Model-driven Engineering of Machine Executable Code

Michael Eichberg, Martin Monperrus, Sven Kloppenburg, Mira Mezini

Model-driven Engineering of Machine Executable Code

Michael Eichberg[2], Martin Monperrus[2], Sven Kloppenburg[1], and Mira Mezini[2]

1. Kimeta GmbH, Germany
kloppenburg@kimeta.de
2.Technische Universität Darmstadt, Germany
{eichberg,monperrus,mezini}@cs.tu-darmstadt.de

Abstract. Implementing static analyses of machine-level executable code is labor intensive and complex. We show how to leverage model-driven engineering to facilitate the design and implementation of programs doing static analyses. Further, we report on important lessons learned on the benefits and drawbacks while using the following technologies: using the Scala programming language as target of code generation, using XML-Schema to express a metamodel, and using XSLT to implement (a) transformations and (b) a lint like tool. Finally, we report on the use of Prolog for writing model transformations.

1 Introduction

Programs implementing static analyses of machine-executable code are complex [1, 2]. In the terms of Brooks [3], they not only contain intrinsic complexity but also significant accidental complexity. In such programs, several modules are highly interdependent: reading machine-executable code at the byte level, inferring higher-order representation such as control-flow or data-flow graphs, and eventually checking this representation against a property to verify.

Even if these problems are more or less tractable, it is impossible to reuse static analyses across different, yet comparable sets of machine level instructions (e.g. between the Java and the Python sets of bytecodes). However, the ability to write analyses that can be reused across projects is of primary importance in commercial settings. Many industrial projects use multiple languages and technologies and reimplementing basically the same analyses again and again for different languages is not feasible. This state of facts motivated us to design from scratch a static analyses tool in a model-driven manner to improve reuse of analysis components.

So far, we have mentioned four main problems in implementations of static analyses: 1) reading low level formats, 2) inferring higher-order representations, 3) writing the analyses and 4) handling different kinds of executable code. In this paper, we present an architecture that separates all these concerns in different and clearly separated blocks, such that all links from one block to another are implemented using code generation or model transformation. Overall, our contribution is twofold: first, we describe a model-driven architectural blueprint for

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)

the application domain of static analysis tools; second, we report on important benefits and drawbacks of using different technologies for the implementation.

The main lessons we have learned from the design and implementation of our model-driven static analysis toolkit are:

- One of the code generators generates Scala code. It seems that this powerful target programming language has much facilitated the implementation of the generator.
- We chose XML-Schema as the implementation technology of the metamodel of executable code. Many, but not all domain-specific constraints could have been implemented with XML-Schema. This confirms the results of [4, 5] showing that expressing the static semantics is never straightforward within only a structured metamodel.
- Implementing model transformations in Prolog to express static analyses allows us to write concise and declarative analyses.

The remainder of the paper is structured as follows: Section 2 gives the big picture of our approach. Sections 3 presents the metamodel for specifying bytecode instructions. Section 4 discusses the implementation of analyses. Section 5 lists the lessons we learned. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Overview

This section presents the architecture of a new static analysis toolkit that we have been implementing for one year. The architure is designed in a fully model-driven way. First, it is based on three different levels of abstraction, layered in an ontological way as defined by Kühne [6], where the main artifact of each layer is an instance of the upper layer (a meta-layer w.r.t. the lower one.) Second, the architecture uses several times both code generation and model transformation.

Fig. 1 depicts this architecture in terms of the main artifacts and dependencies between them. Boxes represent data (in a larger sense: software to analyze, models, generated code, etc.), and arrows represent relationships between the data (also in a larger sense: generation, transformation, etc.). The three ontological layers are stacked, separated with lines and numbered (from "1" for the most abstract to "3" for the most concrete). The boxes that have a gray background are generated artifacts. We now describe each element at a conceptual level. The details about the technology used and the size and complexity are described in the following sections.

