A Java Bytecode Metamodel for Composable Program Analyses

Bugra M. Yildiz[1], Christoph Bockisch[2](✉), Arend Rensink[1], and Mehmet Aksit[1]

[1] University of Twente, Enschede, The Netherlands
{b.m.yildiz,arend.rensink,m.aksit}@utwente.nl
[2] Philipps-Universität Marburg, Marburg, Germany
bockisch@mathematik.uni-marburg.de

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 for manipulating intermediate code, such as ASM for Java bytecode. We show that this hampers composability, interoperability and reuse of analysis implementations.

We propose a complete Ecore-metamodel for Java bytecode as a common basis for program analysis implementations, as well as an Eclipse plug-in to create bytecode metamodel instances from Java bytecode and vice versa. Code analyses can be defined as model transformations in a declarative, domain-specific language. As a consequence, the implementations of program analyses become more composable and more modular in general. We demonstrate the effectiveness of this approach with a case study.

Keywords: Java bytecode · Metamodel · Model transformation
Model-driven software engineering · Program analyses · Composition

1 Introduction

Program analyses are developed for, e.g., checking correctness, performance or real-time requirements. In general, such analyses either determine statically accessible properties, or they modify the code such that information is collected at execution time. Thus, analyses are either static or dynamic, or hybrid analyses combining both [14,18,19]. Much of the research in program analysis targets the Java language; more precisely, it should be said that the bytecode format of the Java Virtual Machine is targeted, as analyses usually inspect and instrument this intermediate representation rather than the source code. This has several advantages; for example, many different source languages compile to the Java bytecode format, thus, multiple languages can be supported at once. Furthermore, typically, the bytecode is available for the whole program, including third-party libraries. A typical implementation strategy for bytecode-level program analyses is to use an imperative, general-purpose language like Java, and to access the program to be analyzed through libraries that offer an API for inspecting or instrumenting intermediate code, such as BCEL or ASM [5,9,11,15,22].

New analyses often *conceptually* extend or combine existing ones by optimizing the information collection or collecting additional information. Considering that many of these analyses involve similar concepts and work on the same intermediate representation, being able to extend and compose existing analysis implementations would save time and effort, and improve interoperability and maintenance. Though Java (and other general-purpose languages) offers language-level modularity mechanisms, such as class libraries and inheritance, they are not sufficient to implement program analyses in a composable fashion. To compose two or more analyses (or parts thereof), the only possibility is to apply one analysis to the output of the previous one. However, since an analysis in general alters the bytecode, the subsequent ones do not see the original code, which may invalidate their results. In this paper, we promote model-based definitions of program analyses as a more flexible mechanism [6].

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 in the class file format, 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 them easily accessible to subsequent manipulation and to other tools. We claim that, as a consequence, the implementation of a program analysis becomes more composable and modular [21]. We have implemented an Eclipse plug-in, called *JBCPP* for the bytecode-to-model and model-to-bytecode transformations.[1]

We demonstrate the effectiveness of this approach with a motivating example comparing the composability of two program analyses in the traditional (using a general-purpose programming language) and in our implementation approach (using a model transformation language utilizing our metamodel).

2 Motivation

While there are metamodels available for high-level programming languages (e.g., JaMoPP [17], MoDisco [7] and domain-specific languages developed with tools like xText [13] or EMFText [16]), we are not aware of any program analyses developed in a model-driven way using these source-level metamodels. There are several reasons why program analyses are typically implemented based on bytecode rather than source code:

– The bytecode is always available for the whole program, also for the third-party components and libraries that are not available in source code.

[1] The plug-in and the metamodel are available on the JBCPP homepage: https://bitbucket.org/bmyildiz/java-bytecode-metamodel-repository.

– Many implicit features in the source code are resolved and represented explicitly in bytecode. Some examples are simple type names, which can only be fully qualified by interpreting `import` statements, or the implicit default constructors.
– Java bytecode is the compilation target for various languages. Therefore, implementing an analysis for Java bytecode generally makes it applicable to programs written in these different source languages.
– A single statement in the source language is typically represented by multiple finer-grained bytecode instructions, which makes Java bytecode more flexible. For example, control flow is not limited to properly nested blocks. Therefore code instrumentations often are easier to be defined at the bytecode level.
– Besides, at least for Java programs, almost no information from the source level gets lost when compiling to bytecode. A notable exception are scopes confined to blocks in the source code, e.g., for a `for` loop, the sections initialization, condition, increment and body are not explicitly represented but can typically be recognized by simple and local analyses [23].

