Unifying Modeling and Programming with ALF

Thomas Buchmann and Alexander Rimer
University of Bayreuth
Chair of Applied Computer Science I
Bayreuth, Germany
email: {thomas.buchmann, alexander.rimer}@uni-bayreuth.de

Abstract—**Model-driven software engineering has become more and more popular during the last decade. While modeling the static structure of a software system is almost state-of-the art nowadays, programming is still required to supply behavior, i.e., method bodies. Unified Modeling Language (UML) class diagrams constitute the standard in structural modeling. Behavioral modeling, on the other hand, may be achieved graphically with a set of UML diagrams or with textual languages. Unfortunately, not all UML diagrams come with a precisely defined execution semantics and thus, code generation is hindered. In this paper, an implementation of the Action Language for Foundational UML (Alf) standard is presented, which allows for textual modeling of software systems. Alf is defined for a subset of UML for which a precise execution semantics is provided. The modeler is empowered to specify both the static structure as well as the behavior with the Alf editor. This helps to blur the boundaries between modeling and programming. Furthermore, an approach to generate executable Java code from Alf programs is presented, which is already designed particularly with regard to round-trip engineering between Alf models and Java source code.**

Keywords–model-driven development; behavioral modeling; textual concrete syntax; code generation.

I. Introduction

Increasing the productivity of software engineers is the main goal of Model-driven Software Engineering (MDSE) [1]. To this end, MDSE 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. Over the years, UML [2] has been established as the standard modeling language for model-driven development. A wide range of diagrams is provided to support both structural and behavioral modeling. Model-driven development is only supported in a full-fledged way, if executable code may be obtained from behavioral models. Generating executable code requires a precise and well-defined execution semantics for behavioral models. Unfortunately, this is only the case for some UML diagrams. 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*" [3]: 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 [4] may help to keep the structural parts consistent, but the problem is the lack of an adequate representation of behavioral fragments.

The Eclipse Modeling Framework (EMF) [5] has been established as an extensible platform for the development of MDSE applications. It is based on the Ecore meta-model, which is compatible with the Object Management Group (OMG) Meta Object Facility (MOF) specification [6]. 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, practical experiences have shown that 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 described by modifications to the generated source code. The OMG standard for the Action Language for Foundational UML (Alf) [7] provides the definition of a textual concrete syntax for a foundational subset of UML models (fUML) [8]. 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 and UML activity diagrams for the behavioral part.

In this paper, an implementation of the Alf standard is presented. To the best of our knowledge, there is no other realization of the Alf standard, which also allows to generate executable code from corresponding Alf scripts (c.f. Section IV). The currently realized features of the Alf editor are discussed, and some insights on the code generator, which is used to transform Alf scripts into executable Java programs are also given in this paper. The paper is structured as follows: In Section II, a brief overview of Alf is presented. The realization of the Alf editor and the corresponding code generation is discussed in detail in Section III before related work is discussed in the following section. Section V concludes the paper.

II. The Action Language for Foundational UML

A. Overview

As stated above, *Alf* [7] is an OMG standard, addressing a textual surface representation for UML modeling elements. Furthermore, it provides an execution semantics via a mapping of 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* [8]. 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 language with a procedural character,

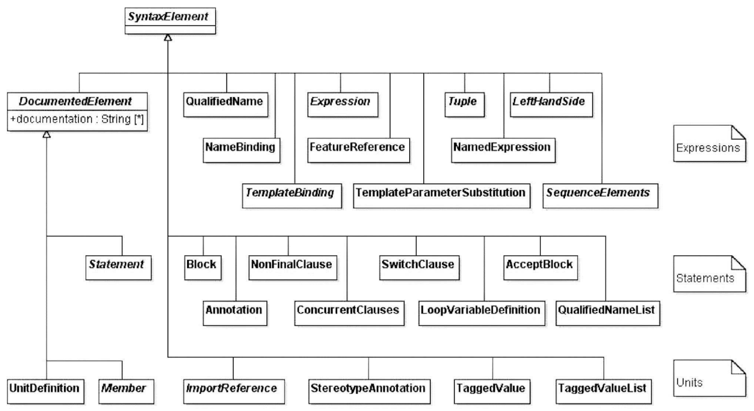


Figure 1: Cutout of the abstract syntax definition of Alf [7]

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.

