JOT
JOURNAL OF
OBJECT TECHNOLOGY

Java Bytecode Verification with OCL Why, How and When?

Christoph Bockisch, Gabriele Taentzer, Nebras Nassar, and Lukas Wydra
Philipps-Universität Marburg, Germany

ABSTRACT Program transformations are frequently developed, e.g., to realize programming language extensions or dynamic program analyses such as profiling. They are typically implemented by manipulating bytecode as the availability of source code is not guaranteed. There are standard libraries such as ASM that are typically used for implementing Java bytecode manipulations. To check their correctness, they are usually tested by applying them to different programs, running the manipulated programs and observing their behaviors. As part of the second step, the Java virtual machine verifies the bytecode, which can uncover errors in the bytecode introduced by the manipulation. That approach uses different technologies that are not well linked making the process of developing and testing bytecode manipulations difficult.
In this paper, we intend to perform bytecode manipulation by using concepts and techniques of model-driven engineering. We are convinced that the declarative nature of model transformation rules allows the debugging and analyzing of bytecode manipulations in more details than classically done. Following this path, a meta-model for bytecode is needed including OCL constraints for bytecode verification. We analyze the semantic rules of the bytecode verifier according to their complexity factor, present a meta-model for Java bytecode, show how the semantic rules can be expressed as OCL constraints on top of this meta-model, and show that basing bytecode manipulation on model transformation can provide more immediate guidance and feedback to the developer.

KEYWORDS Program transformation, Java bytecode, meta-model, OCL, bytecode verification.

1. Introduction

In programming language research, developing language extensions or dynamic program analyses as program transformations is a popular approach. Consider, for example, an extension of the Java language with implicit invocation (Steimann et al. 2010) for direct support of the Observer pattern, and the dynamic analysis of program performance through profiling (Liang & Viswanathan 1999). An implicit invocation allows adding a specification to a method when it should be executed. A language extension will transform a program using implicit invocations such that explicit invocations to methods are inserted at appropriate places. A profiler, e.g., typically inserts a fixed sequence of code at the beginning and at the end of each relevant method in a program to record the start and end time of each method execution.

Program transformation can be applied to either the source code in a preprocessing step (i.e., before the compilation) or in a postprocessing step (i.e., after the compilation) which means that the bytecode is transformed. Bytecode is often chosen over source code because the latter may not be available for the whole program, for example, when closed-source libraries are used or when the bytecode is generated.

For the implementation, a program transformation reads the base program first and parses it into an abstract syntax tree (AST) then. Next, the AST is analyzed and rewritten. Since parsing and processing an AST are very common but non-trivial exercises, there are standard libraries that are typically used for implementing program transformations. In the Java world, Polyglot (Nystrom et al. 2003) is an example of a preprocessor

JOT reference format:
Christoph Bockisch, Gabriele Taentzer, Nebras Nassar, and Lukas Wydra. *Java Bytecode Verification with OCL:Why, How and When?*. Journal of Object Technology. Vol. 19, No. 3, 2020. http://dx.doi.org/10.5381/jot.2020.19.3.a13

that works on the AST of the source code. ASM (Bruneton et al. 2002) and BCEL (BCEL 2020) are popular examples for the postprocessing approach; they work on the AST of the virtual machine code. In this context, program transformations are often called *bytecode manipulation*, a term which we will use throughout this paper to avoid confusion with model transformation.

The libraries mentioned stick relatively close to the original code representation which is optimized for machine-readability. Thus, developers of program transformations have to cope with a fixed, relatively low level of abstraction which boils down to either a list of source code statements or a list of machine code instructions. *We need an approach to develop program transformations in a more flexible and easier way. The representation of code should let developers choose the right abstraction level for formulating program transformations.*

The classical way of implementing a program transformation is an imperative program that manipulates the code of the program under transformation. To check the correctness of a program transformation, it is usually tested by (1) applying it to different programs, (2) running the transformed programs, and (3) observing if they behave as intended. As part of the second step, the Java virtual machine loads the program code and performs so-called *bytecode verification*, which checks that no security-relevant restrictions are violated. For example, there is a restriction that a program must not read from an empty operand stack. Restrictions like this one are defined as part of the Java Virtual Machine (JVM) Specification (Lindholm et al. 2020, Section 4.10). In the case of illegal code, the Java virtual machine stops the execution and reports the violations.

This approach uses different technologies: a Java editor, for example, for developing the program transformation, a Java bytecode viewer for viewing the program before and after a transformation, and the console output of the Java virtual machine for observing the behavior of a transformed program. *The information provided by these technologies is not well linked which makes the process of developing and testing program transformations difficult.* While we cannot generally help to ensure correct behavior of the transformed program, in this paper, we set out to support implementers of bytecode manipulations in the first two steps.

To solve the stated problems of developing program transformations, we intend to use concepts and techniques of model-driven engineering (MDE) (Brambilla et al. 2012). Models allow choosing the right abstraction level and to connect it to the instruction level of programs. Moreover, we are convinced that the declarative nature of model transformation rules allows debugging and analyzing program transformations in more details than classically done. The declarative nature of model-transformation rules allows tracing their application. When an element in the transformed model violates a constraint, it is easy to trace back to find out which transformation rules may have caused the violation. We will further show in this paper that, due to their declarative nature, transformation rules can be checked for potential constraint violations before even applying them in certain cases.

Our approach to mapping the code transformation problem to a model transformation problem is inspired by the analogy between model-driven engineering and compiler construction. Both share several common concepts of defining languages. First, context-free grammars as used in compiler construction are comparable to meta-models as used in MDE (Alanen & Porres 2003). This relation is, for example, exploited in the tooling for (textual) domain-specific languages (DSL) such as Xtext (Efftinge & Völter 2006). Common compilers (Aho et al. 2006) do their work in several phases: they read in a program representation, perform some initial checks and potentially produce a different representation of the program. The first two phases are called *lexical and syntactic analysis* and produce an abstract syntax tree (AST) of a syntactically well-formed program. This kind of analysis is reflected in DSL tool kits as well; in addition to an AST, a domain model is created. The third compiler phase is called *semantic analysis* and performs a number of correctness checks such as type correctness, absence of stack overflow or underflow, a correct behavior of the program counter, and correct register and object initialization. Such semantic checks are hardly reflected in language implementations using DSL tool kits. A potential reason may be that DSLs are typically so simple that semantic checks are not needed.

We want to demonstrate the benefit of implementing bytecode manipulations as model transformations. Thus, our target does not need to be a simple DSL but may be a complex general-purpose language. Continuing the analogy, we want to explore how far the semantic analysis phase of complex compilers can be mapped to the checking of invariants formulated in OCL (OCL 2014).

Since Java bytecode is most popular in research on program transformations, we consider a meta-model for Java bytecode. An instance model of this meta-model may either be automatically generated from an existing bytecode file or result from a transformation. In any case, it has to meet the lexical and syntactic requirements as captured in the meta-model. But it is possible to define an illegal program violating the semantic requirements as indicated above (e.g., by reading from an empty stack).

In this paper, we (1) *analyze the semantic rules in the JVM specification* according to their complexity factor, (2) *present a meta-model for Java bytecode*, (3) *show how the semantic rules of the bytecode verifier can be expressed as OCL constraints* on top of this meta-model, and show that (4) *basing program transformation on model transformation has several advantages*: Expressing the semantic rules in a formalism like OCL allows for checking them earlier than in state-of-the-art approaches to program transformation. We show at an example that our approach can provide more immediate feedback to the developers of bytecode manipulations. The declarative definition of correctness criteria in OCL can even be used to guide the design of transformation rules in some cases and to restrict the application of transformation rules to valid cases in an automated way.

This article is structured as follows: We start with recalling some background on Java bytecode. In Section 3, we recall all concepts of model-driven engineering (MDE) that are needed

for our work. Our approach to bytecode processing based on MDE is presented in Section 4. Section 5 shows a case example that demonstrates the usefulness of our approach. We conclude with considering the related work in Section 6 and the final remarks in Section 7.

2. Background on Java Bytecode

Before discussing how to express bytecode verifier rules as OCL constraints, it is necessary to describe Java bytecode in general as well as the verification rules as defined in the Java Virtual Machine specification.

2.1. Java Bytecode Format

Knowing the Java source code language, Java bytecode looks largely familiar. The general structure of bytecode is pretty similar to that of source code. For each compiled Java class, a `.class` file is created. This file contains type information such as the super class and interfaces, the outer class (in the case of nested classes), and the declarations of fields and methods. For (non-abstract) methods, the implementation is stored as bytecode instructions.

However, there are also several significant differences. This subsection briefly summarizes the most important differences that need to be known to understand the bytecode meta-model and its invariants which are the subject of this paper. A full definition of the Java bytecode is given in the JVM Specification (Lindholm et al. 2020).

The architecture of the bytecode instruction set is a hybrid of a stack-based and a register-based machine. Arithmetic expressions are computed using a stack. For example, an expression like `a + b` is represented by three instructions: first push the value of `a` on the stack, second push the value of `b` on the stack, third pop the two top-most values from the stack, add them and push the result on the stack. For method invocations, arguments are also passed by pushing them on the stack; the result value will be on the stack when the invoked method returns. Local variables are represented by a virtually unlimited number of virtual registers that can be randomly accessed. References to entities such as types, methods or fields, are always fully qualified in Java bytecode.