2.1 Meta Layers

Let us now describe the stacked layers of our architecture. We have defined a metamodel for bytecode instructions of virtual machines, which lies in layer #1, at the top of Fig. 1 (a). This metamodel expresses what a bytecode instruction

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)

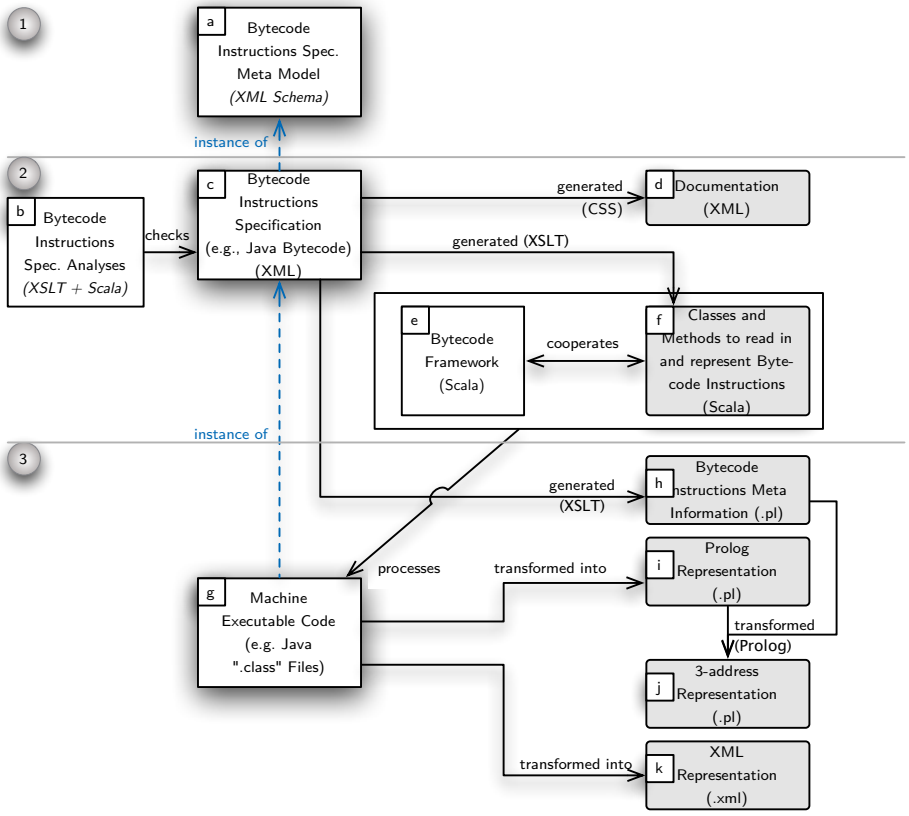

Fig. 1. Overview of the Architecture

is: type of instruction (e.g. add or remove something to the stack), number of arguments, number of bytes in the machine-level format. This metamodel is further discussed in Section 3. An instance of this metamodel is a specification of bytecodes of a particular virtual machine, for instance, the specification of the Java bytecode instruction set [7] is an instance of the bytecode metamodel. In Fig. 1, an instance is represented in layer #2 as a *Bytecode Instructions Specification* (c). Finally, the software that is analyzed is an instance of a particular bytecode format, and is logically in the lowest layer, numbered #3 (i.e. (g)). If one analyzes Java software, each class file is an instance of the Bytecode Instructions Specification. Note that these three elements are shadowed and linked with dashed arrows "instance of" to emphasize the different levels of abstractions. The other boxes are tools (code generators, model transformations) to manipulate these elements.

2.2 Specification of Bytecode Instructions

The central *Bytecode Instructions Specification* (c) is the input of one domain-specific analysis and three code generators. The domain specific analysis (b) takes the specification as input, and checks whether the specification fulfills certain constraints. For instance, that each opcode only identifies one instruction and its directly related variants. This domain specific analysis is necessary since not all domain-specific constraints can be expressed within the uppermost metamodel.

