Prodeling with the Action Language for Foundational UML

Thomas Buchmann
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany

Keywords: UML, Java, Model-driven Development, Behavioral Modeling, Code Generation.

Abstract: Model-driven software development (MDSD) – a software engineering discipline, which gained more and more attention during the last few years – aims at increasing the level of abstraction when developing a software system. The current state of the art in MDSD allows software engineers to capture the static structure in a model, e.g., by using class diagrams provided by the Unified Modeling Language (UML), and to generate source code from it. However, when it comes to expressing the behavior, i.e., method bodies, the UML offers a set of diagrams, which may be used for this purpose. Unfortunately, not all UML diagrams come with a precisely defined execution semantics and thus, code generation is hindered. Recently, the OMG issued the standard for an Action Language for Foundational UML (Alf), which allows for textual modeling of software system and which provides a precise execution semantics. In this paper, an integrator between an UML-based CASE tool and a tool for Alf is presented, which empowers the modeler to work on the desired level of abstraction. The static structure may be specified graphically with the help of package or class diagrams, and the behavior may be added using the textual syntax of Alf. This helps to blur the boundaries between modeling and programming. Executable Java code may be generated from the resulting Alf specification.

1 INTRODUCTION

Model-driven software development (MDSD) (Völter et al., 2006) is a software engineering discipline which receives increasing attention in both research and practice. MDSD intends to reduce the development effort and to increase the productivity of software engineers by generating code from high-level models. To this end, MDSD puts strong emphasis on the development of high-level models rather than on the source code. Models are not considered as documentation or as informal guidelines on how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, MDSE aims at the development of *executable* models.

Throughout the years, UML (OMG, 2015b) has been established as the standard modeling language for model-driven development. It provides a wide range of diagrams to support both structural and behavioral modeling. To support model-driven development in a full-fledged way, it is crucial to derive executable code from executable models. However, generating executable code requires a precise and well-defined execution semantics for behavioral models. Unfortunately, not all behavioral diagrams provided by the UML are equipped with such a well-defined semantics. As a consequence, software engineers nowadays need to manually supply method bodies in the code generated from structural models.

This leads to what used to be called "*the code generation dilemma*" (Buchmann and Schwägerl, 2015): Generated code from higher-level models is extended with hand-written code. Often, these different fragments of the software system evolve separately, which may lead to inconsistencies. Round-trip engineering (Buchmann and Westfechtel, 2013) may help to keep the structural parts consistent, but the problem is the lack of an adequate representation of behavioral fragments.

Over the years, the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) has been established as an extensible platform for the development of MDSE applications, both in the academic community and in industrial projects. It is based on the Ecore metamodel, which is compatible with the Object Management Group (OMG) Meta Object Facility (MOF) specification (OMG, 2015a). Ideally, software engineers operate only on the level of models such that there is no need to inspect or edit the actual source code, which is generated from the models automatically. However, language-specific adaptations to the generated source code are frequently necessary. In EMF, for instance, only structure is modeled by means of class diagrams, whereas behavior is de-

Buchmann, T.
Prodeling with the Action Language for Foundational UML.
DOI: 10.5220/0006353602630270
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 263-270
ISBN: 978-989-758-250-9

scribed by modifications to the generated source code.

The standard for the Action Language for Foundational UML (Alf) (OMG, 2013a), issued by the Object Management Group (OMG), provides the definition of a textual concrete syntax for a foundational subset of UML models (fUML) (OMG, 2013b). In the fUML standard, a precise definition of an execution semantics for a subset of UML is described. The subset includes UML class diagrams to describe the structural aspects of a software system.

In this paper, an integration of our Alf editor and code generator (Buchmann and Rimer, 2016) into the UML-based CASE tool Valkyrie (Buchmann, 2012) is presented. The resulting toolchain allows for seamless integration of UML modeling and Alf programming ("Prodeling") by using bidirectional and incremental model transformations which operate on the abstract syntaxes of both tools. In the integrated environment, Valkyrie may be used for structural modeling and behavior may be specified using Alf. Fully executable Java code may then be generated from the resulting Alf model.