Instructions do not have a nesting structure like source code statements. Instead, the implementation of a method is stored as a sequence of instructions whereby control flow is defined by either conditional or unconditional jump instructions. The target of a jump is defined by a relative offset. Exceptional control flow is defined by a so-called exception table. This table specifies the range of instructions for which an exception handler is active. The specification is given by the offsets of the first and last instruction, respectively, in the range. Each table entry denotes the type of the handled exception and the offset of the first instruction of the handler code.

The Java virtual machine supports only four kinds of primitive values: integers and floating point values, both in 32bit and 64bit. Thus, values that have one of the primitive types boolean, char, byte, short and int in the Java source language are treated equally by the JVM. For example, there is only one bytecode instruction for the addition of 32bit integers. In type declarations such as the declared result type of a method, the exact type is, however, retained in the bytecode.

As a further difference, type names are represented differently in bytecode and source code. A primitive type is represented by a single letter string, for example, "I" for int or "D" for double. A reference type is given by the concatenation of the strings "L", the class descriptor and ";". The class descriptor in turn is more or less the fully-qualified class name where `.` is replaced by `/`. In places where reference types are allowed only, the bytecode format may require a plain class descriptor that is a descriptor without "L" and ";".

2.2. Bytecode Verification

Java bytecode verification is a very important component of the Java security concept which is also called the *sandbox principle* (Li Gong 1998). The JVM can execute code from arbitrary sources including generally untrusted sources like the Internet. Since the JVM is executed locally with relatively broad permissions, malicious bytecode from an untrusted source could potentially exploit the JVM to acquire its permissions for executing an attack on the local system. To prevent such damage to the JVM's integrity, all bytecode loaded is always *verified* before execution.

Beside malicious exploits, it is also possible that, unintentionally, bytecode occurs which violates the specification. This can happen in particular when classes are compiled separately and bytecode from different sources is used inconsistently. When the final keyword is added to a super class, for example, a subclass is not automatically recompiled.

The specification of the Java virtual machine defines an extensive set of rules which must be satisfied by legal bytecode. The verifier examines the bytecode of each loaded class to determine if none of the rules is violated and thus is safe to execute. This set of rules is divided into three categories (Lindholm et al. 2020, ch. 4.9–4.20) :

1. *Static rules* ensure that the class file itself is well-formed (e.g., the target of a jump instruction must be an instruction within the same method).

2. *Structural rules* prescribe correct relationships for the data flow between instructions. (When an instruction is executed, for example, it consumes values from the operand stack that have been produced by other instructions and the consumed values must be of the expected types.)

3. *Additional rules*: In addition to static and structural rules, bytecode verification must ensure that each class has a direct super class and that the final modifier is respected.

These constraints are verified at link time when the classifier is invoked for the first time; note that they are not verified continuously at runtime. Since runtime data is not present at link time, verification is based on types instead of actual values. The bytecode verifier simulates all possible control flow paths that could occur at runtime and checks if all type conditions are fulfilled. The verification at link time saves

expensive type checking per instruction at runtime and implies that the verification of a defensive JVM (which verifies the rules at runtime) never fails. In this paper, we will call bytecode which successfully passes verification to be *well-behaved* bytecode.

In the remainder of this section, we will summarize the rules along the above categories and explain a representative subset of the rules in more detail. The goal of this section is to give an overview of the verifier rules and to illustrate different levels of complexity of verifier rules. As an indicator of complexity, we primarily consider the *scope* within which to access relevant information required to check the rule.

2.2.1. Static Rules

Many of the static rules are concerned with the well-formedness of a class file. For example, arrays that contain data must not be truncated. These rules will not be considered throughout this paper, because bytecode manipulation toolkits do not allow creating of ill-formed class files.

Local information (I). A specific kind of static rules specifies the format of symbols (encoded as UTF-8 strings in the class file) like method and class names (Lindholm et al. 2020). The legal format is specified for each kind of symbol. Thus, rules are specified such as:

Rule 1. *"Names [...] must not contain any of the ASCII characters . ; [/."*

Rule 2. *"Method names are further constrained so that, with the exception of "<init>" and "<clinit>" [N.B.: the reserved names for constructors and static initializers in class files], they must not contain the ASCII characters < >."*

Rule 3. *"Each field of a class may have at most one of its public, private, and protected flags set."*

Rule 4. *"A field descriptor representing an array type is valid only if it represents a type with 255 or fewer dimensions."*

The first two rules require operators for checking the presence of predefined characters within a string and string equality. The third rule needs to check the mutual exclusion of information and the fourth rule applies some counting.

Directly referenced information (I). A more complex example of a static rule is the following one that ensured uniqueness of field definitions within one class. This requires accessing information from directly nested entities (the scope of this rule), in this case, the fields defined within a class.

Rule 5. *"No two fields in one class file may have the same name and descriptor."*

2.2.2. Structural Rules

The structural bytecode verification rules are concerned with type checking the instructions within each method.

Directly referenced information (II). Some of the structural rules specify within which classes or methods certain instructions may be used. Thus, the scope of relevant information is the class or method in which the instruction is contained.

Rule 6. *"An ireturn instruction is type safe if the enclosing method has a declared return type of int."*

Traversing information referenced by name (I). Another kind of rule specifies that visible methods and fields may be accessed only. As an example, consider the following rule for accessing a protected field.

Rule 7. *If invokevirtual or invokespecial is used to access a protected method declared in a superclass that is a member of a different run-time package than the current class, then the type of the class instance being accessed (that is, the type of the target reference on the operand stack) must be assignment compatible with the current class.*

This rule requires access to the computed stack. In addition, it requires access to the type hierarchy defined by the class files of the program. This access is required for two purposes. First, the declaration of the invoked method must be looked-up because it contains the visibility definition (i.e., if it is protected). Second, the type of the receiver object, found on the computed stack, must be checked for assignment compatibility with the current class. This means that the class of the receiver object must be either the current class or a subclass thereof.

Thus, the scope within which information must be accessed to check this rule is not local. Instead it is defined elsewhere and referenced by name. In general, it is even necessary to follow a chain of such named references because the searched method may also be defined in a superclass of the referenced class. Therefore, also this rule involves a topological search operation.

Traversing local information. When a Java bytecode program is executed, the methods defined in the bytecode are called and their instructions are executed. For each method call, also called *method activation*, local storage is allocated at runtime to hold the method's state. It may consist of local variable values on the operand stack which are used to pass values between instructions. This storage and the values are available at runtime only. The layout of the operand stack and the local variables at each instruction, however, can be statically computed. The layout is determined by the amount and types of the values that will be on the stack or held in local variables when the execution reaches the instruction.

The verifier can establish—without running the program—whether the expectations of each instruction will be met in every program run by traversing the possible control-flow paths. For each instruction, the successor in a control-flow path is either the neighbor in the method's sequence of instructions, or it is directly referenced by the instruction in the case of branching instructions. Rule 8 shows the requirements on the stack layout for the integer addition instruction. Analog rules exist for the other instructions.

Rule 8. *The integer addition instruction (iadd) requires two int-like values at the top of the stack.*

2.2.3. Additional Rules

There are only three rules in this category that are concerned with the proper use of extending a class and respecting the keyword final.

Local information (II). The first rule in this category has very low complexity. It simply checks if the current class specifies the name of a superclass.

Rule 9. *"Every class (except Object) has a direct superclass."*

Traversing information referenced by name (II). The other two rules in this category have a complexity that is similar to that of Rule 7 in the previous subsection. The rule searches the class hierarchy for a method definition and checks its access modifiers. The last rule is similar but traverses only one step in the class hierarchy.

Rule 10. *"Final methods are not overridden."*

Rule 11. *"Final classes are not subclassed."*

3. Model-Driven Engineering

This section recalls all those concepts from model-driven engineering that are needed in this paper. The bytecode meta-model will be based on the Eclipse Modeling Framework and the Object Constraint Language which are recalled next. Thereafter we recall the model transformation approach Henshin. As we want to use it for specifying bytecode transformations such that the bytecode remains valid, we also recall how model transformations can be designed to be constraint-preserving.

3.1. The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) (Steinberg et al. 2008) has evolved into a de-facto standard technology for defining models and modeling languages. In EMF, meta-models are defined using Ecore, an implementation of the OMG's EMOF standard (OMG 2016). Meta-models in Ecore prescribe the structures that instance models of the modeled domain should exhibit. Concepts known from UML class diagrams are used, namely the classification of objects and their attributes, references to objects, and constraints on object structures. A specific kind of references is *containments*. There may be a root object that contains all other objects of a model directly or transitively.

Each model that conforms to its meta-model and fulfills the following constraints is called *EMF model*: (1) Each object must not have more than one container. (2) Cycles of containments must not occur. (3) There are no two references of the same type from the same source to the same target object. (4) If reference types $t1$ and $t2$ are opposite to each other: For each reference of type $t1$, there has to be a reference of type $t2$ linking the same objects in the opposite direction. In addition, a model may be constrained by *multiplicities* and OCL constraints which are recalled below.

