An MDE Approach for Modular Program Analyses

Bugra M. Yildiz
University of Twente
Enschede, The Netherlands
b.m.yildiz@utwente.nl

Christoph Bockisch
Philipps-Universität Marburg
Marburg, Germany
bockisch@acm.org

Arend Rensink
University of Twente
Enschede, The Netherlands
arend.rensink@utwente.nl

Mehmet Aksit
University of Twente
Enschede, The Netherlands
m.aksit@utwente.nl

ABSTRACT

Program analyses are an important tool to check if a system fulfills its specification. A typical implementation strategy for program analyses is to use an imperative, general-purpose language like Java, and access the program to be analyzed through libraries that offer an API for reading, writing and manipulating intermediate code, such as BCEL or ASM for Java bytecode. We claim that this hampers reuse and interoperability.

In this paper, we propose an Ecore-metamodel for covering Java bytecode completely, which can act as a common basis for program analyses. Code analyses as well as instrumentations can then be defined as model transformations in a declarative language. As a consequence, the implementation of program analysis becomes more concise, more readable and more modular. We demonstrate the effectiveness of this approach by two case studies: profiling of timing performance and model checking of reachability requirements. We also provide tools to generate instances of our bytecode metamodel from Java code in the class file format and vice versa.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Software functional properties*; *Domain specific languages*; Extra-functional properties;

ACM Reference format:
Bugra M. Yildiz, Christoph Bockisch, Arend Rensink, and Mehmet Aksit. 2017. An MDE Approach for Modular Program Analyses. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 5 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079392

1 INTRODUCTION

A typical implementation strategy for program analyses is to use an imperative, general-purpose language like Java, and access the program to be analyzed through libraries that offer an API for managing intermediate code, such as BCEL or ASM for Java bytecode [3, 7, 8, 12, 20]. Considering that many of these analyses involve the same concepts and operate on the same intermediate representation, reusing common steps or chaining them together would save time and effort, and improve interoperability and maintenance. Modern general-purpose languages offer language-level modularity mechanisms, such as libraries and inheritance, but we consider this to be too low-level to effectively support reuse and extensibility of program analysis components; instead, we promote model-based definition of program analyses as a more effective mechanism [4].

The contribution of this paper, therefore, is a complete Ecore-metamodel for Java bytecode, which can be used as a common basis for arbitrary program analyses. Instances of our metamodel can be created from compiled Java code and vice versa. Code analyses can now be defined as model transformations, in one of the well-researched domain-specific languages available for this purpose. Furthermore, analysis results can be represented directly as extensions of the bytecode model of the analyzed program, making the results easily accessible to subsequent manipulation and as input to other tools. We claim that, as a consequence, the implementation of program analyses becomes more concise, better readable and more modular [19]. We have implemented an Eclipse plug-in, called JBCPP [13] to take care of the bytecode-to-model and model-to-bytecode transformations.

We demonstrate the effectiveness of this approach by two example cases that are implemented as model transformations: (a) an analysis of the timing behavior and (b) generation of timed-automata models for model checking of Java programs.

2 JAVA BYTECODE METAMODEL

We have defined the Java bytecode metamodel using the Ecore format provided by the Eclipse Modeling Framework (EMF) [11]. The model transformations in the example cases have been implemented using the Epsilon Transformation Language (ETL), which is one of the domain-specific languages for model management tasks provided by the Epsilon framework [17]. Transformations are implemented in modules consisting of separate rewrite rules (i.e., rules that define how to create a partial target model based on parts of the source model). These modules and the contained rules can be re-used by importing or extending. These features of ETL support reusability and extensibility. Both EMF and the Epsilon framework have large developer communities working with model-driven engineering techniques [6, 18].

The metamodel mainly follows the organization of Java class files as defined in the Java Virtual Machine specification [16]. In general,

Programming '17, Brussels, Belgium
© 2017 ACM. 978-1-4503-4836-2/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079392

each kind of entity from the class file format (like method declarations, attributes or instructions) is represented as one Ecore class in the metamodel. Lexical nesting (e.g., a method is nested inside its declaring class) is represented as a containment relationship in the metamodel (in terms of the previous example: the model object of a method is contained in the model object of the class that declares it). All containment relationships are navigable bidirectionally.