The paper is structured as follows: Related work is discussed in Section 2. In Section 3, a brief overview of Alf is presented. The integration of Alf into our UML-based modeling environment Valkyrie is described in Section 4. Furthermore, an example demonstrating the use of the integrated tool is presented in Section 5. Section 6 concludes the paper.

2 RELATED WORK

Many different tools and approaches have been published in the last few years, which address model-driven development and especially modeling behavior. The resulting tools rely on textual or graphical syntaxes, or a combination thereof. While some tools come with code generation capabilities, others only allow to create models and thus only serve as a visualization tool.

The graphical UML modeling tool **Papyrus** (Guermazi et al., 2015) allows to create UML, SysML and MARTE models using various diagram editors. Additionally, Papyrus offers dedicated support for UML profiles, which includes customizing the Papyrus UI to get a DSL-like look and feel. Papyrus is equipped with a code generation engine allowing for producing source code from class diagrams (currently Java and C++ is supported). Future versions of Papyrus will also come with an Alf editor. A preliminary version of the editor is available and allows a glimpse on its provided features. The textual Alf editor is integrated as a property view and may be used to textually describe elements of package or class diagrams. Furthermore, it allows to describe the behavior of activities. The primary goal of the Papyrus Alf integration is round-tripping between the textual and the graphical syntax and not executing behavioral specifications by generating source code. While Papyrus strictly focuses on a forward engineering process (from model to source code), the approach presented in this paper explicitly addresses round-trip engineering.

Xcore[1] recently gained more and more attention in the modeling community. It provides a textual concrete syntax for Ecore models allowing to express the structure as well as the behavior of the system. In contrast to Alf, the textual concrete syntax is not based on an official standard. Xcore relies on Xbase - a statically typed expression language built on Java - to model behavior. Executable Java code may be generated from Xcore models. Just like the realization of Alf presented in this paper, Xcore blurs the gap between Ecore modeling and Java programming. In contrast to Alf, the behavioral modeling part of Xcore has a strongly procedural character. As a consequence an object-oriented way of modeling is only possible to a limited extent. E.g. there is no way to define object constructors to describe the instantiation of objects of a class. Since Xcore reuses the EMF code generation mechanism (Steinberg et al., 2009), the factory pattern is used for object creation. Furthermore, Alf provides more expressive power, since it is based on fUML, while Xcore only addresses Ecore.

Another textual modeling language, designed for *model-oriented programming* is provided by **Umple**[2]. The language has been developed independently from the EMF context and may be used as an Eclipse plugin or via an online service. In its current state, Umple allows for structural modeling with UML class diagrams and describing behavior using state machines. A code generation engine allows to translate Umple specifications into Java, Ruby or PHP code. Umple scripts may also be visualized using a graphical notation. Unfortunately, the Eclipse based editor only offers basic functions like syntax highlighting and a simple validation of the parsed Umple model. Umple offers an interesting approach, which aims at assisting developers in rasing the level of abstraction ("umplification") in their programs (Lethbridge et al., 2010). Using this approach, a Java program may be stepwise translated into an Umple script. The level of abstraction is raised by using Umple syntax for associations.

PlantUML[3] is another tool, which offers a textual concrete syntax for models. It allows to specify class

[1] http://wiki.eclipse.org/Xcore

[2] http://cruise.site.uottawa.ca/umple

[3] http://plantuml.com

diagrams, use case diagrams, activity diagrams and state charts. Unfortunately, a code generation engine, which allows to transform the PlantUML specifications into executable code is missing. PlantUML uses *Graphviz*[4] to generate a graphical representation of a PlantUML script.

Fujaba (The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth, 2005) is a graphical modeling language based on graph transformations, which allows to express both the structural and the behavioral part of a software system on the modeling level. Furthermore, Fujaba provides a code generation engine that is able to transform the Fujaba specifications into executable Java code. Behavior is specified using *Story Diagrams*. A story diagram resembles UML activity diagrams, where the activities are described using *Story Patterns*. A story pattern specifies a graph transformation rule where both the left hand side and the right hand side of the rule are displayed in a single graphical notation. While story patterns provide a declarative way to describe manipulations of the runtime object graph on a high level of abstraction, the control flow of a method is on a rather basic level as the control flow in activity diagrams is on the same level as data flow diagrams. As a case study (Buchmann et al., 2011) revealed, software systems only contain a low number of problems, which require complex story patterns. The resulting story diagrams nevertheless are big and look complex because of the limited capabilities to express the control flow.