3.2. The Object Constraint Language

The Object Constraint Language (OCL) (OCL 2014) is a formal language that can be used to specify the bytecode verifier rules for models that are defined over a meta-model. They are formulated as invariants. In the following, we give a rough classification for the complexity of invariants. The rationale for this classification is partly given by the efficiency of checking OCL constraints. The authors of (Franconi et al. 2019) argue for a subset of OCL invariants that are first-order and expressive enough for practical use.

OCL has a type system which consists of mainly three categories: custom types, predefined types and template types. Custom types are either class types or enumeration types defined by the user in the corresponding meta-model. Predefined types are Integer, Real, String, and Boolean, called primitive data types. In meta-models, they are used as types of class attributes. In addition, OCL has two predefined types representing the top (OclAny) and bottom (OclVoid) elements of its type hierarchy. Template types are Collection(T) and Tuple(T1, T2) whose parameters T, T1 and T2 are applied to other types. Collection is an abstract type: its concrete sub types are Set, OrderedSet, Bag and Sequence and differ with respect to frequency and ordering of the contained elements.

(1) The simplest form of invariants is formulated within a *propositional logic* of navigation expressions: In OCL expressions, object structures can be traversed using the so-called dot notation. Accessible elements are objects (i.e. instances of classes) and their features (i.e. attributes and opposite association ends of classes). In this simplest form of invariants, we consider only navigation expressions that yield a single-valued return type (for multiplicities with a lower and an upper bound of 1). If the return type is an object type, the existence of such an object is checked. If the return type is a primitive one, the resulting values can be compared to another by some comparator. In addition, all operators known from propositional logic such as not and and can be used in this category of OCL invariants.

(2) More advanced forms of OCL invariants form a *first-order logic* of navigation expressions: In addition to the OCL features recalled above, all the collection types may be used together with most of their operators. Depending on a feature's multiplicity, a navigation expression may result either in a single-valued return type (for multiplicities with an upper bound of 1) or in a multi-valued type, more precisely in a set (for multiplicities with an upper bound greater than1) or a sequence for ordered associations. If, in a multi-valued reference, there does not exist any target object, the navigation results in an empty set. The multiplicity 0..1 is not supported in this case as the absence of an appropriate value would yield null representing the only value of bottom type OclVoid. Moreover, all expressions are defined such that they do not yield the OCLinvalid value as return value. In practice, this means that expressions have to apply proper safety checks. Hence, this form of invariants still sticks to a two-valued logic.

All forms of collection or tuples may be used. The following collection operations are supported: Collections may be constructed with operations like Set{...} and Sequence{...} or one of the implicit constructors including(e) and excluding(e). An implicit constructor takes an element e as parameter and adds it to a given collection (including) or removes all occurrences of it from a given collection (excluding). Filter operations like select(BExp), reject(BExp), and any(BExp) are used to filter collection elements according to the evaluation of the Boolean expression BExp. Extraction operations extract some information from the given collection except for Boolean values. Examples of this kind of operations are size() and union(Collection(T)). Op-

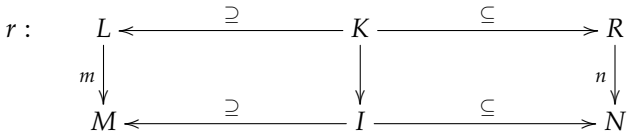
eration size() returns the number of elements in a collection and may be used to compare the size of a collection with some fixed integer but not with another set size neither with some attribute value. Finally, OCL provides a number of operations returning Boolean values. For checking the existence of elements within a collection, isEmpty() and notEmpty() can be used, for example. In order to test the membership in a collection, the operations includes(e) and excludes(e) for testing on single elements e as well as includesAll(Collection(T)) and excludesAll (Collection(T)) for testing if element collections are available. Let expressions are supported as long as the specifications of additional variables and operations use the OCL features in this category and are non-recursive.

(3) The most advanced form of OCL invariants allows all possible OCL expression and operations based on a four-valued logic, i.e. Boolean={true, false, null, invalid} is supported here. Now, the multiplicity 0..1 is supported; the absence of an appropriate value would yield null representing the only value of bottom type OclVoid. Furthermore, collection operations collect and iterate may be used which allow specifying recursive functions. Any kind of let expressions is supported which means that the specifications of additional variables and operations may be recursive. Finally, operations closure and count may be used; size may be used without any restriction. Custom operations are supported as long as they are queries, i.e. do not change the object structure.

3.3. Model Transformations with Henshin

Model transformations can be defined inside one modeling language or between two different languages. They are called *endogenous* or *exogenous*, respectively. While endogenous transformations are used, for example, to edit or optimize instance models, exogenous transformations are typically used to translate a model to another model or to text. There are various approaches to model transformation based on several paradigms. A comprehensive overview of model transformation approaches and languages is given in (Lúcio et al. 2016; Kahani et al. 2019). As most of them are able to transform EMF models, it depends very much on the purpose which model transformation approaches or languages to use.

Henshin[1] (Arendt et al. 2010) is a language and tool environment for rule-based transformations of EMF models based on graph transformation concepts. A transformation rule specifies all the model changes that shall be performed in one transformation step. The left-hand side L of a rule represents a pattern that has to be found to apply the rule. The difference between the left and the right-hand side R of a rule specifies its actions. The intersection $K = L \cap R$ denotes the part that is not changed, the part that is to be deleted is defined by $L \setminus K$, while $R \setminus K$ defines the part to be created. A *model transformation (step)* $M \overset{r,m}{\Longrightarrow} N$ between two models M and N is defined by first finding a *match* m, that is a mapping of the left-hand side L of rule r into M. Model N is obtained erasing $m(L \setminus K)$ in M and adding a new copy of $R \setminus K$ using mapping n. The following diagram shows all the models and interrelations that play a role in a transformation step.



If model transformations shall preserve the validity of models concerning OCL constraints, there are basically two strategies to achieve that: (1) A valid model is transformed first and the resulting model is checked for validity thereafter. This can be done, for example, with the OCLinEcore validator (OCLinEcore 2019). (2) Another possibility is to translate OCL constraints to graph constraints and to enhance model transformation rules with application conditions that are derived from those graph constraints. An application condition restricts the applicability of a rule as a match has to be found that satisfied an additional condition. The second approach is automated with OCL2AC[2] (Nassar et al. 2019, 2018), tool support which is based in Henshin. It is demonstrated at an example in Section 5.2.

4. An MDE Approach to Bytecode Processing

In this paper, our goal is to leverage techniques from model-driven engineering for defining code Java bytecode manipulations by defining them as model transformations. By defining the bytecode verifier rules as OCL constraints it will become easier to test byyecode manipulations because it can immediately be checked whether the result of the manipulation is well-behaved bytecode (c.f. Section 2.2) without executing the bytecode. Moreover, it will become possible to analyze whether a given manipulation will always preserve well-behavedness in bytecode.

For these purposes, we will present a meta-model for Java bytecode in Section 4.1 (precursors of it have already been published (Yildiz, Bockisch, et al. 2017; Yildiz, Rensink, et al. 2017)). In Section 4.2, we will show how to define OCL constraints for the verifier rules choosing a representative set of rules.

We have developed a suite of Eclipse plugins to perform *Modular Bytecode Engineering and Analysis based on MDE*. We call our project *ModBEAM* for short. One of these plugins provides our meta-model which is called *JBC* (short for *Java Bytecode*). *ModBEAM* provides a nature that can be added to Eclipse Java projects such that an instance model is created for every `.class`-file that is contained in its source folder. It is placed in the folder `jbc-model` and it is updated when a `.class`-file changes. The user can choose between generating one single model which contains call class definitions or many models which contain one single class each. In the first style, the models are always created in a full-build, while the second style supports incrementally building the models. Given a JBC instance model, the ModBEAM plugins can also generate proper Java bytecode from such a model again, which is also updated, when the model changes.

[1] Available at: www.eclipse.org/henshin

[2] Available at: ocl2ac.github.io/home

In addition to the meta-model and the work flow, ModBEAM includes an Xtext plugin that provides a textual editor for JBC instance models. This editor automatically validates the OCL invariants contained in our meta-model and marks violations thereof. But we also provide the possibility to persist JBC instance models in the XML format. The implementation of ModBEAM, including the meta-model with the OCL invariants, the Eclipse plugins and the Xtext editor can be accessed from our Git repository at https://bitbucket.org/modbeam/dev/.

4.1. A Meta-model for Java Bytecode

The purpose of the JBC meta-model is to act as an alternative, more accessible representation of Java `.class` files, which is equivalent to the original. Therefore its structure mainly follows the organization of Java class files as defined in the Java Virtual Machine specification (Lindholm et al. 2020). Nevertheless, the original goal of the bytecode format was to facilitate an efficient interpretation of `.class` files and not to be human-readable or easy to manipulate. For this reason, our meta-model represents information differently in some places. Examples will be given along with the discussion of the meta-model in Listung 1.