Also, the *Bytecode Instructions Specification* is used to generate a human-readable and structured documentation of the specification (d). It is also used by two code generators: the first one generates object-oriented classes to read machine-executable code (f) and to represent it in memory with domain classes. This generated code cooperates with a manually written library for static analysis (e). However, note that this piece of code only addresses the concern of reading class files. It is independent of the concern of specifying the analyses themselves. The second code generator generates a library that supports the writing of static analyses called *Bytecode Instruction Meta Information*, (h).

2.3 Specification of Static Analyses

So far, we are able to specify a family of bytecode formats and to generate the tool to read and represent them in a domain-specific manner (i.e. no longer byte arrays, but instance of first-class instructions). Let us now explain how to write static analyses.

Basically, there are two ways to write static analyses, first one can write them directly on top of domain classes using standard programming languages or – as in case of our toolkit – as *declarative* static analyses. To enable writing declarative analyses, the analyzed software is transformed into two different artifacts that both represent the machine executable code at the same granularity. Class files can be transformed either into XML files (k) to write static analyses in an XML based language (e.g. XSLT or XQuery). Additionally, they can also be transformed in a set of Prolog facts (*Prolog Representation* – (i) in Fig. 1).

This set of Prolog facts enables us to write static analyses as Prolog rules which handle the basic facts. Further, these facts are given to a model transformation to obtain a higher-order representation of bytecode, called *3-address Representation* (j). The model transformation is also written in Prolog, using the model of analyzed software (as Prolog facts), and an additional source of information, the *Bytecode Instruction Meta Information* that is obtained automatically from the bytecode specification (and discussed above in Section 2.2). Section 4 provides a more in depth view on the static analyses.

2.4 Recapitulation

The architecture of our static analysis toolkit uses three stacked abstraction levels, one domain specific model validation, three code generators, and three model

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)

transformations. The following Sections (3 and 4) describe technical details. Section 5 then exposes the lessons learned in the design and implementation of this architecture.

3 A Meta-Model to Specify Bytecode Instructions

The meta-model for specifying bytecode instructions is called the OPAL Specification Language (OPAL SPL). It enables the encoding of instructions of stack-based intermediate languages, such as, Java Bytecode [7] or CIL Bytecode [8]. OPAL SPL is rich enough to accommodate different bytecode formats and enables the development of analyses and bytecode parsers that are independent of the concrete instance of the specification [9]. To support this goal, the language supports (i) the specification of the type system of the virtual machine which executes the bytecode, (ii) the format of the bytecode instructions and (iii) the specification of the effect on the stack and registers when the instruction is executed. OPAL SPL is focused on specifying the bytecode instruction set and not the complete class file format since the instruction set's structure is more regular and sufficient for developing certain static analyses. The metamodel also supports the declaration of functions (signatures only) to abstract over information that is not directly specified along with an instruction. The functions are implemented manually in a bytecode specific framework ("e" in Fig. 1).

Listing 1.1 shows the specification of the Java bytecode instruction `getfield` as an instance of the OPAL SPL metamodel: the `getfield` field instruction is an instance of the metaclass "Instruction". Note that this specification uses the functions decl_class_type (Line 4 in Listing 1.1) and field_type (Line 5) which are declared as part of the specification of the Java instruction set, these functions return the type information related to an object's field.

```
1 <instruction mnemonic="getfield">
2   The variable fieldRef is initialized by information in the class file.
3   <stack> <form>
4       <before><operand type="decl_class_type(fieldRef)"/><rest/></before>
5       <after><operand type="field_type(fieldRef)"/><rest/></after>
6   </form> </stack>
7 </instruction>
```

Listing 1.1. Specification of the Java Bytecode instruction `getfield` as an instance of the OPAL SPL metamodel

The specification of Java's `if_icmpne` instruction shown in Listing 1.2 demonstrates some of the features of OPAL SPL. In Line 2-5 the format of the instruction is defined; i.e., how the instruction is stored in a class file. In this case the instruction's opcode (Line 3) is an unsigned byte with the value 161. The opcode is followed by a signed short value representing a branch offset (Line 4). When the instruction is executed it pops two int values and then conditionally branches. The instruction does not push a value onto the stack (Line 10).

```
1 <instruction mnemonic="if_icmpne" transfers_control="conditionally">
```

```
  <format> <sequence>
    <u1 var="opcode">161</u1>
    <i2 type="branchoffset" var="branchoffset"/> <!-- relative PC -->
  </sequence> </format>
  <stack> <form>
    <before> <operand type="int_like"/>
             <operand type="int_like"/>
             <rest/></before>
    <after>  <rest/></after>
  </form> </stack>
</instruction>
```

