a type is needed. Given a type and an integer, the function `array` constructs a new type; here, a one-dimensional array of the base type (line 14).

3.13 Extension Mechanism

OPAL SPL has been designed with extensibility in mind. The extension point for additional information is the `instruction` element's `appinfo` child, whose content can consist of arbitrary elements with a namespace other than OPAL SPL's own.

To illustrate the mechanism, suppose that we want to create a Prolog representation for Java Bytecode, in which information about operators is explicit, i.e., in which the `ifgt` instruction is an if instruction which compares two values using the greater than operator, as illustrated by Fig. 14.

```
1 | instr (METHODID, PROGRAM_COUNTER, if(gt, Branchoffset)).
```

Fig. 14. Prolog representation of an if instruction

To support this feature, we designed a small XML language to encode information about operators. The additional information is specified using child elements of the `appinfo` element as exemplified in Fig. 15, lines 2–4.

```

1 | <instruction mnemonic="ifgt" transfers_control="conditionally">
2 |   <appinfo> <cg:parameterized base="if">
3 |             <cg:operator name="gt"/>
4 |           </cg:parameterized> </appinfo>
5 |   ...
6 | </instruction>

```

Fig. 15. Application specific information.

4 Validating Specifications

To validate an OPAL SPL specification, we have defined an XML Schema which ensures syntactic correctness of the specification and performs basic identity checking. It checks, for example, that each declared type and each instruction’s mnemonic is unique. Additionally, we have developed a program which analyzes a specification and detects the following errors: (a) a format sequence does not have a unique prefix path, (b) multiple format sequences of a single instruction do not agree in the variables bound by them, (c) the number or type of function’s arguments is wrong or its result is of the wrong type.

In addition to these errors, we warn about the following potential issues: (a) a declared type, function or exception is not used, (b) a format sequence defines no variable with the `name` opcode, (c) the same opcode value is used in sequences that belong to different instruction definitions⁷, (d) an instruction mnemonic that contains “if”, “goto”, “ret”, “jsr”, “jump”, “throw”, or “switch” does not set the `transfers_control` attribute, (e) an instruction specifies more than one format sequence and more than one stack form. These additional checks have proven to be useful to detect and fix (subtle) errors early on.

5 Evaluation

Correctness of the Specification

We have used the specification of the JVM’s instruction set [9] for the implementation of a highly flexible bytecode toolkit. The toolkit supports four representations of Java bytecode: a native representation, which is a one-to-one representation of the Java Bytecode; a higher-level representation, which abstracts away some details of Java bytecode—in particular from the constant pool; an XML representation which uses the higher-level representation; a Prolog-based representation of Java Bytecode, which is also based on the higher-level representation.

We have extensively tested the developed framework and were able to import all class files part of JDK 6, Tomcat 6.0.18, and Eclipse 3.5. Additionally, we have compiled Apache Ant 1.7.1 with different compilers (`javac` and Eclipse’s built-in compiler) and different compiler settings and were also able to decode these class files. Hence, we are confident that the encoding of the JVM specification is correct.

Usefulness of the Approach

Based on the specification, we have developed two generators which are both implemented using XSLT. The first XSLT transformation generates the classes to represent all instructions and is 350 lines long. Each generated class represents an instruction as a Java object and offers the functionality to get an XML and a Prolog representation of the concrete instance of an instruction. The second XSLT transformation generates the parser for a code array which creates the instance of the instruction classes on the fly. This transformation is another 300 lines long. We compared this with the Bytecode Code Engineering Library (BCEL) [2] which uses

⁷ The decision to enable multiple sequences that contain the same opcode value was necessary to model the `newarray` instruction.

a similar approach for representing and handling instructions. When compared to the instruction-related code of BCEL, the generator is between 15 and 20 times smaller.

Another advantage of the approach is that changes that affect all instructions are localized. For example, in case of the Prolog representation we tested several different representations which often affected all instructions. Nevertheless, in general less than 40 lines of code of the generator needed to be changed.

6 Related Work

Applying XML technologies to Java bytecode is not a new idea [5]. The XML serialization of class files, e.g., allows for their declarative transformation using XSLT. The XMLVM [11] project aims to support not only the JVM instruction set [9], but also the CLR instruction set [8]. This requires that at least the CLR’s operand stack is transformed [12], as the JVM requires. The description of the effect that individual CLR instructions have on the operand stack is, however, not specified in an easily accessible format like OPAL SPL, but rather embedded within the XSL transformations.