3 THE ACTION LANGUAGE FOR FOUNDATIONAL UML

3.1 Overview

Alf (OMG, 2013a) is an OMG standard, which addresses a textual surface representation for UML modeling elements. It provides an execution semantics by mapping the Alf concrete syntax to the abstract syntax of the OMG standard of *Foundational Subset for Executable UML Models* also known as *Foundational UML* or just *fUML* (OMG, 2013b).

The primary goal is to provide a concrete textual syntax allowing software engineers to specify executable behavior within a wider model, which is represented using the usual graphical notations of UML. A simple use case is the specification of method bodies for operations contained in class diagrams. To this end, it provides a procedural language, whose underlying data model is UML. However, Alf also provides a concrete syntax for structural modeling within the limits of the fUML subset. Please note that in case the execution semantics are not required, Alf is also usable in the context of models, which are not restricted to the fUML subset. The Alf specification comprises both the definition of a concrete and an abstract syntax, which are briefly presented in the subsequent subsections.

3.2 Concrete Syntax

The concrete syntax specification of the Alf standard is described using a context-free grammar in Enhanced-Backus-Naur-Form (EBNF)-like notation. In order to indicate how the abstract syntax tree is constructed from this context-free grammar during parsing, elements of the productions are further annotated.

```
1 ClassDeclaration(d: ClassDefinition)  = [ "
      abstract" (d.isAbstract=true) ] "class"
      ClassifierSignature(d)
```

Listing 1: Alf production rule for a class (OMG, 2013a).

Listing 3.2 shows an example for an EBNF-like production rule, annotated with additional information. The rule produces an instance *d* of the class ClassDefinition. The production body (the right hand side of the rule) further details the ClassDefinition object: It consists of a ClassifierSignature and it may be abstract (indicated by the optional keyword "abstract").

3.3 Abstract Syntax

Alf's abstract syntax is represented by an UML class model of the tree of objects obtained from parsing an Alf text. The Alf grammar is context-free and thus, parsing results in a strictly hierarchical parse tree, from which the so called abstract syntax tree (AST) is derived. Figure 1 gives an overview of the top-level syntax element classes of the Alf abstract syntax. Each syntax element class inherits (in)directly from the abstract base class SyntaxElement. Similar to other textual languages, the Alf abstract syntax tree contains important non-hierarchical relationships and constraints between Alf elements, even if the tree obtained from parsing still is strictly hierarchical with respect to containment relations. These cross-tree relationships may be solely determined from static analysis of the AST. Static semantic analysis is a common procedure in typical programming languages and it is used, e.g., for name resolving and type checking.

[4]http://www.graphviz.org

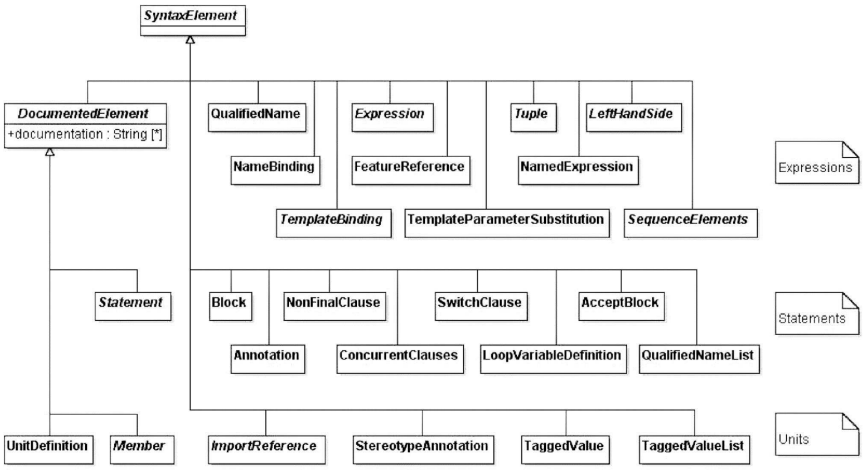

Figure 1: Cutout of the abstract syntax definition of Alf (OMG, 2013a).