2.1 Motivating Example

We will express our problem statement by employing two explanatory program analyses. The first analysis is to count how often each method call in the program is executed. To do so, this analysis instruments each invocation instruction in the bytecode by inserting a call to the method `InvocationCount.increase()`. To identify the instruction whose executions are counted, the analysis numbers all invocation instructions in a method and generates an identifier based on the fully qualified method name, that contains the invocation instruction and the instruction's number. This identifier is passed to the `increase()` method as an argument. After the program execution, the results are written to a file. We call this analysis *invocation count*.

The second analysis measures the time that elapses for each method invocation. This analysis prepends each method invocation with code to store the result of `System.currentTimeMillis()` in a local variable. Then, it appends code to calculate the difference of the current time and the stored start time and to pass the result to the method `Time.increase()`. This method receives a unique identifier of the invocation instruction, which is computed in the same way as for the invocation count analysis. The method `Time.increase()` stores the accumulated elapsed time per invocation instruction, which again is dumped at the termination of the execution. We call this analysis *time* in short.

Neither analysis instruments invocations that occur in its respective `increase()` method or invocations of methods from the system class library, i.e., classes in a subpackage of `java`, to avoid endless recursions.

2.2 Implementing Program Analyses with a Bytecode Toolkit

Several toolkits for reading and manipulating Java bytecode are available [3,8,10]. These toolkits basically support two styles for implementing program analyses: First, the bytecode is transformed to an object-oriented representation and analyses subsequently process this representation, possibly altering it; this facilitates random access to elements in the bytecode. Second, the bytecode is traversed in one pass during which a visitor, implementing the analysis, reacts to encountering relevant elements; in this style the bytecode is naturally traversed sequentially. If a such-implemented analysis employs code instrumentation, each encountered element is, by default, copied to the output, unless the analysis decides to suppress or modify a visited element or insert something at its location.

Both styles have the following points in common: (1) By default, there is no way to combine multiple analyses other than to perform them sequentially; and (2) it is not possible identify which elements in the resulting bytecode stem from the original input or are inserted by an analysis.

In our examples, we employ the ASM bytecode toolkit and make use of the visitor style since this is currently the most common implementation approach for program analyses based on Java bytecode.

Listing 1.1 shows the visitor methods handling the encounter of a method invocation of *invocation count*. What is not shown in the listing is that the visitor does not descend into the methods `InvocationCount.increase()` but copies them verbatim, such that the method invocation instructions within this methods are not visited. The visitor method of *time* is implemented analogously.

```
1  @Override
2  public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean
        itf) {
3    if (! owner.startsWith("java/")) {
4        String id = getInstructionID();
5        super.visitLdcInsn(id);
6        super.visitMethodInsn(Opcodes.INVOKESTATIC, "ic_analysis/InvocationCount",
              "increase", "(Ljava/lang/String;)V", false);
7    }
8    super.visitMethodInsn(opcode, owner, name, desc, itf);
9  }
```

Listing 1.1. Instrumenting method invocations for the invoction count analysis with the ASM toolkit.

As the listing shows, the instrumentation code is inserted into the output by calling the respective `super.visitXXX()` methods. This is the case for original instructions occurring in the input (e.g., line 8 in Listing 1.1) as well as the additional code. The invocations to `getInstructionID()` (e.g., line 4 in Listing 1.1) return the unique identifier of the method call instruction, as described in the previous subsection.

2.3 Composing the Toolkit-Based Analyses

The two described analyses measure the total execution time of each method call as well as an execution count for each method call. Thus, composing both analyses would allow to compute the average execution time for each method call.

The typical approach of combining two analyses implemented in an approach such as outlined above is to apply them sequentially. This is, analysis 1 is applied to the original bytecode yielding intermediate bytecode as result. Second, analysis 2 is applied to this intermediate code yielding the final bytecode.

In the case of our example, both analysis instrument all method invocations: This causes the intermediate code produced by the first instrumentation to contain additional method invocations, which are then unintentionally instrumented by the second analysis. As a result, depending on the application order of the two instrumentations, either method invocations are counted or timed, which were not part of the original program.