B. 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.

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

Listing 1: Alf production rule for a class [7]

Listing 1 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").

C. 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.

III. Realization

In this section, details of the implementation of the Alf standard are presented. As the UML modeling suite *Valkyrie* [9] is built upon Eclipse modeling technology, and the road map includes the integration of Alf into Valkyrie, EMF [5] and Xtext [10] have been used for the realization. In its current state, the Alf editor adopts most language features of the Alf standard. For the moment, language constructs, which are not directly needed to describe the behavior of method bodies contained in operations specified in class diagrams have been omitted (e.g., some statements like inline, accept or classify statements will be implemented in future work). The main focus has been put on the structural modeling capabilities and the description of behavior of activities plus the generation of executable Java code as these are the mandatory building blocks, which are required to integrate Valkyrie and Alf at a later stage.

A. Meta-model

According to the abstract syntax specification given in the Alf standard, a corresponding Ecore model was created. In its current state, the Alf meta-model comprises more than 100 meta classes and thus, only some relevant cutouts can be presented here due to space restrictions. Please note that the Alf specification provides a model for the abstract syntax of the language. However, due to the tools used to implement the specification, tool-specific adaptations had to be done. Furthermore, some language concepts have been omitted and will be added at a later stage, as described above. Figure 2 depicts the cutout of our realization of the Alf meta-model responsible for the structural modeling aspects of the language.

This part of the meta-model comprises all mandatory meta-classes for Packages, Classifiers, and Features, which are required for structural modeling. The root element of each Alf model is represented by the meta-class Model. A Model contains an arbitrary number of PackageableElements, i.e., Packages and Classifiers. Classifiers may establish an inheritance hierarchy using the meta-class Generalization. Like

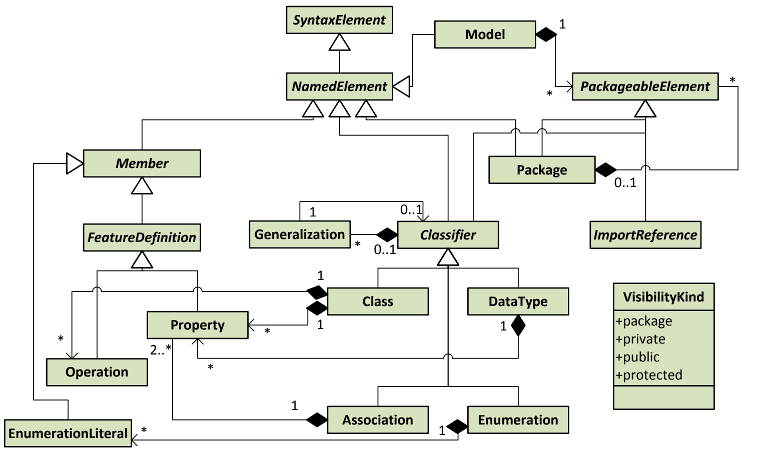


Figure 2: Relevant parts responsible for structural modeling aspects in our realization of the Alf abstract syntax.

UML, Alf supports multiple inheritance between Classifiers. Classifiers are further specialized by the subtypes Class, DataType, Enumeration, and Association. While classes, datatypes and associations contain attributes represented by the meta-class Property, Classes additionally contain Operations.