4 PRODELING WITH Alf

Figure 2 depicts our approach of integrating UML and Alf models. In the context of the Valkyrie project (Buchmann, 2012), we developed a UML-based CASE tool, which puts special emphasis on code generation. Recently, a stand-alone Alf programming environment was added (Buchmann and Rimer, 2016). This paper describes the integration of both tools.

The UML-based CASE tool Valkyrie provides graphical editors for the following kinds of UML diagrams: package diagrams, class diagrams, object diagrams, use case diagrams, activity diagrams and state machine diagrams. The Graphical Modeling Framework (GMF) (Steinberg et al., 2009) was used to implement the editors. User interactions with the diagram editor are automatically translated into commands which modify the abstract UML syntax by the GMF runtime.

On the other hand, the editor for the textual concrete syntax of Alf was implemented using the Xtext [5] framework, which aids the development of programming languages and domain-specific languages. All aspects of a complete language infrastructure are covered by Xtext including parsing, scoping, linking, validation and code generation. It is tightly integrated into the Eclipse IDE, providing features like syntax highlighting, code completion, quick fixes, and many more. The user interacts with the text-based editor for Alf code. The generated parser creates an in-memory Ecore-based representation of the abstract syntax of the Alf language.

As stated above, Alf primarily was designed to provide means of specifying behavior in UML models. The current state of our Alf editor allows for the textual specification of both structural and behavioral models. I.e., the user may specify the static structure of a software system using packages, classes and associations as well as the behavior which is expressed in method bodies. However, textually specifying the structure of a software system may be odd, especially, when UML-based CASE tools are available, which are optimized for graphical modeling of package and class diagrams.

To this end, our integrator, which is presented in this paper, allows for getting the best out of both worlds. The user may specify package and class diagrams in Valkyrie (as long as they comply to the fUML (OMG, 2013b) subset) and then transform the (f)UML model to a corresponding Alf representation. In the Alf editor, the user may supply the method bodies to implement the desired behavior of the software system. The transformation works in an incremental way of operation. I.e., once structural elements are added or changed in the Alf model, these changes may be propagated back to the (f)UML model. Please note that the user supplied method bodies are retained on subsequent transformations. To this end, the corresponding textual Alf fragments of the method bodies are added as comments to the respective UML operations. This allows for preserving the method implementations, even if methods are moved to different classifiers. In order to generate executable Java source code, the Alf code generator is used, which is briefly introduced in (Buchmann and Rimer, 2016).

The bidirectional transformation between the (f)UML model and the Alf model is achieved by a hand-crafted triple graph transformation system

[5] http://www.eclipse.org/Xtext

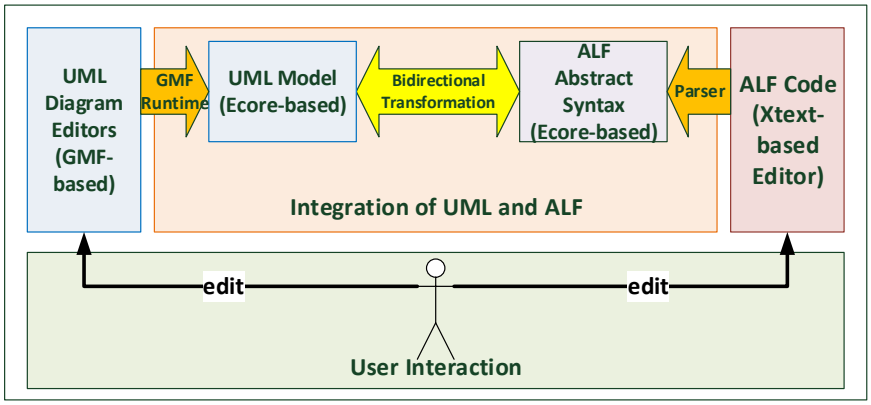

Figure 2: Integration of UML and Alf models.