Listing 1.2. Specification of the Java bytecode instruction `if_icmpne` as an instance of the OPAL SPL metamodel

4 Writing Static Analyses

Many static analyses can be expressed w.r.t. abstract representations of instructions, thus generalising the algorithm for a family of languages. This section demonstrates, how to express an algorithm to construct a control-flow graph. The model of a bytecode instruction set enables the generation of classes representing each instruction, as well as the reader of the binary format. This program is then used to transform machine executable code to a model of the software. A model transformation transforms it to a set of Prolog facts. For instance, Listing 1.4 shows the result of the transformation of a simple "Hello World" method (Listing 1.3) to the corresponding Prolog facts.

```
public static void hello(String[] args) {
  if (args.length == 1) print("Hello " + args[0]) else print("Hello World");
}
```

Listing 1.3. Hello World in Java

```
method(cf_1,m_3,'hello',sig([array(class('java/lang','String'))],void),
                public,abstract(no),final(no),static(yes),...).
/*Method Implementation: */
/*PC=1-3 */    Put the value 1 and the length of the array on the stack.
/*PC=3 */      instr(m_3,3,if_icmpne(13)). // conditionally jumps to PC=16
/*PC=5-14 */   print("Hello "+args[0]);
/*PC=15 */     instr(m_3,15,goto_w(4)).
/*PC=16-18*/   print("Hello World");
/*PC=19 */     instr(m_3,19,return(void)).
```

Listing 1.4. Result of a Model Transformation from a Java Class File to a Prolog Representation

Let us now assume that we want to calculate the control-flow graph of a method. In this case, it is necessary to identify all instructions that start with basic blocks and to determine the order in which the basic blocks are executed.

This requires that all control transfer instructions can be identified and also all instructions that – at runtime – are potentially directly executed after these instructions. As shown in Listing 1.2, Line 1 the information that the if_icmpne instruction is a control transfer instruction is directly encoded in the bytecode model. Since the instruction is a conditional control transfer instruction, the next instruction is a potential successor instruction and also the instruction where the program counter (PC) is the PC of the `if` instruction plus the branchoffset

This meta-information is extracted from the Bytecode Instructions Specification (c) and also transformed to Prolog facts (h), as shown in Listing 1.5. A generic model transformation to identify a method's basic blocks is shown in Listing 1.6. The algorithm only assumes that instructions are encoded using a specific syntax (`instr(METHOD_ID, PROGRAM_COUNTER, INSTRUCTION)`) and, if the instruction is a conditional transfer instruction, that the `INSTRUCTION` is encoded as follows: `MNEMONIC(BRANCHOFFSET, ...)`. Furthermore, the algorithm uses the meta-information about instructions (Line 4) to identify all control transfer instructions and all potential successor instructions. Hence, the algorithm does not make any assumptions about specific instructions and can provide a foundation for a complete control-flow graph algorithm.

```
1 control_transfer(if_icmpne,conditionally). % ''conditionally'' is defined by OPAL SPL
2 control_transfer(goto_w,always). % ''always'' is defined by OPAL SPL
3 control_transfer(return,caller). % ''caller'' is defined by OPAL SPL
```

Listing 1.5. Meta-information Related to Control Transfer (They are generated as Prolog facts from the Bytecode Instructions Specification).

```
1 bb_start_instr(MID,0) :- instr(MID,0,_). % the first instr. starts a basic block
2 bb_start_instr(MID,PC) :-
3   instr(MID,CurrentPC,Instr),
4   Instr =.. [Mnemonic|_],control_transfer(Mnemonic,T), T \= 'no',
5   ( ( PC is CurrentPC + 1, instr(MID,PC,_)); % ... if "PC" is valid
6     ( T = 'conditionally',
7       Instr =.. [_,Branchoffset|_], PC is CurrentPC + BranchOffset
8   ) )
```

Listing 1.6. Model transformation in Prolog to identify the method's basic blocks

5 Lessons Learned

In this section, we report on important lessons that we have learned when realizing the discussed architecture.