The most relevant elements of the metamodel are shown in textual form in Listing 1 and described below. Entities not relevant for our case studies (e.g., fields, local variables, etc.) are omitted.

```
1  package jbcmm : jbcmm = 'http://nl.utwente.fmt/jbcmm' {
2    class Project {
3      property classes#parent : Clazz[+] { ordered composes };
4      property mainClass : Clazz;
5    }
6    class AttributeOwner {
7      property attributes#parent: Attribute[?];
8    }
9    class Clazz extends AttributeOwner {
10       attribute majorVersion : ecore::EInt;
11     property modifiers : Modifier[*] { ordered composes };
12     property superclass : Clazz;
13     property subinterfaces : Clazz[*] { ordered };
14     property subclasses : Clazz[*] { ordered };
15     attribute name : String;
16     property parent#classes : Project[?];
17     property methods#parent : Method[*] { ordered composes };
18   }
19   class Method extends AttributeOwner {
20     property parent#methods : Clazz[?];
21     property modifiers : Modifier[*] { ordered composes };
22     attribute name : String;
23     attribute descriptor : String;
24     property instructions#parent : Instruction[*] { ordered
            composes };
25     property firstInstruction : Instruction[?];
26   }
27   class Field extends AttributeOwner { ... }
28   abstract class ControlFlowEdge {
29     property start#outEdges : Instruction;
30     property end#inEdges : Instruction;
31   }
32   class UnconditionalEdge extends ControlFlowEdge;
33   class ConditionalEdge extends ControlFlowEdge {
34     attribute condition : Boolean;
35   }
36   class ExceptionalEdge extends ControlFlowEdge {
37     property exceptionTableEntry : ExceptionTableEntry[1];
38   }
39   class SwitchCaseEdge extends ControlFlowEdge {
40     attribute condition : ecore::EInt;
41   }
42   class SwitchDefaultEdge extends ControlFlowEdge;
43   abstract class Instruction {
44     property parent#instructions : Method[?];
45     attribute linenumber : ecore::EInt[?];
46     attribute index : ecore::EInt;
47     attribute opcode : String;
48     property outEdges#start : ControlFlowEdge[*] { ordered
            composes };
49     property inEdges#end : ControlFlowEdge[*] { ordered };
50   }
51   abstract class MethodInstruction extends Instruction {
52     attribute owner : String;
53     attribute name : String;
54     attribute desc : String;
55   }
56   class InvokevirtualInstruction extends MethodInstruction;
57   class InvokespecialInstruction extends MethodInstruction;
58   ... }
```

Listing 1: An excerpt of the bytecode metamodel

- `Project` is the root of the Ecore model containing all classes and a dedicated reference to the designated main class.
- `Class` (`Clazz` in Listing 1) represents a class or an interface. It contains zero or more `Methods`.
- `Method` represents a method in a class, specifying the signature of the method. If the method has an implementation, it has a reference to its first instruction. The method also has an unordered set of all contained instructions
- `Instruction` is an abstract entity that serves as the top entity in the bytecode instruction hierarchy, and contains the common properties of all instruction types. An instruction contains zero or more `Control Flow Edges` and a reference to the instruction following it in the bytecode (`nextInCodeOrder`). This reference is currently necessary for technical reasons; in the future, we will remove this necessity.

  The subclasses of `Instruction` form a hierarchy organized according to shared semantics and structure of instructions. An example by means of method call instructions is shown in Listing 1 (lines between 51 and 57). In this example, the abstract `MethodInstruction` entity captures the common properties of all method call instructions: the object type on which the method is invoked (`owner`), and the signature of the called method (`name` and `desc`). It is extended by the instantiable method call instructions, represented by the `Invoke...Instruction` entities.
- The control flow information, which is implicitly available in the class file through the ordering of instructions or targets of jump instructions, is explicitly stored as a property of an instruction and is presented in our metamodel via `Control Flow Edges` between instructions. For most instruction types, there is exactly one outgoing edge (represented as an `UnconditionalEdge`), corresponding to the successor instruction or to the single jump target for a `goto` instruction. For branching or switch instructions, the outgoing edges (`ConditionalEdge`) represent the targets of the conditional jumps respectively a reference to the instruction to be executed in the default case.