(TGTS). The TGTS is written in the Xtend[6] programming language. Implementation details of the TGTS are given in (Buchmann and Greiner, 2016), where it was used in a different application scenario. For the integration of (f)UML and Alf, the rules for forward and backward transformations had to be adjusted. The correspondence model and the code responsible for the incremental way of operation could be reused without modifications. In its current state, the following restrictions for the integrator hold:

- Only package and class diagrams are allowed on the UML side
- Class diagrams must conform to the fUML restrictions. In particular, the usage of interfaces and derived attributes is forbidden
- When transforming from (f)UML to Alf, the text formatting information is lost.

While the latter one is a technical issue related to Xtext, the second issue may be tackled by transforming interfaces to abstract classes and by generating getter methods for derived attributes without adding a field in the Alf model.

5 EXAMPLE

In this section, we briefly show a workflow of creating an executable software system, starting with structural modeling in Valkyrie and supplying behavior in form of method bodies in Alf. Finally, executable code is generated from the resulting Alf specification.

As an example, we use a *mobile banking* application in this section. The static structure of the system is modeled using package and class diagrams. The package diagram is shown in Figure 3, and one of the class diagrams is depicted in Figure 4.

[6]http://www.eclipse.org/xtend

The integrator presented in this paper is used to transform the static structure modeled with UML into a corresponding Alf document which is enriched with method bodies specifying the desired behavior of the mobile banking app.

The mobile banking system allows to perform various actions on bank accounts. The execution of these actions is realized using the *Command*-pattern (Gamma et al., 1994). The usage of this pattern allows for decoupling between the caller (an instance of the class BankingAndroidApp, c.f., Figure 3) and the actual execution of the command.

Different types of commands within the mobile banking system are encapsulated in different classes which are specializations of the abstract base class BankingTransactionCommand (c.f., Figure 3). The actual execution of the commands is delegated to an instance of the class BankingPortalHandler. The handler manages a set of bank accounts and executes commands on the proper account taking into account the supplied bank account number.

Listing 5 depicts a cutout of the Alf program that is used to implement the mobile banking system. The class declarations for Bank, Customer, and BankingAccount belonging to the package bankModel are shown, as well as the associations BankHasCustomer and CustomerHasAccount. The code shown in Listing 5 is created after executing the transformation provided by our integrator starting with the UML model as input.

```
1 package mbs{
2   package bankModel {
3     public class Bank { ... }
4
5     public class Customer { ... }
6
7     public class BankingAccount {
          ... }
8
9     public assoc BankHasCustomer {
```

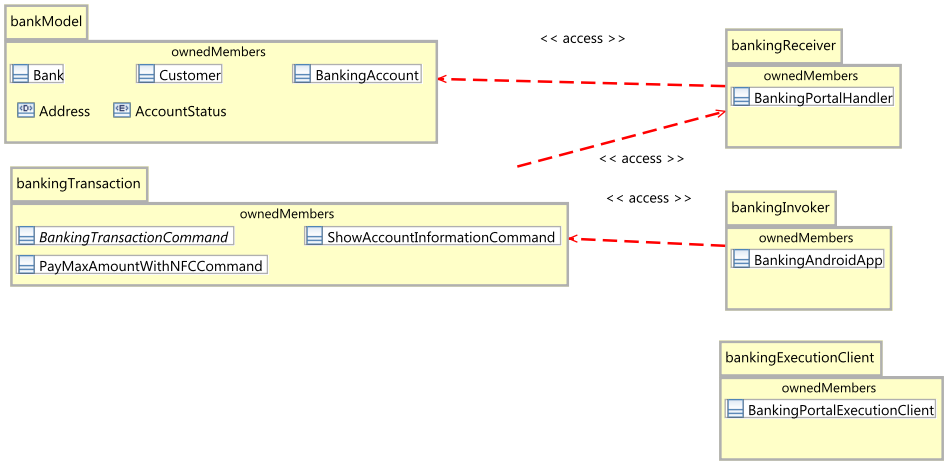

Figure 3: Package diagram of the mobile banking system example.