In Alf, Operations are used for behavioral modeling. Figure 3 depicts a simplified cutout of the Alf meta-model showing the relevant parts. Operations may be parameterized and may contain an "Operation Method". Parameters of an operation are typed and they possess a name. Additionally, the direction of the parameter is indicated by the enumeration ParameterDirectionKind. Possible values are in, out or inout. The type of an operation is determined by its return type. The method of an Operation contains the complete behavior realized by the operation. One possible way of realizing this method is using a Block [7], which represents the body of the operation. The block itself comprises an arbitrary number of Statements.

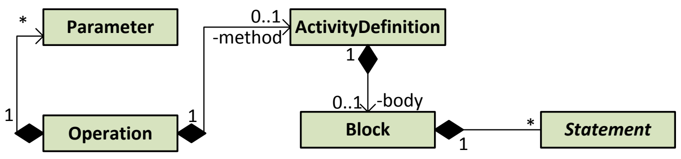


Figure 3: Simplified cutout of the Alf meta-model for Operations.

The Alf standard defines different types of Statements. Figure 4 shows the statements, which are currently realized in our implementation of the Alf standard as subtypes of the abstract class Statement.

Besides statements realizing the return values of an operation (ReturnStatement), several statements dealing with the control flow are included. Local variables may be expressed using the LocalNameDeclarationStatement. The initialization of local variables is done using Expressions, which are encapsulated by ExpressionStatements.

Expressions constitute the most fine-grained way of modeling in Alf and may be used in different contexts. For example, they are used for assignments, calculation, modeling of constraints or the access to operations and attributes. The current state of the Alf meta-model comprises various specializations of the meta-class Expression, as depicted in Figure 5.

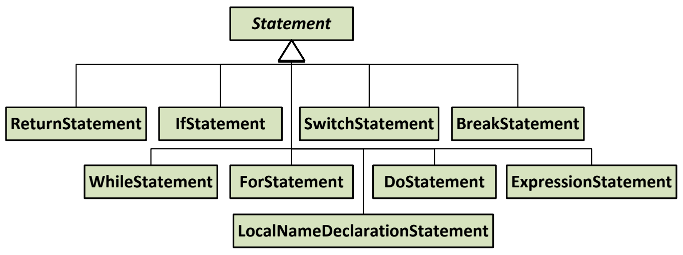


Figure 4: Simplified cutout of the Alf meta-model for Statements.

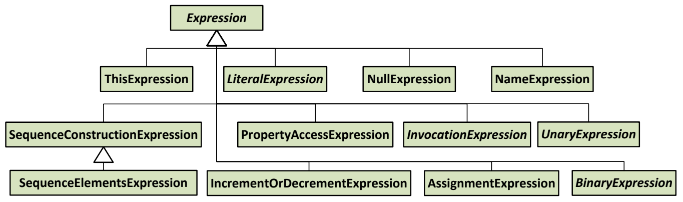


Figure 5: Simplified cutout of the Alf meta-model for Expressions.

LiteralExpression is the superclass for various kinds of literal expressions. They constitute the simplest kind of expressions and may represent, e.g., strings, boolean values or numbers. Literal expressions may be used in combination with assignment expressions or for comparison operations in conditional expressions. While UnaryExpressions, which are used for boolean or arithmetic negations or type queries (instanceof) only contain one operator and one operand, BinaryExpressions have one operator and two operands. Each of the operands may be an Expression as well and they may be used in conditions as well as in assignment expressions.

B. Editor

Xtext [10], an Eclipse framework aiding the development of programming languages and domain-specific languages

(DSLs), has been used to create a textual editor on top of the meta-model described in the previous subsection. Xtext allows for a rapid implementation of languages and it covers all aspects of a complete language infrastructure, like parsing, scoping, linking, validation, code generation plus a complete Eclipse IDE integration providing features like syntax highlighting, code completion, quick fixes and many more. Furthermore, it provides means to easily extend the default behavior of the IDE components using the Xtend [11] programming language. Since Xtext uses ANTLR [12] as a parser generator, there are some restrictions on the grammar, such as that there must not be any left-recursive rules. The Alf standard uses left-recursion in various places and thus, these rules had to be rewritten in order to be used with Xtext. For example this was needed when realizing various Expressions, e.g., expressions used for member access (dot notation).