  Instructions that are within the range of an exception handler, additionally have an outgoing `ExceptionalEdge`. This has an `ExceptionTableEntry` property (not shown in the listing), which holds information about the type of exceptions handled, the handler's scope as well as the first instruction of the handler.

2.1 Bytecode-to-Model and Model-to-Bytecode Transformation

To conveniently create instances of our Java bytecode metamodel from existing code, we have developed an Eclipse plug-in, called Java Bytecode++ (JBCPP) plug-in [13]. To use this plug-in, a class within the build path of an Eclipse Java project can be chosen, e.g., in the Package Explorer View. When the JBC++ button in the tool bar is pressed, a model is created for the project. For this purpose, it is expected that the selected class defines a main method, which acts as the entry point of the project.

For the set of all classes that comprise the project under analysis, we create instances of our Java bytecode metamodel entities. This number is typically excessively large because all classes in the standard class library or third-party libraries are included. However, typically, we want to focus analyses on the code developed in the project itself, instead of re-used code. We therefore limit the classes included in the model by excluding unreachable classes and classes defined in libraries. This is, nevertheless, no restriction imposed by our implementation but merely an optimization attempt.

Although there are several frameworks for parsing textual representations of metamodels and generating model instances in this way (as well as for generating textual representations from metamodel instances), the existing frameworks are not applicable for handling bytecode directly. One reason is that the tokens in the bytecode are sequences of bytes rather than characters. But what is more, for efficient access to bytecode in memory, some instructions have different forms depending on their position in the bytecode. Therefore, bytecode does not belong to a grammar class supported by existing frameworks; and even if it did, the grammar definition would consist of non-printable characters for terminals. Because of these reasons, we use the ASM bytecode manipulation toolkit [5] to parse class files and instantiate model elements according to the definitions found in there. In the end, the created model is stored in the standard XMI format. This can subsequently be processed by standard EMF-based tooling.

The generation of class files from our models works analogously. The model is traversed in the order in which elements appear in the Java bytecode format and the ASM toolkit is used to create definitions in this format. All generated class files are stored in a specific subfolder (organized according to the package structure) of the Eclipse plug-in, which can be added to the classpath to execute the bytecode generated (possibly after transformation).

To test our bytecode-to-model and the model-to-bytecode transformation, we chain both transformations: First we generate a model from a bytecode file and then we generate a bytecode file from this model. If both files are identical, we can be confident that our implementation is correct, unless we have made mistakes in both transformations that compensate each other by chance. In our case, we do not get identical files, but semantically equivalent ones. The reason is that for some entities the order is not semantically important, and for such cases, we do not preserve the order in the bytecode. For example, local variables have an index in the bytecode (rather than a name), and if indexes are re-numbered consistently (which happens in our transformations) the result is equivalent but not identical. Therefore, in our tests, we have to manually compare the original and the result; in all our tests, we did not find any discrepancy.

We have not yet spent effort on optimizing our implementation, but we nevertheless have already performed some benchmarks, see Table 1. The table presents for three real-world programs: the number of classes in their implementation, the number of implemented methods, the number of elements in the extracted model and the time (in seconds) required for extracting the model.

The results show that bytecode model extraction can take some time, for example almost 25 minutes for extracting the model of the Groove program, which is also the largest (in terms of number of classes or methods) program used in our benchmark. While 25 minutes is a long time, it should be noted that the programs are large with up to almost 1500 classes, which are all represented in the extracted models. Therefore, these benchmarks demonstrate the scalability of our approach. Besides, we expect that optimization opportunities exist.

Program	Classes	Methods	Model Size	Time
LiveGraph	131	350	24,049	18s
Groove	1,482	9,232	418,269	1,480s
Weka	1,041	8,322	756,063	764s

Table 1: Preliminary benchmarking results of Java bytecode model extraction.