Overall Approach: Having an explicit meta-model [9] for specifying bytecode instructions did prove useful. First, given the XML-Schema numerous tools were available that facilitate writing documents according to the XML-Schema. These tools provide code completion and immediately report violations of the defined

structure. Additionally, having a schema helped us to get a consistent specification of the Java Bytecode instructions. Several times during the development of the framework we did have to extend and adapt the meta-model to accommodate for the specifics of further instructions. Given the meta-model we were able to rethink and adapt parts of it while being sure to understand the impact on the instructions that have been specified so far, i.e., having an explicit meta-model made it easier to change and extend it since it is possible to assess the impact of changes. Given the meta-model also facilitated the development of generic analyses since it is well-defined which information is generally available. If the specification is only implicitly available one is tempted to look at the concrete instance of it; e.g., the specification of Java bytecode instructions, and to make wrong assumptions about the information that will be common to all instantiations.

Checking Specifications: XML-Schema enables us to express syntactic and, to some extent, semantic constraints which are useful to validate concrete bytecode specifications. However, using XML-schema it is not possible to prevent or detect more complex errors. For example, to make sure that a sequence of instructions is parseable, every instruction has to have a prefix path that uniquely identifies the instruction.[1] In case of the `if` instruction shown in Listing 1.2 the opcode uniquely identifies the instruction. But, in case of some other instructions it is necessary to read multiple values before it is possible to identify the (variant of) the instruction. Using XSLT we were able to efficiently implement an analysis (basically using XPath expressions) that checks that every instruction has a unique prefix path. But, implementing a type checker in XSLT worked out to be too troublesome due to XPATH / XSLT's lack of support of other data structures than lists of nodes. We decided to use Scala for this task. The combination of XML-Schema, XSLT and Scala to fully express the static semantics of our bytecode metamodel is heavyweight. However, to our knowledge and at the time of implementing our architecture, there was no metamodeling paradigm that was powerful enough to express all kinds of constraints in a concise and elegant manner.

Overall, writing a lint like tool for OPAL SPL provided two significant benefits. First, we were able to find numerous errors early on. Second, it helped us designing the language, because writing the analyses requires to take the perspective of the user of the language. This helps to identify issues that are relevant when the specification language is used later on. The effect of writing analyses on the design of the language seems to be roughly comparable to the effect of writing test cases early on.

Scala as the Target Language for Code Generation: From our experience using Scala (compared to, e.g., Java) as the target language for code generation is beneficial. Scala offers the following features that are of particular interest:

[1] In Java Bytecode the instructions do not have the same length, further some instructions even have a flexible length.

flexible syntax, case classes, type inference, implicit type conversions, semicolon inference, an expressive type system, built-in support for XML and tuple types. In the following, we discuss some of these features to highlight the effect on the code generator.

The flexible and concise syntax of Scala is exemplified by class and constructor definitions. Some code that defines a class that inherits from another class and which defines a field that cannot be changed and is publicly available is shown in Listing 1.7.

```
1 class ANEWARRAY ( val cmpType : ReferenceType ) extends Instruction {... }
```

Listing 1.7. Definition of the class ANEWARRAY in Scala

If we compare this class definition with a corresponding class definition in Java (cf. Listing 1.8) the number of parts that are dynamically generated is much smaller. In Scala, the name of the class (ANEWARRAY), the name of the variable (cmpType) and the variable's type (ReferenceType) occur exactly once. In case of Java, the name of the generated class, and the type of the field both appear twice. The field's name even appears four times. Hence, in case of Scala three parts are generated while in case of Java eight would need to be generated. This advantage of Scala is directly reflected in the code generator, it is correspondingly less complex.

```
1 public class ANEWARRAY extends Instruction {
2   public final ReferenceType cmpType;
3   public ANEWARRAY(ReferenceType cmpType) { this.cmpType = cmpType; }
4   ...
5 }
```