In order to provide meaningful error messages and modern IDE feature like quickfixes for the end-users of the Alf editor, a set of validation rules has been implemented. Furthermore, validation rules are mandatory to check the static semantics given in the Alf standard. Among others, the validation rules comprise:

- Access restrictions on features with respect to their visibility
- Uniqueness of local variables in different contexts (bodies, control structures and operations)
- Validation of inheritance hierarchies (no cycles)

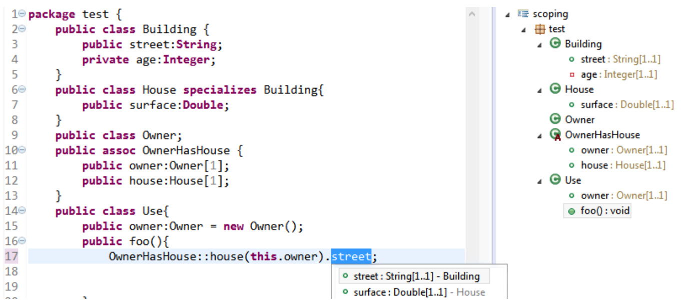
In order to provide an import mechanism for Alf elements, the scoping rules had to be specified. The default Xtext scoping mechanism calculates visibilities for each AST element, and the result is used, e.g., for linking and for validation of DSL programs. For the Alf editor the scope for different contexts had to be determined: A scope for blocks (in operations and control structures) is needed, furthermore a different scope for realizing feature access (attributes or operations) on classifiers is required (taking into account possible inheritance hierarchies). For example, inherited properties and operations are considered when the scope is determined. In addition, the correct scope for accessing link operations of associations and enumeration literals is also provided.

Figure 6 shows the running Alf editor. The code completion menu depicts possible values, which could be used at the current cursor position. The list of possible matches was computed using the implemented scoping rules as described above.

C. Type System

The Alf specification uses an implicit type system, which allows but does not necessarily require the explicit declaration of typing within an activity. However, static type checking is always provided based on the types specified in the structural model elements. In general, requirements for type systems comprise the following tasks:

- **Type definition:** Various model elements are defined as actual and fixed types (e.g. primitive types).



```
package test {
    public class Building {
        public street:String;
        private age:Integer;
    }
    public class House specializes Building{
        public surface:Double;
    }
    public class Owner;
    public assoc OwnerHasHouse {
        public owner:Owner[1];
        public house:House[1];
    }
    public class Use{
        public owner:Owner = new Owner();
        public foo(){
            OwnerHasHouse::house(this.owner).street;
```

Figure 6: Running Alf editor, showing scoping applied in the code completion menu.

- **Type calculation** A type system should be able to calculate the type of an element and assign a type to an element respectively.
- **Type validation** A type system should provide a set of validation rules, which ensure the well-typedness of all model elements.

In the Xtext context, several frameworks exist, which assist DSL engineers when implementing a type system. For the Alf editor, the tool *Xtext Typesystem (XTS)* [13] has been used, which is optimized for expression-oriented DSLs. XTS provides a DSL, which allows to declaratively specify type system rules for Xtext DSLs. The validation of the type system rules integrates seamlessly into the Xtext validation engine. As an example, Listing 2 shows a simplified cutout of the type definition for a local variable using XTS.

```
1  public class AlfTypesystem extends DefaultTypesystem {
2  private AlfPackage lang = AlfPackage.eINSTANCE;
3  protected void initialize() {
4  //Type definition
5  useCloneAsType(lang.getIntegerType());
6  ... // do the same for all other primitive types
7
8  //Type assignment
9  useTypeOfFeature(lang.getLocalNameDeclarationStatement()
       , lang.getLocalNameDeclarationStatement_Type());
10 useFixedType(lang.getNaturalLiteralExpression(), lang.
       getIntegerType());
11 ... // define other fixed types
12
13 //Type validation
14 ensureOrderedCompatibility(lang.
       getLocalNameDeclarationStatement(),    lang.
       getLocalNameDeclarationStatement_Type(),
15 lang.getLocalNameDeclarationStatement_Expression());
16 }
17 }
```