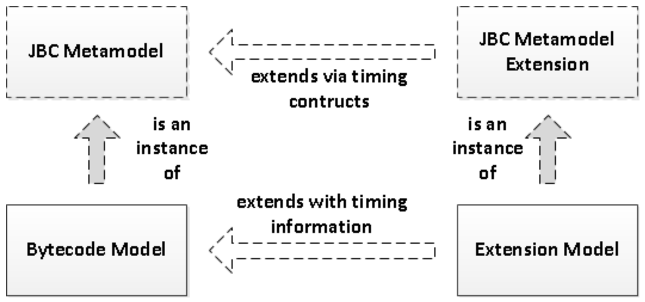

Figure 1: Metamodel extension to represent timing information

3 EXAMPLES

In this section, we demonstrate two example cases to show the effectiveness of working with our metamodel. The first example case, described in subsection 3.1, profiles the timing performance of Java programs in an iterative manner via instrumentation and represents the timing values as a model that conforms to an extension to our bytecode metamodel. The second example case, described in subsection 3.2, generates timed-automata models from instances of our bytecode metamodel for model checking.

3.1 Example 1: Profiling Timing Performance

Profiling via automatic instrumentation is a frequently used technique [9, 14, 15]. In this example case, we present an example of automatic instrumentation of the bytecode for profiling timing performance using our bytecode metamodel. Having a bytecode model at hand offers the advantages of writing the instrumentation code as a model transformation using a domain-specific language designed for this task, and utilizing the systematic extension mechanism of the metamodeling technology to represent the generated profiling information at the structure of the bytecode model, such that it can easily be used in other analyses such as average timing behavior.

Figure 1 shows how we have used the extension mechanism of Ecore to represent the timing information. `JBC Metamodel Extension` is an extension to our bytecode metamodel, which keeps the elements related to timing information and their reference to the `Instruction` elements in the bytecode metamodel. Assuming

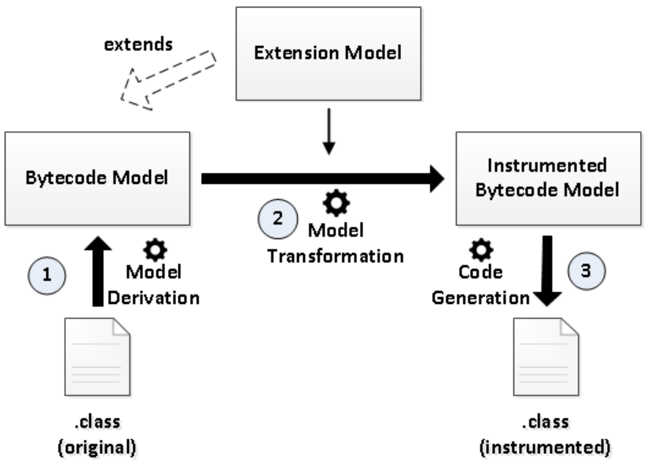

Figure 2: Bytecode instrumentation via model transformation

that a bytecode model (which is an instance of the bytecode metamodel) is available, the timing information for this model can be attached (without modifying the model itself) by creating an extension model (which is an instance of the metamodel extension) and setting necessary references to the original bytecode model.

The implementation for this example can be divided into 5 steps. The first three steps, shown in Figure 2, correspond to the instrumentation of the class file:

In *Step 1*, the original bytecode model, which is an instance of the metamodel, is automatically derived from a project's class files using the JBCPP plug-in.

In *Step 2*, a new bytecode model with instrumentation is generated via the model transformation. The transformation inserts the static calls

`startProfiling(instructionId: String)`
`stopProfiling(instructionId: String)`
`finishProfiling()`

to a profiling library that we have implemented for this example. The instrumentation considers the timing performance ofdiptio method calls, whereby the passed `instructionID` is a textual reference to the `Instruction` object in the Java bytecode model which was instrumented; this ID will later be used to relate the profile data to the model.