Furthermore, the identifier computed for each method call (`getInstructionID`) is based on the position of the instruction in the bytecode, which changes because of the instrumentations. Therefore, the identifiers of both analyses do not match. Only the identifiers of the first analysis can be mapped to the original bytecode.

One might think that analyses implemented in the outlined approach could also be composed by inheritance. However, this does not solve the problem, as the visitor method (`visitMethodInsn`), which is implemented to react to the encounter of a method call, is also called to insert additional method calls.

2.4 Problem Statement

The current state of the art bytecode manipulation toolkits follow approaches that do not support composition of independently developed program analyses. While it could be possible to employ specific patterns for extensibility and composability when implementing a program analysis, we are not aware of such approaches, let alone bytecode manipulation toolkits supporting this.

For this reason, we suggest a new approach, implementing program analyses in a model-driven way. Model transformation approaches from the model-driven engineering (MDE) world have a strong focus of declarative definitions, composability and extensibility, which is why we think that the ability to implement program analyses as model transformations is a significant added value.

3 Java Bytecode Metamodel

To facilitate implementing program analyses in a model-driven way, we have developed a metamodel (using the Ecore format provided by the Eclipse Modeling Framework (EMF) [1]) of Java bytecode, called *JBCMM*[2]. Bytecode models, instances of the metamodel, act as a basis for analyses. Furthermore, the metamodel can be extended to meet various concerns such as the representation of analysis results. In our metamodel, all relevant elements are uniquely identifiable. For example, classes have a unique name (made up of the package name plus the simple class name), and the name and descriptor of a method is unique within

[2] Published on the JBCPP homepage: https://bitbucket.org/bmyildiz/java-bytecode-metamodel-repository.

a class. In addition to the names used in the bytecode specification, additional elements like instructions have names in our metamodel to make them identifiable. These names have to be unique within their scope, e.g. the fully qualified identifier of an instruction is composed of the instruction name, the containing method's name and descriptor and the declaring class's name.

Therefore, information generated by an analysis can be uniquely associated with model elements by using these fully qualified identifiers, staying valid independently of which modifications are applied to the model. Java bytecode (thus also our metamodel) accommodates for storing a mapping between bytecode and the source line from which it was compiled. Therefore, it is also possible to trace each object in a JBCMM model to the corresponding source line.

3.1 Structure of Java Bytecode Metamodel

The metamodel mainly follows the organization of Java class files as defined in the Java Virtual Machine specification [2]. In general, 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: a method is contained in the class that declares it). To simplify implementation of analyses, all containment relationships are navigable bidirectionally.

The most relevant elements of the metamodel are shown in graphical form in Fig. 1 and described below. Entities not relevant for our case studies (e.g., fields) are omitted here, but are treated analogously.

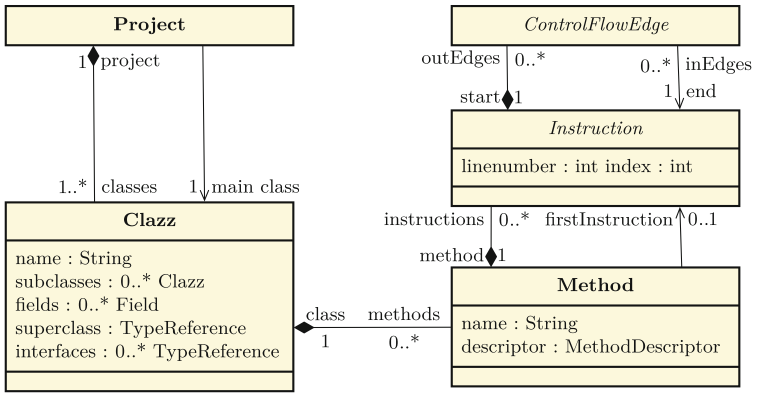

Fig. 1. A view from the bytecode metamodel

`Project`, the root of the Ecore model, contains all classes and refers to the designated main class. `Clazz` represents a class or an interface, storing type hierarchy information, type-level declarations such as annotations or modifiers, and all the members declared inside the class. `Method` stores method-level declarations and (for non-abstract methods) contains the method's instructions and a reference to the first instruction.