Listing 2: Type definition of a local variable using XTS

When the type system is initialized, primitive types are defined as clones of their own type (c.f. line 5 in Listing 2). For the meta-class LocalNameDeclarationStatement the assignment of the type is depicted in line 9. The fixed type IntegerType is set for the meta-class NaturalLiteralExpression afterwards before rules for validating the types are specified.

D. Code Generation

In order to execute Alf specifications, they need to be translated into executable source code. In this case, the Alf model acts as platform independent model (PIM), and has to be transformed into a platform specific one (PSM) first. As future plans for the Alf editor comprise the integration into the UML case tool Valkyrie [9], the MoDisco Java meta-model was chosen as platform specific model. To this end, a uni-directional model-to-model transformation from the Alf model to the MoDisco [14] Java model has been implemented. MoDisco is an Eclipse framework dedicated to software modernization projects. It provides, among others, an Ecore based Java meta-model (resembling the Java AST) and a corresponding discovery mechanism, which allows to create instances of the Java meta-model given on Java source code input. Furthermore, a model-to-text transformation is included allowing to create Java source code from Java model instances.

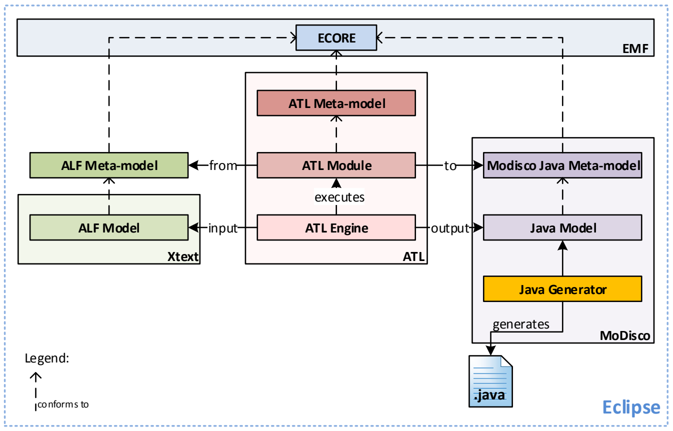


Figure 7: Conceptual design of the code generation using model transformations.

Figure 7 depicts the conceptual design of the code generation engine of the Alf editor. The Atlas Transformation Language (ATL) [15] was used as the model transformation tool, since it works very well for uni-directional model-to-model (M2M) transformations. ATL follows a hybrid approach, providing both declarative and imperative language constructs. Furthermore, ATL offers a concept for module superimposition, allowing to modularize and reuse transformation rules. The M2M transformation implemented for the Alf editor takes the Alf abstract syntax as an input and produces an instance of the MoDisco Java meta-model as an output (c.f. Figure 7). Afterwards, the model-to-text transformation provided by the MoDisco framework is invoked on the resulting output model to generate Java source code files. Please note that writing the ATL transformation rules was a tedious task, since the level of abstraction in Alf is much higher than in Java. For example, a property of an Alf class has to be transformed into a Java field declaration plus corresponding accessor methods. In case of an association, navigability also has to be taken into account and the resulting Java model must contain all (AST-) elements allowing to generate Java code, which ensures consistency of the association ends. In total, the ATL transformation implemented for the Alf editor comprises more than 9000 lines of ATL code distributed over 8 modules.

IV. 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** [16] 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 [17] 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 [5], 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** [18]. 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 [19]. 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 [20] is another tool, which offers a textual concrete syntax for models. It allows to specify class 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* [21] to generate a graphical representation of a PlantUML script.