Listing 1.8. Class definition in Scala

A similar advantage is offered by Scala's case classes. Case classes are Scala's way to allow pattern matching on objects. Basically, for case classes the scala compiler generates default implementations of the equals and hashCode methods that operate on the object's state and not on its reference. Furthermore, factory methods are provided to create objects of the particular type and functionality is provided to take the objects apart to enable pattern matching. To get this functionality it is just required to add the keyword case in front of a class declaration (cf. Listing 1.9). If we would need to generate the corresponding code, the generator would be orders of magnitude more complex.

```
1 case class ANEWARRAY ...
```

Listing 1.9. Case Class

To sum up, from our experience a language, such as Scala, that provides advanced language features (e.g., higher-order functions, advanced type systems) does make developing a generator easier. Writing the generator will require less code and more errors in the generated code will be detected early on. Overall, the generator will be more comprehensible and maintainable. Many features of Scala

that sometimes are considered "syntactic sugar" were, however, at least as important when developing the generators. As outlined above, semicolon inference, case classes, and implicits support also made the generators less verbose. We are confident that the features proposed for the upcoming versions of Scala (e.g., named arguments and default arguments) will further strengthen the position of Scala as a target language for code generation.

Handling XML-based code generators with Scala: In our architecture, Scala is not only used as target language, but also as an implementation language of certain generators. As shown in Fig. 1 (Artifact (k)), our framework supports an XML representation of bytecode. The functionality to transform the bytecode into XML is provided by Scala classes. Thanks to Scala's built in support for XML, writing a Scala program that generates XML is facilitated.

For instance, the Scala code that generates the XML representation of the `if_icmpne` instruction is shown in Listing 1.10. The method body `toXML` contains an XML pattern which contains values to be replaced (e.g. `pc.toString`). Thanks to Scala, there is no need to explicitly creates nodes of the generated XML document or to enclose the generated text in print-like statements.

```
1  def toXML(pc : Int) =
2    <if_icmpne pc={ pc.toString }>
3      <branchoffset value={ branchoffset.toString }/>
4    </if_icmpne>
```

Listing 1.10. Excerpt of Scala Code that Transforms Java Bytecode into XML

To conclude this section, Table 1 sums up the lessons that we learned while designing and implementing a model-driven static analysis toolkit. These findings are rarely explicitly stated in the literature and supported by empirical facts. Especially, to our knowledge, there is little work explaining the pros of using a powerful and high level language (such as Scala) as a target language of a code generator (see [10]).

6 Related Work

This paper presents a successful application of the model-driven principles to the domain of static-analysis. Although model-driven architecture has been applied to the development of a wide range of domains, e.g. simulation [11] or multi-agent systems [12], we are the very first to report on its use for static-analyses of programs.

However, both our motivations (extensibility and reuse) and the idea of using modeling to facilitate the implementation of static analyses were alread raised in survey papers. For instance, Jackson and Rinard [1] coined the term *"model-driven code analysis"*. They emphasize on the need for explicit models in analysis. We are going further: in our approach, we handle: (i) explicit models of types of

#	Short description
1	Having explicit layers of abstraction helps to identify generic and specific parts.
2	The mature tool support for XML and XML-Schema is really useful for modeling and metamodeling (e.g. code completion).
3	XML-Schema can only be used to only express a small part of the static semantics of a real-world metamodel.
4	XSLT is a pragmatic and good choice to express most of the static semantics of a metamodel implemented with XML-Schema.
5	The need for expliciting the static semantics has a positive impact on the metamodel structure.
6	Using Scala as target language of a code generator ends up in a more readable, maintainable and concise generator.
7	The syntactic support of Scala for writing/reading XML files simplifies the implementation of XML based code generators.

Table 1. Main lessons learnt while implementing a model-driven static analysis toolkit

machine code (Section 3), (ii) explicit models of programs (Section 3) and (iii) explicit models of analyses (written declaratively in Prolog, see Section 4). Also, note that Binkley [2] also states that writing static analyses is difficult as well as designing them as flexible.