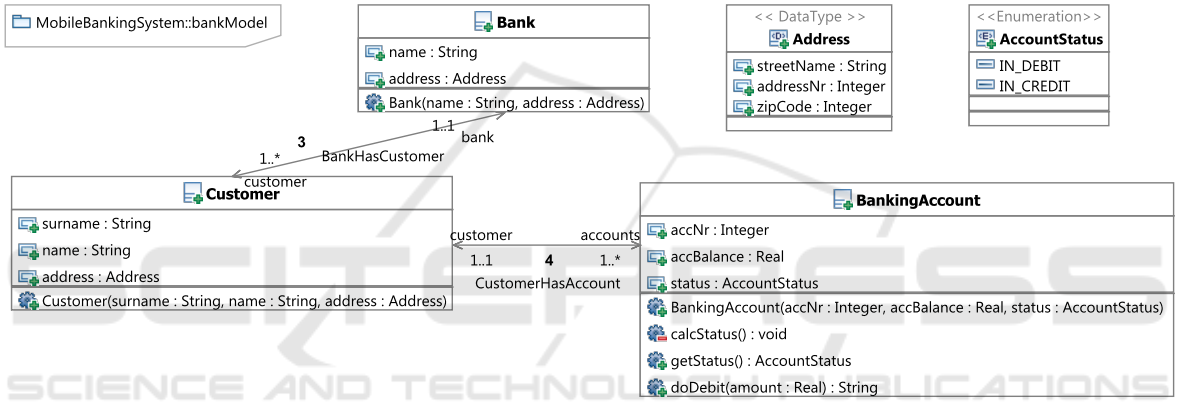

Figure 4: Cutout of the class diagram of the mobile banking system example.

```
10        public bank : Bank[1];
11        public customer : Customer
              [1..*];
12      }
13
14      public assoc CustomerHasAccount
            {
15        public customer : Customer[1];
16        public accounts :
              BankingAccount[1..*];
17      }
18    }
19  }
```

Listing 2: Cutout of the Alf program after the transformation from (fUML) to Alf.

Listing 5 depicts the implementation of the operation responsible for creating the information associated with a certain account in the class BankingPortalHandler. The information is retrieved based on the account number which is passed as an input parameter in the method head. The implementation uses control structures like loops (c.f., line 8 in Listing 5) or conditional statements (c.f., line 9). The Alf code was supplied with the corresponding Alf editor.

```
1  public createAccountInformation(in
       passedAccNr:Integer):String {
2    let result:String = "Account: ";
3    let ownerName:String = "";
4    let ownerSurname:String = "";
5    let balance:Double = 0.0;
6    let accNr:Integer = 0;
7
8    for (BankingAccount acc: this.
       accs) {
9    if (acc != null && acc.accNr ==
       passedAccNr) {
10     balance = acc.accBalance;
11     accNr = acc.accNr;
12     let accOwner:Customer =
         CustomerHasAccount::
         customer(acc);
13     if (accOwner != null) {
14       ownerName = accOwner.name;
15       ownerSurname = accOwner.
           surname;
```

```
16      }
17    }
18    }
19
20    result += accNr + ":\n";
21    result += ownerName + " " +
          ownerSurname + "\n";
22    result +=  "Balance:" + balance;
23    return result;
24  }
```

Listing 3: Example of a supplied method body in Alf.

In case structural modifications are required while supplying the method bodies, the user has the choice of performing these changes in the UML model or in the Alf code. In case the user decides for the first option, the backward transformation supplied with our integrator needs to be invoked. The execution of the transformation results in an update of the UML model, where the user supplied method bodies of the Alf specification are added to comments of the corresponding operations in the UML model. Afterwards, the user may perform the structural changes in the respective diagram editors of the Valkyrie tool. Once he finished this task, the forward transformation of our integrator may be invoked and the Alf model gets updated accordingly.

After all method bodies have been supplied, the user may invoke the code generator of our Alf editor in order to generate fully executable Java code. The resulting Java source code is not shown in this paper due to space restrictions.