The JBC meta-model is defined using Ecore. In general, each kind of entity of the class file format (such as method declarations, attributes and instructions) is represented as one ECore class in the meta-model. Lexical nesting is represented as a containment relationship, i.e., a method is contained in the class that declares it. The top-level of the Java class file format follows the Java source language. A simplified and incomplete version of the most relevant EClasses is shown in Listing 1.

Project is the root of the meta-model and has no other purpose than to contain Clazz EObjects[3], which stands for a `.class`-file each. A series of Boolean attributes in Clazz specifies which modifiers have been used in the corresponding Java class. The TypeReference basically encapsulates a type name and is used here to refer to the super class and implemented interfaces.[4] Besides this type-related information, a Clazz refers to EObjects modeling the methods and fields defined within the class. To simplify the analysis of instance models, all containment references are bidirectional.

The EClasses Method and Field (not shown in the listing) follow the same design strategy. The property descriptor references a MethodDescriptor (not shown here) which encodes the type names of the method's list of parameters.

In contrast to the sequential organization of instructions in actual bytecode, our meta-model organizes them in a control-flow graph, which is a representation used in virtually flow-sensitive code analyses. The property firstInstruction of a Method refers to the root of this graph, i.e., the instruction at which the execution of a method starts. The property instructions is required because all EObjects of type Instruction must be contained in another EObject and thus, holds an unordered collection of all instructions of the method. Since a method may be abstract, the property firstInstruction is optional; in that case, the method does not contain any instruction at all.

The abstract EClass Instruction is the root of a type hierarchy, which contains one concrete EClass for each possible Java bytecode instruction. Each instruction type may have specific attributes; an instruction for invoking a method, for example, has an attribute MethodReference. All instructions have in common that they contain a reference to outgoing control flow edges, represented by a subtype of the EClass ControlFlowEdge. That EClass has two properties, namely a reference to the start Instruction and a reference to the end one.

```
1  package jbc : jbc = 'https://modbeam.bitbucket.io/jbc' {
2    class Project {
3      property classes : Clazz[*|1] { ordered composes };
4    }
5    abstract class NamedElement {
6      attribute name : String[1];
7    }
8    abstract class MethodNamedElement extends NamedElement {
9    }
10   abstract class AccessFlaggedElement {
11     attribute public : Boolean[1];
12     attribute private : Boolean[1];
13     attribute protected : Boolean[1];
14   }
15   class TypeReference {
16     attribute typeDescriptor : String[1];
17   }
18   class Clazz extends NamedElement, AccessFlaggedElement {
19     attribute final : Boolean[1];
20     attribute super : Boolean[1];
21     attribute abstract : Boolean[1];
22     attribute name : String[1];
23     property superClass : TypeReference[1] { composes };
24     property interfaces : TypeReference[*|1] { ordered composes };
25     property methods#class : Method[*|1] { ordered composes };
26     property fields#class : Field[*|1] { ordered composes };
27     // et cetera
28   }
29   class Method
30   extends MethodNamedElement, AccessFlaggedElement {
31     attribute final : Boolean[1];
32     attribute abstract : Boolean[1];
33     property class#methods : Clazz[?];
34     property descriptor : MethodDescriptor[1] { composes };
35     property instructions#method : Instruction[*|1] { composes };
36     property firstInstruction : Instruction[?];
37     // et cetera
38   }
39   class Field extends NamedElement, AccessFlaggedElement {
40     attribute final : Boolean[1];
41     attribute volatile : Boolean[1];
42     property class#fields : Clazz[?];
43     property descriptor : TypeReference[1] { composes };
44     // et cetera
45   }
46   abstract class Instruction {
47     property method#instructions : Method[1];
48     property outEdges#start : ControlFlowEdge[*|1] {
49       ordered composes };
50     // et cetera
51   }
52   abstract class ControlFlowEdge {
53     property start#outEdges : Instruction[1];
54     property end : Instruction[1];
55   }
56 }
```

Listing 1 Relevant part of the JBC meta-model

[3] This spelling is used to avoid conflicts with the type java.lang.Class when generating Java classes from the Ecore model.

[4] Type names are specified following the "Field Descriptor" notation, cf. (Lindholm & Yellin 1999, Section 4.3.2).

Program	Classes	Methods	Model Size	Time
LiveGraph	131	350	24049	18s
Weka	1041	8322	756063	764s
Groove	1482	9232	418269	1480s

Table 1 Preliminary performance of JBC instance model creation from `.class`-files.

There are three major kinds of control flow edges represented by subtypes of ControlFlowEdge. The type UnconditionalEdge is used to define sequential control flow and can be used for most of the instructions. Some instructions are so-called branching instructions; when these are executed, the control flow can continue at different instructions depending on a condition that is checked by the branching instruction. Most branches are binary which means that the corresponding Instruction EObject has two outgoing edges of the type ConditionalEdge. That EClass has a Boolean attribute condition which specifies whether one or the other edge is followed depending on whether the instruction's condition is satisfied or not. There are also instructions with n-ary branches to express the switch statement from Java source code. Outgoing edges for them are modeled in a similar way. Lastly, there are instructions specifying control flow in a try block with a corresponding catch block to formulate an exception. Those instructions have outgoing edges of the type ExceptionalEdge.

4.1.1. Digression: Performance of ModBEAM

To illustrate the feasibility of using the ModBEAM meta-model, we present a few details on the implementation and its performance. More details can be found in (Yildiz, Bockisch, et al. 2017).

We use the ASM bytecode manipulation toolkit (Bruneton et al. 2002) to parse class files when creating instance models, as well as for generating bytecode files from instance models. For the performance study below, we have used the XMI format for persisting instance models, and we only consider full builds.

For our performance study, we have used three real-world, open-source projects of different sizes. The results are presented in Table 1. The table columns show the Java project's name, the number of classes and methods it contains, the number of EObjects in the instance model and the time spent on creating the instance-models. The required time ranges from 18 seconds for a small project to almost 25 minutes for a relatively large project with nearly 1,500 classes. While 25 minutes is a long time, a full-build is rarely necessary. A thorough analysis of the impact of incremental instance model creation in ModBEAM on the development work flow still has to be performed. In this study, the average time for creating an instance model for a `.class` file was between 0.14 seconds and 1 second. Since incremental creation works in the background and therefore, this does not cause a perceivable delay during the development. For these reasons, we think that our approach is sufficiently scalable for use in practice. It should also be mentioned that we have not explored optimization opportunities yet.

4.2. Bytecode Verification Rules as OCL Constraints

In Section 2.2, we have presented an overview of the Java bytecode verifier rules as they are presented in the Java Virtual Machine specification. In this section, we will revisit all those example rules and show their definition in OCL. This subset of the verifier rules has been selected to be representative with regard to the complexity of the logic statement and the information accessed by them. We have analyzed each rule to determine the proper EClass to be used as the context and defined an invariant for the rule. In Section 3.2, we have classified OCL invariants into 1. *propositional logic*, 2. *first-order logic*, and 3. *general OCL expressions*. Here, we consider our example invariants and discuss the level of logic they have.

Local information. Rule 1 applies to every entity with a name such as a Field, FieldReference or Clazz. All these EClasses extend the abstract EClass NamedElement which defines the corresponding invariant. For method names, there is the additional rule 2, so the invariant is strengthened in this case. Therefore, we define the EClass MethodNamedElement—extending NamedElement—as super class of Method with an additional invariant.

```
1 context NamedElement inv nameHasNoIllegalCharacters:
2   name.indexOf('.') = 0 and name.indexOf(';') = 0
3   and name.indexOf('[') = 0 and name.indexOf('/') = 0;
4 context MethodNamedElement inv nameHasNoIllegalBrackets:
5   (name <> '<init>' and name <> '<clinit>')
6   implies (name. indexOf('<') = 0 and name.indexOf('>') = 0);
```

Listing 2 OCL invariants for verification rules 1 and 2.

Rule 3 restricts the kinds of visibility allowed; it is defined for fields. Equivalent rules exist for classes and methods, for all of them at most one visibility modifier may be specified. Given our meta-model, we do not need to define this invariant multiple times. Instead we use the EClass AccessFlaggedElement as context which contains Boolean attributes that specify the presence of modifiers. This EClass is extended by all EClasses representing entities with visibility modifiers.

```
1 context AccessFlaggedElement inv hasAtMostOneAccessFlag:
2   (public implies (not private and not protected)) and
3   (private implies not protected);
```

Listing 3 OCL invariant for verification rule 3.

The next example in this category of rules is Rule 4. It specifies that the type of a field must not be an array type with more than 255 dimensions. In fact, this constraint applies to each entity with a type reference following the"field descriptor" notation, including method result types. An array type is encoded in this notation by prepending the character `[` for each array dimension. In our meta-model, we define the EClass TypeReference which is used for all type references; therefore, it is the ideal context for specifying this invariant.

```
1 context TypeReference inv arrayDimensionValid:
2   typeDescriptor.characters()->count('[') <= 255;
```

Listing 4 OCL invariant for verification rule 4.