Fujaba [22] 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 [23] 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.

V. Conclusion and Future Work

In this paper, an approach to providing tool support for unifying modeling and programming has been presented. To this end, an implementation of the OMG Alf specification [7], which describes a textual concrete syntax for a subset of UML (fUML) [8] has been created. Using the Alf editor, the software engineer may specify both the structure as well as the behavior of a software system on the model level. As a consequence, model transformations may directly be applied to Alf scripts. In order to execute Alf programs, a Java code generator is provided, which allows for the creation of fully executable Java programs and which is already designed particularly with regard to round-trip engineering.

Future work comprises the integration of the (currently) stand-alone Alf editor into the UML modeling tool suite *Valkryie* [9]. To this end, besides integrating textual and graphical modeling, also a mapping of the Alf abstract syntax to the fUML abstract syntax is required as proposed in the Alf standard [7]. Furthermore, a case study is performed in order to evaluate the modeling capabilities of the Alf editor.

References

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

[2] OMG, *Unified Modeling Language (UML)*, formal/15-03-01 ed., Object Management Group, Needham, MA, Mar. 2015.

[3] T. Buchmann and F. Schwgerl, "On A-posteriori Integration of Ecore Models and Hand-written Java Code," in *Proceedings of the 10th International Conference on Software Paradigm Trends*, M. v. S. Pascal Lorenz and J. Cardoso, Eds. SCITEPRESS, July 2015, pp. 95–102.

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

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modeling Framework*, 2nd ed., ser. The Eclipse Series. Boston, MA: Addison-Wesley, 2009.

[6] OMG, *Meta Object Facility (MOF) Core*, formal/2011-08-07 ed., Object Management Group, Needham, MA, Aug. 2011.

[7] OMG, *Action Language for Foundational UML (ALF)*, formal/2013-09-01 ed., Object Management Group, Needham, MA, Sep. 2013.

[8] OMG, *Semantics of a Foundational Subset for Executable UML Models (fUML)*, formal/2013-08-06 ed., Object Management Group, Needham, MA, Aug. 2013.

[9] T. Buchmann, "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)*, Rome, Italy, 2012, pp. 147–157.

[10] "Xtext project," http://www.eclipse.org/Xtext, visited: 2015.09.30.

[11] "Xtend project," http://www.eclipse.org/xtend, visited: 2015.09.30.

[12] T. Parr and K. Fisher, "LL(*): the foundation of the ANTLR parser generator," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 425–436. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993548

[13] L. Bettini, D. Stoll, M. Völter, and S. Colameo, "Approaches and tools for implementing type systems in xtext," in *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, K. Czarnecki and G. Hedin, Eds., vol. 7745. Springer, 2012, pp. 392–412. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36089-3_22

[14] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM International Conference on Automated software engineering (ASE 2010)*, Antwerp, Belgium, 2010, pp. 173–174.

[15] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, pp. 31–39, 2008, special Issue on Experimental Software and Toolkits (EST).

[16] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier, "Papyrus UML: an open source toolset for MDA," in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer, 2009, pp. 1–4.

[17] "Xcore," http://wiki.eclipse.org/Xcore, visited: 2015.09.30.

[18] "Umple Language," http://cruise.site.uottawa.ca/umple/, visited: 2015.09.30.

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

[20] "PlantUML," http://plantuml.com/, visited: 2015.09.30.

[21] "Graphviz," http://www.graphviz.org, visited: 2015.09.30.

[22] The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth, "The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth," in *Proceedings of the 3rd international Fujaba Days*, H. Giese and A. Zündorf, Eds., September 2005, pp. 1–13.

[23] T. Buchmann, B. Westfechtel, and S. Winetzhammer, "The added value of programmed graph transformations — a case study from software configuration management," in *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, A. Schürr, D. Varro, and G. Varro, Eds., Budapest, Hungary, 2012.