The rules of Hoare-style program logic can also serve as a specification of the JVM instruction set [1]. While such a specification goes beyond OPAL SPL as far the instructions’ effect on the VM’s state (operand stack, registers, etc.) is concerned, it does not describe instruction formats and also groups instructions (cf. Sec. 3.4) only implicitly, through derivation rules, if at all.

The Project Maxwell assembler system [10] is able to describe instruction formats that are more complex than those commonly encountered in high-level intermediate languages, namely those of the IA32, PowerPC, and SPARC instruction set architectures. These descriptions are then used to generate assemblers and disassemblers as well as test cases for either. The system is unable, however, to describe the instructions’ effect; only their format is described. Unlike OPAL SPL, these descriptions are not made available in a language-independent format like XML, but rather constructed programmatically, using a domain-specific language embedded into Java.

Vmgen [6] is a generator for efficient interpreters for stack-based intermediate languages. While it can also be used to generate code for register-based intermediate languages, it cannot describe such instructions declaratively, as can be done using the `load` and `store` elements in OPAL SPL. Its descriptions also do not cover the format of the bytecode itself; thus, it is not possible to generate a parser from vmgen’s descriptions. One notable feature of vmgen is its (almost) uniform treatment of operand stack and instruction stream, which simplifies the description of instructions with immediate operands. OPAL SPL does not achieve the same degree of uniformity because it describes how instructions are stored in class files.

7 Conclusion and Future Work

In this paper, we have first discussed a language for the specification of both the format and the execution semantics of bytecode based instructions with respect

to memory access. The language was used to encode the semantics of the JVM's instruction set. The resulting encoding of the JVM Specification was subsequently used for the development of a Java Bytecode framework that reads in class files and performs standard control- and data-flow analyses; e.g., to transform the stack-based bytecode representation into an SSA representation. Our evaluation shows that using the specification as the foundation for the development of bytecode toolkits significantly reduces the number of lines of code that need to be developed and also reduces the development time of such toolkits.

In future work, we will investigate the use of OPAL SPL for the encoding of other bytecode languages, such as the Common Intermediate Language. This would make it possible to develop (control- and dataflow-) analyses with respect to the OPAL SPL and to use the same analysis to analyze bytecode of different languages.

Acknowledgments

The authors would like to thank Lucas Satabin for implementing the type checker. This work was supported by www.cased.de.

References

- [1] Fabian Bannwart and Peter Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):255–273, 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005).
- [2] The Bytecode Engineering Library (BCEL). <http://jakarta.apache.org/bcel/manual.html>, 2006.
- [3] Eric Bruneton. ASM 3.0: A Java bytecode engineering library. <http://download.forge.objectweb.org/asm/asm-guide.pdf>, February 2007.
- [4] Shigeru Chiba. Javassist - Java Programming Assistant 3.11.0.ga. <http://www.csg.is.titech.ac.jp/~chiba/javassist/>, 2009.
- [5] Michael Eichberg. BAT2XML: XML-based java bytecode representation. *Electronic Notes in Theoretical Computer Science*, 141(1):93–107, 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005).
- [6] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Software Practice & Experience*, 32(3):265–294, 2002.
- [7] IBM. The t. j. watson libraries for analysis. <http://wala.sourceforge.net/>, 2006.
- [8] ISO/IEC, Geneva, Switzerland. *Information technology – Common Language Infrastructure (CLI) Partitions I to VI*, ISO/IEC 23271:2006(E) edition, 2006.
- [9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- [10] Bernd Mathiske, Doug Simon, and Dave Ungar. The Project Maxwell assembler system. In *PPPJ '06: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, pages 3–12, New York, NY, USA, 2006. ACM.
- [11] Arno Puder. Byte code transformations using XSL stylesheets. In *SNPD '08: Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 563–568, Washington, DC, USA, 2008. IEEE Computer Society.
- [12] Arno Puder and Jessica Lee. Towards an XML-based bytecode level transformation framework. *Electronic Notes in Theoretical Computer Science*, 253(5):97–111, 2009. Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009).
- [13] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, volume 1781, 2000.

JRebel Tool Demo

Jevgeni Kabanov¹

*Dept. of Computer Science
University of Tartu
Tartu, Estonia*

Abstract

JRebel started as an academical project that became a successful commercial product used by thousands of developers worldwide. It extends the Java Virtual Machine with a mechanism that allows seamless class reloading. It uses bytecode manipulation extensively, both for the just-in-time class translator and numerous integrations with the Java SE and EE APIs. In this live demo we will show how it can be used in real-life projects to cut development time by 8 to 18 per cent.