`Instruction` is an abstract entity that build the root of a hierarchy of bytecode instruction entities, containing the properties shared all instruction types. Also, an instruction contains zero or more `ControlFlowEdges`. The subclasses of `Instruction` form a hierarchy organized according to shared semantics and, thus, also shared structure of instructions. For example, all instructions for invoking methods are represented as the subtype `MethodInstruction`, which is further extended by types for the specific kinds of invocation such as `InvokestaticInstruction`.

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 a hierarchy of `ControlFlowEdges` between instructions. Concrete types of edges are unconditional, different types of conditional or exceptional control flow edges.

3.2 JBCPP Plug-in

To conveniently create instances of our Java bytecode metamodel from existing code, we have developed an Eclipse plug-in, called *Java Bytecode++ (JBCPP)*.

In [24] we have evaluated the performance of JBCPP when processing projects at different scales. The largest model derived from a Java program with over 1,000 classes consisted of over 750,000 objects. The generation of the model took almost 90 min. This shows that our approach is feasible at least for, e.g., nightly analysis runs, but performance improvements are needed.

```
pattern InstrumentInvocationCountIncrease
     thisMethodInstruction:InOutJBCModel!MethodInstruction {
  match: thisMethodInstruction.isPatternApplicable()

  do {
    var newLcdInstruction = thisMethodInstruction.getNewLcdStringInstruction(
          thisMethodInstruction.uuid );
    var newInvokeStaticInstruction =
          thisMethodInstruction.getNewInvokeStaticInstruction();
    var parameterList = new OrderedSet();
    parameterList.add("Ljava/lang/String;");
    newInvokeStaticInstruction.methodReference = getMethodReference( "increase",
          "Lic_analysis/InvocationCount;", "V" , parameterList);
    createNewUnconditionalEdge(newLcdInstruction, newInvokeStaticInstruction);
    insertBefore (newInvokeStaticInstruction, thisMethodInstruction);
  }
}
```

Listing 1.2. Implementation of the invocation count analysis as a model transformation.

We have used the Epsilon Pattern Language (EPL) to implement these transformations, which is one of the domain-specific languages for model management tasks provided by the Epsilon language family [20].

In EPL, the transformation actions are defined in terms of *patterns*, which in the first place filter by the type of model objects to which they are applicable. Second, the `match` part can filter based on the properties and attributes of the selected model object. When both the type-based and the property-based filtering selects a model object, the transformation specified in the `do` part is executed.

For our example, the pattern is defined on `MethodInstruction` instances, represented by `thisMethodInstruction`. The guard in line 4 checks if `thisMethodInstruction` is not a call to a method of classes in the system library and that the call does not appear inside the analysis' `increase()` method. After the matching, the `do` part starts. From line 6 to line 10, the two bytecode instructions that will be inserted are generated. In line 11, these bytecode instructions are connected with a control flow edge. Finally, in line 12, the newly created instructions are inserted before `thisMethodInstruction` via the `insertBefore` operation. This operation redirects any incoming edges of `thisMethodInstruction` to the first instruction of instrumentation, and creates a new control flow edge between the last instruction of instrumentation and `thisMethodInstruction`.

Patterns implemented independently in different EPL modules can be easily composed: A new transformation can be implemented that imports both the module for the invocation count and the module for the time analysis. Then, both modules will be applied to the same input model at once, yielding one output model that has the extra (instrumented) instructions added by both analyses for all method call instructions present in the input model – and only for these instructions. Since both analyses use the unique identifiers of call-instruction objects in the input model, the data produced by both analyses can be easily mapped back to the original method calls.

4 Related Work

There are not many attempts in the field of metamodeling of bytecode. Eichberg et al. [12] 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. A similar approach, however working at the level of source code is MoDisco [7]. Their approach is to derive a language-independent model from source code, which then acts as store for analysis results. Like the work of Eichberg et al., they do not facilitate code instrumentation using the model.

Heidenreich et al. [17] propose an Ecore metamodel for the Java source code language, including a parser to create instances of this metamodel from Java code and Eclipse plug-ins to create Java source code from the instances of this metamodel. We can use JaMoPP to investigate our claim that implementing (hybrid) analyses as transformations of a bytecode metamodel is more suitable than using a source code metamodel.