Rule 9 prescribes that every class except java.lang.Object must have a superclass specified. The respective OCL invariant in Listing 5, therefore, requires that a type reference is specified for the superClass property. This is sufficient since the existence of the reference class is checked by the linker and not the bytecode verifier.

```
context Clazz inv hasDirectSuperClass:
  name = 'java/lang/Object' or not superClass.oclIsUndefined();
```

Listing 5 OCL invariant for verification rule 9.

The first three invariants presented above clearly belong to the simplest category of OCL expressions, namely those that use *propositional logic* only. The one in Listing 4, however, belongs to the second category, *first-order logic*, since the size of a collection is compared against a constant value. The invariant in Listing 9 uses the OCL operation oclIsUndefined which makes use of four-valued logic. Thus, this invariant belongs to the category of *general OCL constraints*.

Directly referenced information. Rule 5 could be considered as an invariant of a class (it must not contain duplicate fields) or of a field (it must be unique within its class). We decided to define it as an invariant of the EClass Field because, in case of a violation, the error reported will be more specific. This means that the two (or more) duplicate Field EObjects will be marked. For the second rule in this category, Rule 6, the JVM specification is more clear about the context which is IreturnInstruction: an instruction returning an integer is legal only when it appears within a method with a matching return type. Let us recall here that a primitive type in bytecode is represented by a single letter and the types boolean, char, byte, short and int are all represented as integer values internally.

```
context Field inv noNameAndDescriptorDuplicates:
  class.fields->forAll(field | (field <> self)
  implies (field.name <> self.name
    or field.descriptor.typeDescriptor <>
      self.descriptor.typeDescriptor));
context IreturnInstruction inv properReturnType:
  let desc = method.descriptor.resultType.typeDescriptor
  in desc = 'I' or desc = 'Z' or desc = 'B' or desc = 'S' or
    desc = 'C';
```

Listing 6 OCL invariants for verification rules 5 and 6.

Both invariants navigate along eOpposite edges and use simple equality comparison of primitive attribute values. The second invariant does not use other operators and therefore, belongs to the category of propositional logic. As the first invariant uses universal quantification, it belongs to the category of first-order logic. It also navigates along a containment edge, which does not add to the complexity.

Traversing information referenced by name. In the Java language, entities such as classes and methods are referenced by name. Also in the bytecode, method-invocation instructions must specify the name of the called method and a class definition must specify the name of the superclass. We formulate this requirement in the EClasses MethodReference and TypeReference.

To resolve name references to a Clazz, we have implemented the operation in Listing 7. It searches through all Clazzes in the instance model and returns the one whose attribute name matches the provided name. The comparison is not made with the class name directly but with the expression 'L' + cls.name + ';' because this is the notation for class-type descriptors in the Java virtual machine.

If a unique matching Clazz is found, it is returned by the operation. Otherwise, it returns null. This may be the case, for example, when a class is referred to that is loaded dynamically at runtime from a source that was not available when the instance model was created. The operation also returns null when multiple matches are found as ambiguous class definitions would exist. This cannot be the case if the instance model is created by our tooling.

```
context Clazz::getClassFromReference(ref : String[?]) : Clazz[?]
  body: let availableClasses = Clazz.allInstances()->
    select(cls | ref = 'L' + cls.name + ';')
  in if(not availableClasses->isEmpty())
  then availableClasses->first()
  else null
  endif;
```

Listing 7 Operation to resolve a Clazz EObject from a type descriptor.

For method references we have implemented an analogous mechanism. This means that an operation getMethodFromReference is added to the context Method, which internally uses the operation getClassFromReference to lookup the Clazz for the receiver type of the method reference.

The operation isAssignmentCompatible(target : Clazz[?], operand : Clazz[?]) : Boolean[?] (not shown here) traverses the class hierarchy by recursively calling getClassFromReference to determine if the type referred to by operand is a sub type of the one referred to by target. Using these operations, we can formulate the bytecode verifier rule 7 as an OCL invariant.

```
context InvokevirtualInstruction inv invocationRespectsProtected :
  let target = Method::getMethodFromReference(methodReference)
  in (target.protected and target.class.package() <>
    self.method.class.package())
  implies TypeReference::isAssignmentCompatible(
    self.method.class, target.class);
```

Listing 8 OCL invariant for verification rule 7.

Rule 10 specifies that no final methods may be overridden. The corresponding OCL invariant is shown in Listing 9. The invariant uses the custom operation getAllMethods(super : Boolean [?]) : Method[*|1] which is defined in the context Clazz. Similar to isAssignmentCompatible, this operation recursively calls getClassFromReference to follow the chain of superclasses and to collect all methods defined in those classes. In both invariants, recursive custom operations are used. Therefore, they belong to the hardest category of *general OCL constraints*.

```
context Method inv overridesNoFinalMethod:
  not class.getAllms()->exists(m |
    (not private) and m.name = self.name and
    m.descriptor = self.descriptor and (m.public or m.protected or
      not (m.public or m.protected or m.private) and m.final));
```

Listing 9 OCL invariant for verification rule 10.

Rule 11 states that a final class must not be subclassed. Listing 10 shows the equivalent OCL invariant. Since it is costly to locate all sub classes for a given class, we invert this condition and specify that the super class must not be final. We must explicitly handle the case that the name of the super class is java/lang/Object as that class does not have a super class.

```
context Clazz inv superClassNotFinal:
  name = 'java/lang/Object' or not self.getClassFromReference(
    self.superClass.typeDescriptor).final;
```

Listing 10 OCL invariant for verification rule 10.

This invariant uses the custom operation getClassFromReference. This operation is not recursive such that it could also be inlined into the invariant (only at the expense of readability). The most complex operations used in this invariant are allInstances and select so that it is *first-order*.

Traversing local information. This category of verifier rules is concerned with type checking the arguments of instructions that are passed along the operand stack or along with local variables. As outlined in Section 2.2, this requires the computing of the stack layout and the layout of local variables at each instruction. Implementing this computation in OCL is lengthy and its comprehension requires a deeper knowledge of how instructions in Java bytecode look like. Thus, showing the implementation would be beyond the scope of this paper.

The overall implementation strategy can, nevertheless, be presented here. The layout of the stack and the local variables at an instruction can be computed by traversing all possible paths from the first instruction to the instruction in question and collecting information along the way. We have implemented an operation in the context of the Instruction EClass, called simulateStack, for performing this traversal. Since there may be any number of instructions in a method, the control-flow paths to be traversed may be of any length. For that reason, the operation simulateStack is implemented recursively. In general, the control-flow graph of a method may contain cycles (namely when the method contains loops). To avoid endless recursion, this operation also has to keep track of EObjects Instruction already visited.

The operation simulateStack is used in an invariant of Method , cf. Listing 11, which checks the compliance of the stack layout and the layout of local-variable computed with the layout expected by each instruction. It is used to specify the invariant for bytecode verifier rule 8, among others. The invariant calls the operation simulateStack on the method's first instruction and passes the following arguments: the initial stack layout, the initial local-variable layout and the sequence of already visited instructions.

```
context Instruction inv isTypeSafe:
  (not firstInstruction.oclIsUndefined())
  implies firstInstruction.simulateStack(Sequence{},
    localLayoutFromParameters, Sequence{});
```

Listing 11 OCL invariant for verification rule 8.

Regarding the classification of OCL constraints, the above invariant belongs to the highest class of *general OCL expressions*. This is because we use recursive custom operations and maintain the state of all the EObjects of type Instruction that are already visited, to avoid endless recursion.

5. Example: Bytecode Manipulation as Model Transformation

To demonstrate the benefits of our approach of specifying bytecode verifier rules as OCL constraints, let us have a look at how bytecode manipulation can be developed as model transformations based on our meta-model. As an example, we will develop a simple mock-up tool to develop unit tests before implementing a method. It will replace the stub body of that method with an implementation returning predefined values to make the test cases pass.

For illustration, let us assume that we are developing a program to compute the n^{th} Fibonacci number and start by writing a test driver as well as an empty method, as seen in the listing below.

```
public class Fibonnaci {
  public static void main(String[] args) {
    assert fibonacci(1) == 1;
    assert fibonacci(4) == 3;
  }
  public static int fibonacci(int n) {
    return 0;
  }
}
```

Listing 12 Empty implementation of the method fibonacci().

Our mock-up tool is now supposed to insert code into the method fibonacci() that returns the correct value for the first two method calls to make the test pass. For brevity, we assume that a class test.MockDataProvider with a static method next() exists that returns 1 and 3 for the first two calls and -1 for all others. The bytecode manipulation needs to insert an instruction into the method fibonacci() that invokes the method next() as new first instruction. More instructions need to be inserted to test whether the result is negative. In that case, the execution should jump to the original first instruction (i.e., the first instruction compiled from the statement return 0;), otherwise the execution should continue at an instruction returning the result of next(). That instruction also needs to be inserted. In Java bytecode, different return instructions exist, one for each kind of result type (int-like, long, float, double and reference type). We generate an ireturn instruction that returns an int value.

Applying this manipulation to the bytecode of the class in Listing 12 is successful and the execution of the test driver will succeed. As we know that the Fibonacci numbers grow very quickly, we decide to change the method's return type to long. After that change, the application of the same manipulation will produce illegal code that will be rejected by the bytecode verifier. When the manipulated class is loaded by the Java Virtual Machine, the error message shown in Listing 13 on page 12 is produced.