Keywords: bytecode, JRebel, ClassLoader, API, retroactive

1 Introduction

Java EE development day-to-day activities involves deploying the application to the Java EE server. This step is necessary after the application has been compiled and packaged into an archive as per Java EE specification. Every time developers want to make changes to the running application they need to deploy it, which can take from a few seconds in the best case to several minutes in the worst.

An alternative way to update an application is using the HotSwap protocol [1], available from the Java EE debugger. This allows to update the application classes without redeploying it. Unfortunately only a very restricted set of changes is allowed; namely HotSwap allows changes to the method bodies, but does not allow changing the class signature or inheritance hierarchy. Thus no new methods, fields or constructors are allowed.

At the end of 2006 we came up with an idea for extending the Java virtual machine with a mechanism that would allow to change the class bytecode beyond the limits of the HotSwap protocol [2]. During 2007 we developed and released a prototype initially code-named “Badger” and for the public release renamed to “JavaRebel”. In 2009 we released the version 2.0, which supported a layer of indirection on top of the ClassLoader API that allowed the users to edit classes and

¹ Email: ekabanov@gmail.com

resources in their workspace instead of packaging them into .WAR or .EAR archives as per Java EE specification. We also introduced an extension API that allowed to make use of JavaRebel features in third-party applications as well as build plugins for JavaRebel to support changes in framework configuration. We also renamed the tool once more to “JRebel” due to trademark issues.

When we started working on the tool most of the research was focused on using ClassLoaders to dynamically update code [3] or on modifying JVMs to do so [4]. Recently there has been more investigation into similar systems [5,6], but no industry tools are available to compete with JRebel.

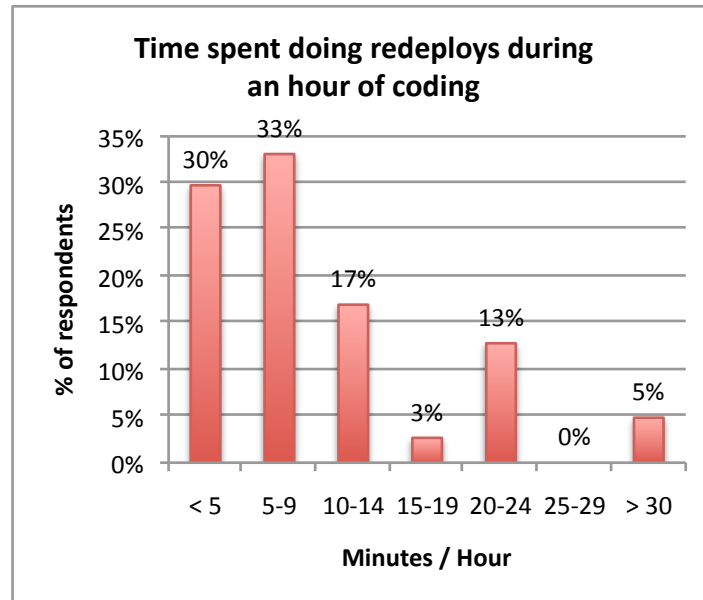
At the moment we estimate over ten thousand of JRebel users around the world. This number includes commercial users as well as various otherwise licensed users, e.g. Open Source developers and Scala developers who can request a free license.

The rest of the paper is organized as follows. In Section 2 we review the problem and its scope, Section 3 gives a brief overview of the tool, Section 4 describes the technical background and Section 5 covers the supporting artifacts and third-party extensions.

This work was partially supported by the OÜ Tarkvara Tehnoloogia Arenduskeskus, Enterprise Estonia and Estonian Science Foundation grant No. 8421.

2 Problem Scope

In 2009 we conducted a survey reaching over 1000 Java developers to estimate the amount of time spent in the redeploy phase of development. The survey asked how long a server redeploy takes and how many times an hour they are performed and is summarized by the following chart:



The average is about *10.5 minutes per hour*, accounting for 17.5% of total development time. The standard deviation is 8, which means that the actual per cent varies quite a lot. The estimated number of Java developers worldwide is *nine*

million. Estimating conservatively that only half of them develop for Java EE we have an estimate for the annual worldwide cost to the economy of *\$56,700,000,000* a year (assuming 48 working weeks a year, 5 hours of coding per day and a \$30 per hour salary).

3 Tool Overview

JRebel installs a `-javaagent` JVM plugin that monitors the classes and resources in the workspace and propagates their changes to the running application. The following types of changes are supported:

- Changes to Java classes beyond what is supported by HotSwap.
- Changes to framework configuration (e.g. Spring XML files and annotations, Struts mappings, etc).
- Any changes to static resources (e.g. JSPs, HTMLs, CSSs, XMLs, .properties, etc)

JRebel works by rewriting the bytecode of Java classes to enable versioning. To do that we use just-in-time bytecode translation in a manner akin to dynamic languages compilation and runtime support. This enables us to support changes to Java class schema, though not to the type hierarchy.