Evans and Larochelle [13] presented a lightweight and extensible static analysis. The design of their tool anticipated the support for new checks and annotations. On the contrary, in our approach, all new analyses are supported in a standard way, with no special ad hoc tool. For instance, one can write a new analysis for the bytecode specification (Section 3) as an XSLT program, or a bytecode analysis using Prolog.

The research on reverse engineering has investigated for a long time the need for parsing and understanding software. Rugaber proposes a generic solution called *"model-driven reverse engineering"* [14]. While our main goal is not reverse engineering, we also manipulate program models. Hence, it seems to be straightforward to use our toolchain for reverse-engineering which would be another proof of the flexibility of the approach.

Finally, it is important to differentiate between metamodels of source code and metamodels of machine executable code. They are not at the same level of abstraction. For instance, Strein et al. [15] presented a metamodel for program analysis. While we share similar motivations (extensibility and performance of analyses), their metamodel is much closer to the program structure of the source code with goals such as vizualisation. On the contrary, we reason at the level of the execution machine, with other kinds of verification such as pointer analysis. The same argument applies for [16] in which Störrle uses Prolog not to represent machine code but high-level models.

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)

7 Conclusion

Engineering machine-executable code to write static analyses is a complex task. To tame this complexity, we experimented with the design and implementation of a new static analysis toolkit following a model-driven architecture. We are the first to report on a concrete design and implementation of a model-driven tool chain for implementing static analyses of machine executable code. We managed to obtain a system that is loosely coupled and that allows us to reuse code and semantics across different types of machine-level code (different bytecode instruction sets).

Furthermore, this experiment showed that XML based technologies (XML-Schema, XSLT, XSLT, Scala support for XML) nicely fit together in a model-driven architecture and that using an advanced, high-level language as target of a code generator leads to a more clean and concise code generator.

References

1. Jackson, D., Rinard, M.: Software analysis: A roadmap. In: Proceedings of the Conference on The Future of Software Engineering. (2000) 133–145
2. Binkley, D.: Source code analysis: A road map. In: Future of Software Engineering (FOSE'07), Washington, DC, USA, IEEE Computer Society (2007) 104–119
3. Brooks, F.: No silver bullet: Essence and accidents of software engineering. IEEE computer **20**(4) (1987) 10–19
4. Garcia, M.: Formalizing the well-formedness rules of EJB3QL in UML+ OCL. Volume 4364., Springer (2007) 66
5. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Software: Practice and Experience **39**(15) (2009)
6. Kuehne, T.: Matters of (meta-) modeling. Software and System Modeling **5**(4) (2006) 369–385
7. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley (1999)
8. ISO/IEC Geneva, Switzerland: Information technology – Common Language Infrastructure (CLI) Partitions I to VI. ISO/IEC 23271:2006(E) edn. (2006)
9. Eichberg, M., Sewe, A.: Encoding the java virtual machine's instruction set. In: Proceedings of the Fifth Bytecode Workshop. Electronic Notes in Theoretical Computer Science, Elsevier (to appear)
10. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008)
11. Monperrus, M., Jaozafy, F., Marchalot, G., Champeau, J., Hoeltzener, B., Jézéquel, J.M.: Model-driven simulation of a maritime surveillance system. In: Proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA'2008). Volume 13., Springer (2008)
12. Pavon, J., Gomez-Sanz, J., Fuentes, R.: Model driven development of multi-agent systems. In: Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA'2006). (2006)
13. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE SOFTWARE (2002)

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)

14. Rugaber, S., Stirewalt, K.: Model-driven reverse engineering. IEEE Software **21**(4) (2004) 45–53
15. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An extensible meta-model for program analysis. IEEE Transactions on Software Engineering **33** (2007) 592–607
16. Störrle, H.: A prolog-based approach to representing and querying software engineering models. In: Proceedings of the Workshop on Visual Languages and Logic (VLL'2007). (2007) 71–83

In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'2010)