To understand that error, developers of bytecode manipulations need to perform the following steps: (1) Execute the transformed program, leading to a verification error. (2) Read the error message to figure out the location of the violating instruction in terms of its bytecode index, the containing method

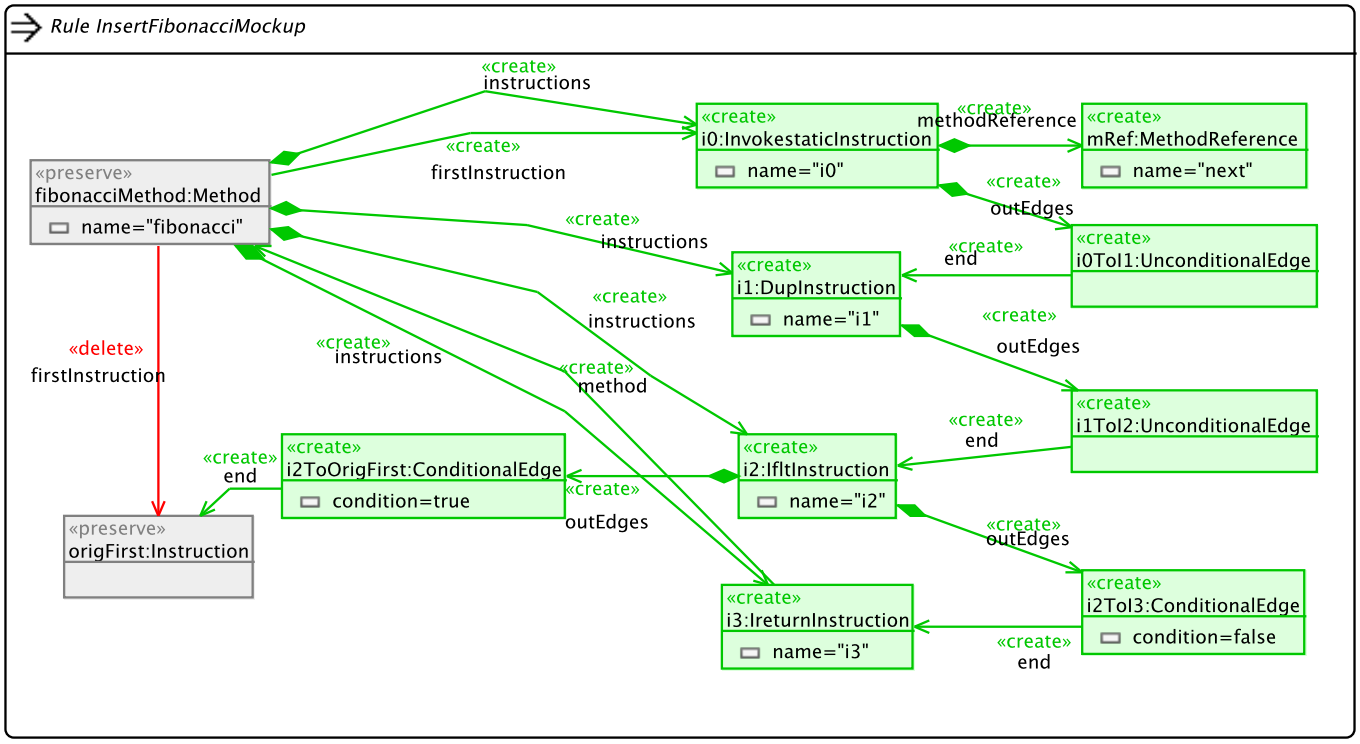


Figure 1 A sample transformation rule in Henshin.

and the containing class. (3) Open a bytecode viewer for this class and locate the correct instruction. (4) Analyze the violation. (5) And finally, locate the part of their implementation that is responsible for creating the faulty instruction.

5.1. Bytecode verification by OCL Checking

To specify bytecode manipulations such as the one presented above, we can use model transformation rules instead. For this case example, we use Henshin to formulate the transformation rule, which is shown in Figure 1. When applied to a JBC instance model, this rule identifies all parts of the model that has the expected structure, which is defined by the Method and Instruction EObjects (stereotype <<preserve>>) and firstInstruction reference (stereotype <<delete>>) in the figure. This means that all methods with the name "fibonacci" that have a first instruction are selected. Such a pattern occurs exactly once in our example. The matched part of the instance model is then transformed by removing the reference with the stereotype <<delete>> and creating the EObjects and references with the stereotype <<create>>.

The example rule creates the following instructions linked to sequential control flow by UnconditionalEdge EObjects: InvokestaticInstruction invokes the static method test .MockDataProvider.next()[5] which will push its result on the operand stack, DupInstruction duplicates this value and IfltInstruction tests whether the result is negative. The last instruction pops a copy of the value from the operand stack such that it contains a single value which is the result of the next() method. This instruction has a conditional control flow defined by the two ConditionalEdge EObjects. The two possible targets are IreturnInstruction, if the result of next() was not negative, to return this value, and the original first instruction to perform the stub implementation, otherwise. The invocation instruction will become the new firstInstruction.

The rule explicitly defines the reference method from the IreturnInstruction to the Method. This is the EOpposite reference instructions and does not have to be specified. We have included it into the rule because the OCL2AC tool, applied in the following subsection, can use it to create a more readable application condition.

In our approach, the first three steps of understanding an error listed for the classical approach above are not needed. Program execution is not necessary because the verification is performed by validating the transformed instance model against our OCL invariants. The resulting instance model can be viewed and EObjects of violating instructions are directly annotated with error messages.

Figure 2 shows a screenshot of our Xtext editor for JBC instance models. The editor shows an excerpt of the model for our Fibonacci implementation with result type long (the type descriptor ("J") in bytecode) after our transformation has been applied. It can be seen immediately that the IreturnInstruction violates the invariant *properReturnType*, cf. Listing 6.

The creative task of understanding the mistake in step 4 can be supported by tracing model transformations such that the transformation rules that are responsible for creating failure

[5] The invocation target is specified by a MethodReference. This EObject references further EObjects, which are not shown in the example.

```
Error: Unable to initialize main class Fibonacci
Caused by: java.lang.VerifyError: Bad return type
Exception Details:
 Location:
  Fibonacci.fibonacci(I)J @7: ireturn
 Reason:
  Type integer (current frame, stack[0]) is not
               assignable to long
               (from method signature)
 Current Frame:
  bci: @7
  flags: { }
  locals: { integer }
  stack: { integer }
 Bytecode:
  0000000: b800 2959 9b00 04ac 09ad
 Stackmap Table:
  same_locals_1_stack_item_frame(@8,Integer)
```

Listing 13 Error message by the bytecode verifier.

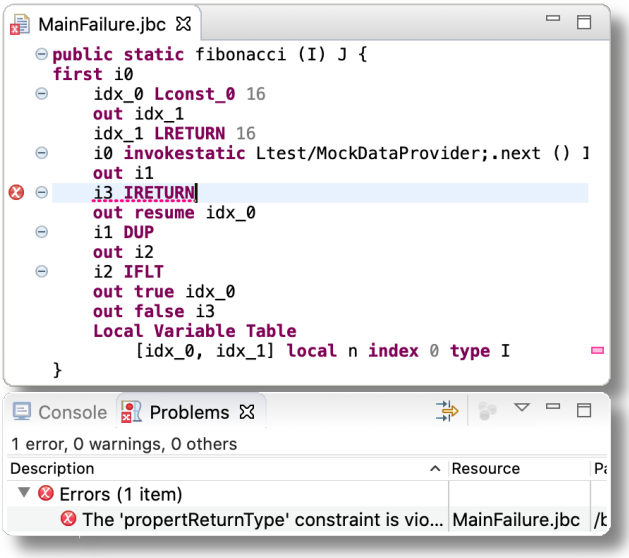


Figure 2 Xtext editor with the transformed instance model violating an invariant.

is identified. Considering Henshin, however, such tracing is not implemented in a user-friendly way yet. Step 5, however, is supported with our solution as each model resulting from a transformation can be checked against the bytecode verifier rules using the OCL constraints of our meta-model.

5.2. A Correct-by-Construction Approach

In contrast to performing bytecode verification by OCL checking as above, we present an approach here that enhances a model transformation rule with application conditions. They control the applicability of the transformation rule such that all possible applications lead to transformation results that respect the given set of constraints. The construction of such application conditions is implemented in an Eclipse-based tool called OCL2AC (Nassar et al. 2018, 2019). For a given Henshin rule and an OCL constraint, OCL2AC can automatically construct *constraint-preserving application conditions*. Such an application condition ensures that the rule applies to a valid model if and only if the resulting model after the rule application satisfies the constraint. OCL2AC implements a formal approach; its correctness is shown (Radke et al. 2018; Nassar et al. 2019). However, OCL2AC comes with the following limitations: It cannot yet translate all OCL constraints but only those that are two-valued and in first-order logic. Thus, the expression oclIsUndefined and the operation iterate are not supported. Moreover, there is no support to translate user-defined operations.