Type of change	HotSwap	JRebel
Changes to method bodies	Yes	Yes
Adding/removing methods	No	Yes
Adding/removing constructors	No	Yes
Adding/removing fields	No	Yes
Replacing superclass	No	No
Adding/removing implemented interfaces	No	No

Since version 2.0 we extend the ClassLoader API to allow injecting classes and resources from locations outside default classpath. We use this functionality to allow our users to specify the layout of their projects on the filesystem using a `rebel.xml` configuration file and make application servers read the classes and resources directly from those projects, instead of the .WAR or .EAR archives as prescribed by the Java EE specification. As most of the build phase time is spent packaging those classes and resources into the archive, it allows us to save most of the time spent in that phase.

To make the tool more convenient to use we provide IDE plugins for Eclipse, IntelliJ IDEA and NetBeans. These plugins improve debugging with JRebel by hiding the synthetic fields and methods introduced by the translation process. We also provide a plugin for the Maven build system, that automatically generates the `rebel.xml` configuration file necessary to make use of the project direct mapping functionality.

4 Technical Background

To explain how JRebel works we need to start with the reasons why support for full schema change was not implemented in Java HotSwap. This section draws mainly on [7,8] and some private conversations with Thomas Wuerthinger.

When loaded into the JVM, an object is represented by a structure in memory, occupying a continuous region of memory with a specific size (its fields plus metadata). In order to add a field, we would need to resize that structure, but since nearby regions may already be occupied, we would need to relocate the whole structure to a different region where there is enough free space to fit it in. Now, since we're actually updating a class (and not just a single object) we would have to do this to every object of that class.

Fortunately object relocation is something that Java does all the time. Java garbage collectors relocate objects every time they compact the heap. However the problem is that the abstraction of one heap is just that, an abstraction. The actual layout of memory depends on the garbage collector that is currently active and, to be compatible with all of them, the relocation would have to be implemented in the active garbage collector. This, however, presents quite a challenge, as the Sun JVM features at least four garbage collectors (some of them multi-threaded), two JIT compilers and a multitude of hardware platforms and operating systems it supports. Implementing this functionality in each of the garbage collectors and ensuring compatibility with the other components of the environment is a challenging enough task that since the 2001 when the initial HotSwap implementation the full schema update has yet to make it into the Sun JVM.

It would seem that adding methods to classes would be easier, but due to optimizations in the class layout (specifically inlined v-tables), it assumes resizing and relocating the class structure, returning to the same issue.

How does JRebel solve this problem?—JRebel works on a different level of abstraction than HotSwap. Whereas HotSwap works at the virtual machine level and is dependent on the inner workings of the JVM, JRebel makes use of two remarkable features of the JVM—abstract bytecode and classloaders. Classloaders allow JRebel to recognize the moment when a class is loaded, then translate the bytecode on-the-fly to create another layer of abstraction between the virtual machine and the executed code.

The problem in reloading classes is that once a class has been loaded it cannot be unloaded or changed; but we are free to load new classes as needed. To understand how we could theoretically reload classes, let's take a look at the implementation of JRuby [9].

Ruby is required to support any runtime changes to the object, including adding fields and methods (albeit named differently). JRuby implements those features on the JVM, by treating objects as not much more than a runtime map from method names to their implementations and from field names to their values. The implementations for those methods are contained in anonymously named classes that are generated when the method is encountered. When a method is added, JRuby generates a new anonymous class that includes the body of that method. As each anonymous class has a unique name there are no issues loading them and as a

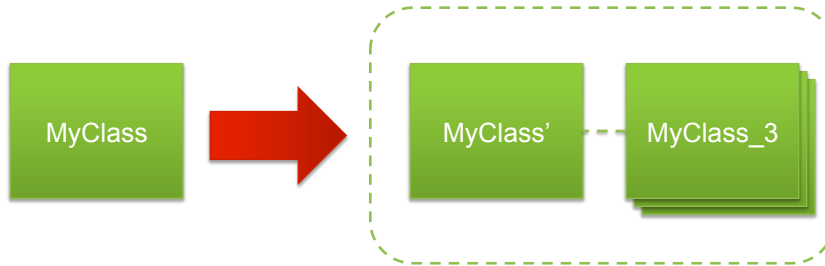
result the application is updated on-the-fly.