6 CONCLUSION AND FUTURE WORK

In this paper, an approach to providing tool support for unifying modeling and programming has been presented. The OMG Alf specification (OMG, 2013a) describes a textual concrete syntax for a subset of UML (fUML) (OMG, 2013b). In previous projects, a UML-based CASE tool (Buchmann, 2012) and an implementation of the Alf specification (Buchmann and Rimer, 2016) have been created. The work presented in this paper integrates both tools by means of a bidirectional and incremental transformation, which allows for seamless integration where the user may work either with graphical diagrams or textual code to specify the static structure of the system. Behavior may only be supplied in the textual syntax. In order to execute the resulting software systems, a Java code generator from the Alf editor may be used, which allows for the creation of fully executable Java programs.

Future work comprises case studies in order to evaluate our approach. Furthermore, we are aiming at a tighter integration of Valkyrie and Alf, where the user of the Alf editor is working in a local context, i.e. directly on the method without having to deal with the complete Alf code resulting from the UML model. In addition, we are investigating fall-back mechanisms, which will be applied in case the UML model does not conform to the fUML subset.

ACKNOWLEDGMENTS

The author wants to thank Johannes Schröpfer for implementing parts of the Alf integrator in his Bachelor thesis. Further acknowledgments go to Bernhard Westfechtel for his valuable and much appreciated comments on the draft of this paper.

REFERENCES

Buchmann, T. (2012). Valkyrie: A UML-Based Model-Driven Environment for Model-Driven Software Engineering. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 147–157, Rome, Italy. SciTePress.

Buchmann, T. and Greiner, S. (2016). Handcrafting a triple graph transformation system to realize round-trip engineering between UML class models and java source code. In Maciaszek, L. A., Cardoso, J. S., Ludwig, A., van Sinderen, M., and Cabello, E., editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.*, pages 27–38. SciTePress.

Buchmann, T. and Rimer, A. (2016). Unifying Modeling and Programming with ALF. In Kaindl, H. and Meli, R., editors, *Proceedings of the 2nd International Conference on Advances and Trends in Software Engineering (SOFTENG 2016)*, page 6. IARIA.

Buchmann, T. and Schwägerl, F. (2015). On A-posteriori Integration of Ecore Models and Hand-written Java Code. In Pascal Lorenz, M. v. S. and Cardoso, J., editors, *Proceedings of the 10th International Conference on Software Paradigm Trends*, pages 95–102. SciTePress.

Buchmann, T. and Westfechtel, B. (2013). Towards Incremental Round-Trip Engineering Using Model Transformations. In Demirors, O. and Turetken, O., editors, *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013)*, pages 130–133. IEEE Conference Publishing Service.

Buchmann, T., Westfechtel, B., and Winetzhammer, S. (2011). The added value of programmed graph transformations - A case study from software configuration management. In Schürr, A., Varró, D., and Varró,

G., editors, *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE 2011, Budapest, Hungary, October 4-7, 2011, Revised Selected and Invited Papers*, volume 7233 of *Lecture Notes in Computer Science*, pages 198–209. Springer.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Upper Saddle River, NJ.

Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., and Gérard, S. (2015). Executable modeling with fuml and alf in papyrus: Tooling and experiments. In Mayerhofer, T., Langer, P., Seidewitz, E., and Gray, J., editors, *Proceedings of the 1st International Workshop on Executable Modeling colocated with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 27, 2015.*, volume 1560 of *CEUR Workshop Proceedings*, pages 3–8. CEUR-WS.org.

Lethbridge, T. C., Forward, A., and Badreddin, O. (2010). Umplification: Refactoring to incrementally add abstraction to a program. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 220–224. IEEE.

OMG (2013a). *Action Language for Foundational UML (ALF)*. Object Management Group, Needham, MA, formal/2013-09-01 edition.

OMG (2013b). *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Object Management Group, Needham, MA, formal/2013-08-06 edition.

OMG (2015a). *Meta Object Facility (MOF) Version 2.5*. OMG, Needham, MA, formal/2015-06-05 edition.

OMG (2015b). *Unified Modeling Language (UML)*. Object Management Group, Needham, MA, formal/15-03-01 edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2nd edition.

The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth (2005). The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. In Giese, H. and Zündorf, A., editors, *Proceedings of the 3rd international Fujaba Days*, pages 1–13.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.