In the following, we show how to enhance the Henshin rule shown in Figure 1 to preserve the OCL constraint properReturnType with the help of OCL2AC. This means that the rule will be turned to be applicable to a valid model if and only if the model that results from an application of that rule satisfies the constraint. The integration process consists of two main steps as follow: First, the OCL constraint is translated into a semantically equivalent graph constraint as shown for our example in Figure 3. A graph constraint is a formula in first-order logic; it is formulated over (model) graph patterns that are checked for (non-)existence in an instance model. The graph constraint in Figure 3 states that for each IreturnInstruction, there exists a Method connected to a MethodDescriptor, which is connected to a TypeReference. The value of the attribute typeDescriptor may be C, S, B, Z, or I. The name self requires the nodes of type IreturnInstruction to be the same.

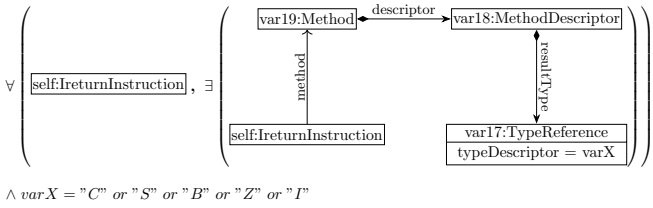


Figure 3 Graph constraint properReturnType

Second, the graph constraint is integrated as a constraint-preserving application condition for the rule. This is mainly done by overlapping the right-hand side of the rule with the constraint in every possible way to find all possibilities how a rule application may violate the constraint. Often, the resulting application condition contains unnecessary checks that can be eliminated. Figure 4 shows the relevant snippet of the application condition that is constructed from the graph constraint in Figure 3. It requires that the Method that will contain the IreturnInstruction has a MethodDescriptor connected with a TypeReference, and the value of its attribute typeDescriptor must be C, S, B, Z, or I.

Given a metamodel with a set of constraints and a set of rules, we show in (Nassar et al. 2019) that a constraint has to

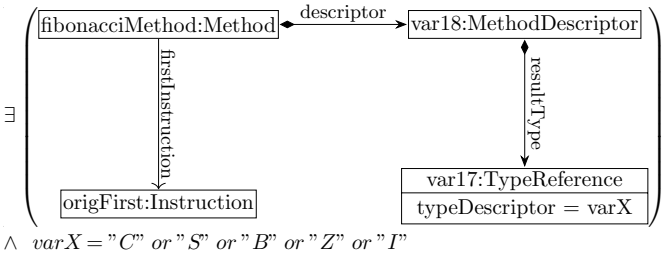


Figure 4 Snippet from the resulting properReturnType-preserving application condition

be integrated into a rule only if it can violate that constraint. Thus, in general, not all the rules have to be enhanced with application conditions. The decision of whether to integrate a constraint into a rule is based on a static analysis of their declarative specifications.

6. Related Work

Our considerations of related work cover several areas: approaches to bytecode manipulation, formalization of Java bytecode verifier rules, meta-models for general-purpose programming languages, especially Java, and using MDE to implement bytecode manipulations.

In the programming-language-implementation community, there are approaches to support developers of bytecode manipulations by offering high-level interfaces (e.g., (Marek et al. 2015; Bytebuddy 2020)). They offer convenience and prevent some kinds of mistakes but also limit the possible kinds of manipulations. General toolkits such as ASM or BCEL (Bruneton et al. 2002; BCEL 2020) facilitate the checking of generated bytecode against the verifier rules of the JVM specification. But they have several shortcomings compared to our approach: First, there is no persistent representation of the bytecode to which the error messages can be attached. Second, the manipulation must be implemented in an imperative way and thus, cannot be analyzed itself.

There are also some *formalizations of Java bytecode verifier rules*. The JVM specification itself (Lindholm et al. 2020) uses Prolog terms to define some of the verifier rules. But it does not an actual implementation for all relations so that these rules are not executable. Leroy has formulated the rules as data-flow equations (Leroy 2003) but not implemented in a machine-processable language. Reynolds (Reynolds 2013) follows a very similar approach to ours. He defines a meta-model for Java bytecode in the modeling language Alloy which contains constraints for some of the verifier rules. Since Alloy is restricted to first-order logic, not all verifier rules can be supported.

There are already *meta-models for general-purpose programming languages*, in particular for Java. SPOON (Pawlak et al. 2016) is a meta-model for the Java language; it has been developed specifically for supporting analysis and transformation of source code. Analyses and transformations in the SPOON approach are implemented in plain Java and therefore, are not as declarative as in our approach. This means that analyses of the transformations themselves, as we have shown using the OCL2AC tool, are out of reach.

There are EMF-based meta-models of the Java source language are provided by JaMOPP (Heidenreich et al. 2009) and MoDisco (Bruneliere et al. 2014). These meta-models do not contain invariants for semantic properties yet. But since they are defined in Ecore, OCL can be used of course to specify such invariants also there. Nevertheless, we believe that bytecode is more appropriate for implementing program transformations in general as bytecode can always be assumed to be present and it can also be modified more flexibly.

Regarding other meta-models for bytecode, besides the work by Reynolds discussed above, we are aware of only one further approach by Eichberg et al. (Eichberg et al. 2010). They provide a meta-model for Java bytecode as an XML Schema. The focus of that approach is on supporting static analyses, however, it would also be possible to implement bytecode manipulations as XML transformation using, for example, XSLT. There are also approaches to validate XML documents based on OCL invariants. However, Ecore is more commonly used in model-driven engineering and more tools exist for processing Ecore-based instance models. We have demonstrated this benefit by using the powerful OCL2AC tool to analyze a transformation itself.

When using *MDE to implement bytecode manipulations*, there are tools that offer interesting functionality: The UML-based Specification Environment (USE) (Gogolla et al. 2007) by Gogolla et al. supports the execution of UML models and the checking of OCL constraints. As advanced features, USE can check the consistency of models, and moreover, the independence and implication of invariants. In the future, it is interesting to check the independence and implications of the OCL constraints deduced from bytecode verifier rules for proving their soundness.

We have seen that *model transformation* can be used to implement program transformation. Having translated Java bytecode to an instance model, model transformations shall enhance and rewrite it; so an endogenous model transformation approach is needed. While there are several endogenous model transformation approaches for EMF such as ETL (Kolovos et al. 2008), Henshin, ViaTra (Varró et al. 2016), Henshin is the only one that can enhance model transformation rules with application conditions such that resulting transformations are constraint-preserving. While Henshin uses a translation of OCL constraints to graph constraints (Radke et al. 2018), Desai and Gogolla (Desai & Gogolla 2019) use OCL directly to develop pre- and post-conditions of state transitions in operation calls. In addition, a certain kind of independence between different program analyses shall be ensured. This requirement demands a model transformation approach that supports independence analyses between transformation modules. Henshin is the only transformation tool with such support. It offers a conflict and dependency analysis of transformation rules. It is up to future work to consider a toolbox for modular bytecode engineering and analysis.

7. Conclusion

As standard approaches for developing and testing program transformations need to use several technologies that are not well linked, we investigated how model-driven engineering can be used in this context and when this approach is promising. In the context of Java bytecode, we analyzed the semantic rules of the bytecode verifier according to their complexity factor, presented a meta-model for Java bytecode, showed how to express the semantic rules as OCL constraints using this meta-model, and showed that using MDE for developing program transformations can provide more immediate guidance and feedback to the developer.

As we report on work-in-progress, there is a number of improvements to be made and open questions to be answered. To start with, we plan to further optimize our meta-model and especially, the invariants; in particular, we will focus on the following problems. First, there are some properties in our meta-model that are optional. Using them in invariants may lead to the use of four-valued logic. We will investigate if we can improve the meta-model such that optional properties are avoided. Then, corresponding invariants may be reformulated such that they will belong to a simpler logic class and tools like OCL2AC could analyze bytecode manipulations in more cases. Second, when using the stack and local-variable layout for checking verifier rules, violations are currently attributed to the Method instead of an Instruction EObject. In the future, we will attempt to implement this check as invariants of instructions to produce more specific error messages.

Expressing semantic rules of programming languages as invariants of the languages meta-model can provide a huge benefit to developers of bytecode manipulations, as we have shown in this paper. We will therefore further enrich our work with additional semantic rules, beyond the ones of the Java bytecode verifier, to extend this advantage even more.

A first demonstration of the benefits of implementing bytecode manipulations as model transformation has been given in our case example. In future, we will investigate more complex bytecode manipulations for a variety of purposes to find out how far the usage of MDE concepts and techniques allows developing correct bytecode manipulations in a developer-friendly way.

Acknowledgments

This work was partially funded by the German Research Foundation (DFG) projects "Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages" (grant no. HA 2936/4-2 and TA 294/13-2). We would also like to thank Mehmet Akşit, Arend Rensink and Bugra Yildiz for their contributions to early versions of our Java bytecode meta-model.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools (2nd edition)*. USA: Addison-Wesley Longman Publishing Co., Inc.

Alanen, M., & Porres, I. (2003). *A relation between context-free grammars and meta object facility metamodels* (Tech. Rep.). Turku Centre for Computer Science.