We could then use the same transformation as JRuby and split all Java classes into a holder class and method body classes. Unfortunately, such an approach would be subject to (at least) the following problems:

Performance Such a setup would mean that each method invocation would be subject to indirection. We could optimize, but the application would be at least an order of magnitude slower. Memory use would also skyrocket, as so many classes are created.

Compatibility Although Java is a static language it includes some dynamic features like reflection and dynamic proxies. If we apply the “JRuby” transformation none of those features will work unless we replace the Reflection API with our own classes, aware of the transformation.

Therefore, JRebel does not take such an approach. Instead we transform the class into a frontend class with a signature compatible to the original and an anonymous backend class, a new version of which can be loaded when the original class is updated. We rewrite all invocations among transformed classes introducing a level of indirection where necessary. However we use advanced Just-In-Time compilation techniques to inline as many indirections as we can, so as to keep performance overhead to a minimum.



To demonstrate the extent of our optimization we chose the Chameneos [10] benchmark that is highly concurrent and is implemented in multiple classes thus needing a lot of indirection in a naive implementation. Running this benchmark with JRebel agent enabled we can see that there is no discernible difference from running it in vanilla configuration. Even if we update all of the classes in the benchmark the overhead is still under 60%.

Time	Vanilla	JRebel	JRebel Updated
real	0m38.673s	0m37.457s	0m58.130s
user	1m10.747s	1m7.946s	1m38.661s
sys	0m0.821s	0m0.832s	0m1.317s

Sixty per cent may sound like a lot, but this implies that every single class in the application is updated, which is an extraordinary case. We optimize heavily to reduce overhead for the unchanged classes, as even a 60% overhead on updated classes will translate to a negligible total overhead as only a small fraction of an application is usually updated.

5 Artifacts and Extensions

The JRebel distribution includes an installer, extensive reference manual and configuration wizard. Dozens of articles available from our website and third-party publications describe various applications of JRebel in the real world. A support forum with over 2000 posts is also available to our users.

Although JRebel is a commercial product, a significant portion of its code is available as Open Source in the Subversion repository at <http://repos.zereturnaround.com/svn/>. The parts unavailable as Open Source include the just-in-time translation engine and high-level `rebel.xml` handling.

The following Open Source artifacts are available from the Subversion repository:

Test suite To ensure the stability and compatibility of the product we have compiled a test suite that contains test cases for both JVM compatibility and application server compatibility.

Tool plugins Plugins for Eclipse, IntelliJ IDEA, NetBeans and Maven.

JRebel SDK and utils The SDK and utility classes that support writing JRebel plugins or using it in a third-party environment.

JRebel plugins Plugins for various servers (Tomcat, JBoss, Weblogic, ...) and frameworks (Spring, Struts, ...).

The JRebel SDK enables third-party contributors to submit additional plugins for JRebel. To date the plugins for Struts 2, Stripes, Wicket and Log4J have been contributed.

References

- [1] Java HotSwap, <http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html#hotswap>.
- [2] Kabanov, Jevgeni, *METHOD AND ARRANGEMENT FOR RE-LOADING A CLASS*, 2008 (US Patent Application 20080282266).
- [3] Liang, S. and Bracha, G., *Dynamic class loading in the Java virtual machine*, ACM SIGPLAN Notices **33/10**, ACM, 1998.
- [4] Malabarba, Scott and Pandey, Raju and Gragg, Jeff and Barr, Earl and Barnes, J. Fritz, *Runtime Support for Type-Safe Dynamic Java Classes*, ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming [337–361], Springer-Verlag, London, 2000.
- [5] Gregersen, A.R. and Jørgensen, B.N., *Extending eclipse RCP with dynamic update of active plug-ins*, Journal of Object Technology **6/6** [67–89], 2007.
- [6] Pukall, M. and Kästner, C. and Saake, G., *Towards unanticipated runtime adaptation of Java applications*, Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC) [85–92], 2009.
- [7] Dmitriev, Mikhail, *Towards flexible and safe technology for runtime evolution of java language applications*, OOPSLA Workshop on Engineering Complex Object-Oriented Systems for Evolution, 2001.
- [8] Thomas Wuerthinger, *Dynamic Code Evolution for the Java HotSpot(TM) Virtual Machine*, 2009.
- [9] Nutter, C.O. and Enebo, T.E., *JRuby – Java powered Ruby implementation*, 2003.
- [10] Kaiser, C. and Pradat-Peyre, JF, *Chameneos, a Concurrency Game for Java, Ada and Others.*, ACS/IEEE Int. Conf. AICCSA03.