First, the model transformation checks if any timing information is available in the extension model. If no timing information is available, only the main method of the project is instrumented. The transformation rule that conducts this task is given in Listing 2. The `@greedy` tag forces this rule to be applied to all method instruction subtypes. The rule starts with the rule name, `InstrumentMethodCallsInMain`. The `transform` keyword defines from which input elements to which output elements this rule does the mapping. For this case, for each method instruction `mi` in `JBCModel`, the static method call instructions `startProfilingCall` and `stopProfilingCall` are created in `InstrumentedJBCModel`. The `guard` keyword defines under which condition this rule will be executed; i.e., that there is no timing information available in the extension model and that `mi` is contained in the main method. The body of the rule (omitted here for improving readability) sets up instruction objects, corresponding to the described method calls, and inserts them into the target model.

Second, if there already exists timing information in the form of an extension model, only the implementations of the method calls whose execution time are over a threshold (specified as a parameter of the transformation) are instrumented. The idea of refining the instrumentation scope for profiling is inspired from the work in [1].

In *Step 3*, the JBCPP plug-in creates the (instrumented) class files from the instrumented bytecode model.

```
@greedy
rule InstrumentMethodCallsInMain transform mi:JBCModel!
      MethodInstruction to startProfilingCall:InstrumentedJBCModel
      !InvokestaticInstruction, stopProfilingCall:
      InstrumentedJBCModel!InvokestaticInstruction
{
  guard: ExtensionModel!Timing.allInstances().size() = 0 and mi.
        parent.name = "main" and mi.parent.descriptor = "([Ljava/
        lang/String;)V"

  ...
}
```

Listing 2: Example transformation rule

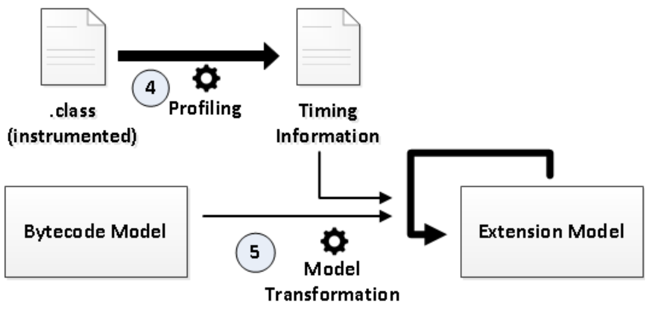

Figure 3: Attaching profiling information to original model

The remaining two steps, shown in Figure 3, correspond to the run-time profiling and representing of the observed timing information as an extension model:

In *Step 4*, the timing information is generated by executing the instrumented classes. The generated timing information is also a model conforming to a simple metamodel since using a model as an input to a model transformation is much more practical than generating a text-based file and parsing it later.

In *Step 5*, the generated timing information is again expressed as an extension model via model transformation. The transformation actually updates the existing extension model: It keeps the existing timing information unchanged and instantiates new timing information entities and sets their references to related instructions in the original bytecode model.

3.2 Example 2: Model-Derivation Framework for Model Checking

One of the analysis approaches for checking if a system fulfills its specification is model checking. Model checkers exhaustively check whether a model of a system meets required properties.

In our previous work, we have proposed a framework to derive timed-automata models for model checking purposes from instances of an earlier version of the bytecode metamodel [21]. The framework transforms the bytecode models to extended models in order to handle recursion and to enrich them with loop and timing information. All these steps are implemented via model transformations. At the end of the process, the framework produces timed-automata models compatible with the UPPAAL [2] model checker.

4 RELATED WORK

There are not many attempts in the field of metamodeling of bytecode. Eichberg et al. [10] provide an XML Schema-based metamodel of bytecode supporting multiple instruction set architectures, such as Java bytecode. They report the benefits of using an explicit metamodel: ease of changing and extending a metamodel in case of new requirements, and facilitating the development of generic analyses with the help of a well-defined data structure. Whereas their approach focuses on supporting static analyses only, our approach is capable of supporting dynamic analyses as well, including the possibility to represent (dynamic) analysis results directly as extension of the analyzed code's model. Furthermore, with the EMF framework and its Ecore facility to define our metamodel, we use a technology specifically designed for metamodeling and model transformation purposes. Since we use a standard framework for metamodeling purpose, we also benefit from a large number of available domain-specific languages, such as ETL, designed for model management tasks to implement analyses.