Arendt, T., Biermann, E., Jurack, S., Krause, C., & Taentzer, G. (2010). Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. models* (pp. 121–135). Springer. doi: 10.1007/978-3-642-16145-2_9

The BCEL homepage. (2020, May). http://commons.apache.org/proper/commons-bcel/. (Accessed: May 15, 2020)

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-driven software engineering in practice* (1st ed.). Morgan & Claypool Publishers.

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014, August). MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology*, *56*(8), 1012-1032. Retrieved from https://hal.inria.fr/hal-00972632 doi: 10.1016/j.infsof.2014.04.007

Bruneton, E., Lenglet, R., & Coupaye, T. (2002). ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, *30*, 19. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769

Desai, N., & Gogolla, M. (2019). Developing comprehensive postconditions through a model transformation chain. *J. Object Technol.*, *18*(3), 5:1–18. Retrieved from https://doi.org/10.5381/jot.2019.18.3.a5 doi: 10.5381/jot.2019.18.3.a5

Efftinge, S., & Völter, M. (2006). oaw xtext: A framework for textual dsls. In *Workshop on modeling symposium at eclipse summit* (Vol. 32).

Eichberg, M., Monperrus, M., Kloppenburg, S., & Mezini, M. (2010). Model-driven engineering of machine executable code. In *Modelling foundations and applications* (pp. 104–115). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-13595-8_10 doi: 10.1007/978-3-642-13595-8_10

Franconi, E., Mosca, A., Oriol, X., Rull, G., & Teniente, E. (2019, August). OCL_{FO}: First-order expressive ocl constraints for efficient integrity checking. *Softw. Syst. Model.*, *18*(4), 2655 - 2678. Retrieved from https://doi.org/10.1007/s10270-018-0688-z doi: 10.1007/s10270-018-0688-z

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A uml-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, *69*(1-3), 27–34. Retrieved from https://doi.org/10.1016/j.scico.2007.01.013 doi: 10.1016/j.scico.2007.01.013

Heidenreich, F., Johannes, J., Seifert, M., & Wende, C. (2009). Closing the gap between modelling and java. In *Software language engineering, second international conference, SLE 2009, denver, co, usa, october 5-6, 2009, revised selected papers* (Vol. 5969, pp. 374–383). Springer. Retrieved from https://doi.org/10.1007/978-3-642-12107-4_25 doi: 10.1007/978-3-642-12107-4_25

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., & Varró, D. (2019, August). Survey and classification of model transformation tools. *Softw. Syst. Model.*, *18*(4), 2361- 2397. Retrieved from https://doi.org/10.1007/s10270-018-0665-6 doi: 10.1007/s10270-018-0665-6

Kolovos, D. S., Paige, R. F., & Polack, F. (2008). The epsilon transformation language. In *Theory and practice of model transformations, first international conference, ICMT 2008, zürich, switzerland, july 1-2, 2008, proceedings* (Vol. 5063, pp. 46–60). Springer. Retrieved from https://doi.org/10.1007/978-3-540-69927-9_4 doi: 10.1007/978-3-540-69927-9_4

Leroy, X. (2003). Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, *30*(3), 235–269. Retrieved from https://doi.org/10.1023/A:1025055424017 doi: 10.1023/A:1025055424017

Li Gong. (1998). Secure java class loading. *IEEE Internet Computing*, *2*(6), 56-61. doi: https://doi.org/10.1109/4236.735987

Liang, S., & Viswanathan, D. (1999). Comprehensive profiling support in the javatm virtual machine. In *Proceedings of the 5th conference on usenix conference on object-oriented technologies and systems - volume 5* (p. 17). USA: USENIX Association. Retrieved from http://www.usenix.org/publications/library/proceedings/coots99/liang.html

Lindholm, T., & Yellin, F. (1999). *Java virtual machine specification* (2nd ed.). USA: Addison-Wesley Longman Publishing Co., Inc.

Lindholm, T., Yellin, F., Bracha, G., Buckley, A., & Smith, D. (2020). *The java virtual machine specification, java se 14 edition.* https://docs.oracle.com/javase/specs/jvms/se14/html/.

Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G. M., … Wimmer, M. (2016, July). Model transformation intents and their properties. *Softw. Syst. Model.*, *15*(3), 647?684. Retrieved from https://doi.org/10.1007/s10270-014-0429-x doi: 10.1007/s10270-014-0429-x

Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., & Tuma, P. (2015). Introduction to dynamic program analysis with disl. *Sci. Comput. Program.*, *98*, 100–115. Retrieved from https://doi.org/10.1016/j.scico.2014.01.003 doi: 10.1016/j.scico.2014.01.003

Nassar, N., Kosiol, J., Arendt, T., & Taentzer, G. (2018). OCL2AC: automatic translation of OCL constraints to graph constraints and application conditions for transformation rules. In *Graph transformation - 11th international conference, ICGT 2018, held as part of STAF 2018, toulouse, france, june 25-26, 2018, proceedings* (Vol. 10887, pp. 171–177). Springer. Retrieved from https://doi.org/10.1007/978-3-319-92991-0_11 doi: 10.1007/978-3-319-92991-0_11

Nassar, N., Kosiol, J., Arendt, T., & Taentzer, G. (2019). Constructing optimized validity-preserving application conditions for graph transformation rules. In *Graph transformation - 12th international conference, ICGT 2019, proceedings* (Vol. 11629, pp. 177–194). Springer. Retrieved from https://doi.org/10.1007/978-3-030-23611-3_11 doi: 10.1007/978-3-030-23611-3_11

Nystrom, N., Clarkson, M. R., & Myers, A. C. (2003). Polyglot: An extensible compiler framework for java. In *Compiler construction* (pp. 138–152). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/3-540-36579-6_11 doi: 10.1007/3-540-36579-6_11

Object Constraint Language, Version 2.4, Object Management Group. (2014). http://www.omg.org/spec/OCL/2.4.

OCLinEcore. (2019). *Eclipse OCL.* Retrieved from https://wiki.eclipse.org/OCL/OCLinEcore

OMG Meta Object Facility (MOF). Version 2.5.1 [Computer software manual]. (2016, 11). Retrieved from http://www.omg.org/spec/MOF/

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., & Seinturier, L. (2016, September). Spoon: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exper.*, *46*(9), 1155?1179. Retrieved from https://doi.org/10.1002/spe.2346 doi: 10.1002/spe.2346

Radke, H., Arendt, T., Becker, J. S., Habel, A., & Taentzer, G. (2018). Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. *Sci. Comput. Program.*, *152*, 38–62. Retrieved from https://doi.org/10.1016/j.scico.2017.08.006 doi: 10.1016/j.scico.2017.08.006

Reynolds, M. C. (2013). Modeling the java bytecode verifier. *Science of Computer Programming*, *78*(3), 327 - 342. doi: https://doi.org/10.1016/j.scico.2011.03.008

Steimann, F., Pawlitzki, T., Apel, S., & Kästner, C. (2010, July). Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, *20*(1). Retrieved from https://doi.org/10.1145/1767751.1767752 doi: 10.1145/1767751.1767752

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *Emf: Eclipse modeling framework* (2nd ed.). Upper Saddle River, NJ: Addison Wesley.

Team, T. B. (Ed.). (2020). *The bytebuddy homepage.* https://bytebuddy.net/#/.

Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., & Ujhelyi, Z. (2016). Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling*, *15*(3), 609–629. Retrieved from https://doi.org/10.1007/s10270-016-0530-4 doi: 10.1007/s10270-016-0530-4

Yildiz, B., Bockisch, C., Rensink, A., & Aksit, M. (2017, 7). A java bytecode metamodel for composable program analyses. In *Software technologies: Applications and foundations* (pp. 30–40). Springer. doi: 10.1007/978-3-319-74730-9_4

Yildiz, B., Rensink, A., Bockisch, C., & Aksit, M. (2017, 4). A model-derivation framework for software analysis. In *Proceedings 2nd workshop on models for formal analysis of real systems (mars)* (pp. 217–229). arXiv.org. doi: 10.4204/EPTCS.244.9

About the authors

Christoph Bockisch is a professor for practical computer science and the chair of the *Programming Languages and Tools* group at the Philipps-Universität Marburg, Germany. His current research focus is on software development tools, program analysis and software engineering methods, with a focal point on software engineering for energy efficiency. You can contact the author at bockisch@acm.org.

Gabriele Taentzer is a professor for software engineering at the Philipps-Universität Marburg, Germany. Her research interests

include the foundation and application of model-based software engineering, software quality assurance, and graph transformation. You can contact the author at taentzer@informatik.uni-marburg.de.

Nebras Nassar is a post-doctoral researcher in the software engineering group at Philipps-Universität Marburg in Germany, where he received the doctoral degree in 2019. His research interests are in model-driven software engineering, quality assurance techniques, and (meta-)modeling tools. You can contact the author at nassarn@informatik.uni-marburg.de.

Lukas Wydra finished his bachelor thesis on the topic *Verification of the well-formedness of Java bytecode with OCL constraints* in the *Programming Languages and Tools* group at the Philips-Universität Marburg, Germany. He received his degree in 2020. You can contact the author at wydral@students.uni-marburg.de.