5 Conclusion

In this paper, we have presented our complete Java bytecode metamodel and 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 well-studied domain-specific languages. In this way, program analyses become more composable and the results of these analyses can be associated with the entities in bytecode via unique identifiers, which have been demonstrated with the motivating example.

The scalability of this approach has not yet been evaluated systematically, but we have made an initial assessment. For four realistically sized programs we derived byteceode models, transformed them and converted them back to bytecode (also cf. [24]). The relevant sizes and times are shown in Table below. The data already shows that our approach is feasible even of realistic programs. We expect significant performance gains through better engineering of the derivation and bytecode generation. By incrementalizing our approach we believe that the performance can reach a sufficient level for use in practice.

	Model size					Duration [seconds]			
	Classes	Methods	Instr.	Flow edges	Total	Deriv.	Trans.	To bc	Total
LiveGraph	131	350	11, 795	11, 740	24, 016	18	51	35	104
Groove Gen.	930	5, 392	99, 738	98, 634	204, 694	1, 414	86	364	1, 864
Groove Sim.	1, 482	9, 232	203, 030	203, 071	416, 815	1, 480	300	977	2, 757
Weka	1, 041	8, 322	367, 774	374, 854	751, 991	764	803	2, 402	3, 969

As future work, we will re-implement several published static and dynamic analyses in our approach and compare them to their original implementations. One result of this exercise will be the provision of a library of reusable and composable fine-grained building blocks of program analyses.

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 [24]. 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 [4] model checker. We will update this framework to the most recent version of our Java bytecode metamodel.

References

1. Eclipse Modeling Framework (2016). https://www.eclipse.org/modeling/emf/
2. Java Class File Format, May 2016. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html
3. BCEL, June 2016. https://commons.apache.org/proper/commons-bcel/

4. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
5. Binder, W., Hulaas, J., Moret, P.: Reengineering standard Java runtime systems through dynamic bytecode instrumentation. In: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 91–100, September 2007
6. Bockisch, C., Sewe, A., Yin, H., Mezini, M., Aksit, M.: An in-depth look at ALIA4J. J. Object Technol. **11**(1), 7:1–7:28 (2012)
7. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. IST **56**(8), 1012–1032 (2014)
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems (2002). http://asm.ow2.org/current/asm-eng.pdf
9. Chander, A., Mitchell, J.C., Shin, I.: Mobile code security by Java bytecode instrumentation. In: Proceedings of DARPA Information Survivability Conference and Exposition II, DISCEX 2001, vol. 2, pp. 27–40 (2001)
10. Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45102-1_16
11. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE TSE **30**(12), 859–872 (2004)
12. Eichberg, M., Monperrus, M., Kloppenburg, S., Mezini, M.: Model-driven engineering of machine executable code. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 104–115. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13595-8_10
13. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of OOPSLA, pp. 307–309. ACM (2010)
14. Fairley, R.E.: Tutorial: static analysis and dynamic testing of computer software. Computer **11**(4), 14–23 (1978)
15. Gates, A.Q., Mondragon, O., Payne, M., Roach, S.: Instrumentation of intermediate code for runtime verification. In: Proceedings of 28th Annual NASA Goddard Software Engineering Workshop, pp. 66–71, December 2003
16. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with EMFText. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 322–345. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35992-7_9
17. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the gap between modelling and Java. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 374–383. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_25
18. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A tool framework for static and dynamic analysis of object-oriented software with templates. In: ACM/IEEE SC, p. 49, November 2000
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (2005)
20. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Proceedings of International Conference on Engineering of

Complex Computer Systems, pp. 162–171. IEEE Computer Society (2009). http://dx.doi.org/10.1109/ICECCS.2009.14
21. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. Sigplan Not. **35**(6), 26–36 (2000)
22. Wang, C., Mao, X., Dai, Z., Lei, Y.: Research on automatic instrumentation for bytecode testing and debugging. In: IEEE International Conference on Computer Science and Automation Engineering (CSAE), vol. 1, pp. 268–274 (2012)
23. Yildiz, B.M., Rensink, A., Bockisch, C., Akşit, M.: 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 (2015)
24. Yildiz, B.M., Rensink, A., Bockisch, C., Aksit, M.: A model-derivation framework for software analysis. In: Hermanns, H., Höfner, P. (eds.) Proceedings MARS. EPTCS, vol. 244, pp. 217–229. Open Publishing Association (2017)