5 CONCLUSION

In this paper, we have presented our Java bytecode metamodel and the current state of our implementation of the JBCPP plug-in to be used for the bytecode-to-model and model-to-bytecode transformations. The metamodel allows code analyses to be written as model transformations in a semi-declarative, domain-specific language. In this way, the implementations of program analyses become shorter, more readable and more modular in general.

We have demonstrated the effectiveness of our approach with two examples. In addition to the mentioned benefits in the previous paragraph, these examples have shown that the metamodeling approach offers a systematic extension mechanism to represent the information needed for analyses without modifying the metamodel.

As future work, we will re-implement several published static and dynamic analyses in our approach and compare this to their original implementations. We will do so by implementing building blocks of these analyses as modular model transformations. We expect to deliver a library of reusable implementations of tasks required by existing program analyses. Furthermore, we are planning to update the framework we presented in [21] with the new version of the metamodel and integrate the timing profiling example with the framework.

REFERENCES

[1] D. Ansaloni. Self-refining aspects for dynamic program analysis. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development Companion*, AOSD '11, pages 75–76, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0606-5. doi: 10.1145/1960314.1960342. URL http://doi.acm.org/10.1145/1960314.1960342.

[2] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a tool suite for automatic verification of real–time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[3] W. Binder, J. Hulaas, and P. Moret. Reengineering standard Java runtime systems through dynamic bytecode instrumentation. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 91–100, Sept 2007. doi: 10.1109/SCAM.2007.20.

[4] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):7:1–28, Apr. 2012. ISSN 1660-1769. doi: 10.5381/jot.2012.11.1.a7.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30: 19, 2002.

[6] F. Budinsky. *Eclipse Modeling Framework: a developer's guide.* Addison-Wesley Professional, 2004.

[7] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 27–40 vol.2, 2001. doi: 10.1109/DISCEX.2001.932157.

[8] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30 (12):859–872, Dec 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.91.

[9] J. Dongarra, A. D. Malony, S. Moore, P. Mucci, and S. Shende. Performance instrumentation and measurement for terascale systems. In *Computational ScienceâĂŤICCS 2003*, pages 53–62. Springer, 2003.

[10] M. Eichberg, M. Monperrus, S. Kloppenburg, and M. Mezini. Model-driven engineering of machine executable code. In *European Conference on Modelling Foundations and Applications*, pages 104–115. Springer, 2010.

[11] EMF. Eclipse Modeling Framework website. https://www.eclipse.org/modeling/emf/, Apr. 2016.

[12] A. Q. Gates, O. Mondragon, M. Payne, and S. Roach. Instrumentation of intermediate code for runtime verification. In *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, pages 66–71, Dec 2003. doi: 10.1109/SEW.2003.1270727.

[13] JBCPP. Java bytecode metamodel and jbc++ plug-in website. https://bitbucket.org/bmyildiz/java-bytecode-metamodel-repository, June 2016.

[14] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 49–49, Nov 2000. doi: 10.1109/SC.2000.10052.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034.

[16] Oracle. Java class file format, website. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html, May 2016.

[17] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE Int. Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 162–171, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3702-3. doi: 10.1109/ICECCS.2009.14. URL http://dx.doi.org/10.1109/ICECCS.2009.14.

[18] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[19] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[20] C. Wang, X. Mao, Z. Dai, and Y. Lei. Research on automatic instrumentation for bytecode testing and debugging. In *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, volume 1, pages 268–274, May 2012. doi: 10.1109/CSAE.2012.6272595.

[21] B. M. Yildiz, A. Rensink, C. M. Bockisch, and M. Akşit. A model-derivation framework for timing analysis of Java software systems. Technical Report TR-CTIT-15-08, Centre for Telematics and Information Technology, University of Twente, Enschede, December 2015.