On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf

Zoltán Micskei*, Raimund-Andreas Konnerth, Benedek Horváth, Oszkár Semeráth, András Vörös, and Dániel Varró*

Budapest University of Technology and Economics
Magyar tudósok krt. 2., 1117, Budapest, Hungary
* {micskeiz,varro}@mit.bme.hu

Abstract. Executable and well-defined models are a cornerstone of model driven engineering. We are currently working on a transformation chain from UML models to formal verification tools. In the context of the UML language, the fUML and Alf specifications offer a standardized way for the semantics of the basic model elements and a textual specification language. Open source modeling tools started to adapt these specifications. However, their support is of varying degree. This paper summarizes our experiences with the open source tools regarding fUML and Alf support, and different model transformation technologies in order to analyse them with formal verification tools.

Keywords: fUML, Alf, tool, open source, verification, MDE

1 Introduction

Model driven engineering (MDE) is changing software and systems development in different application domains. In MDE, models are the primary artifacts from which other artifacts (code, deployment descriptions, etc.) are generated [2]. Thus MDE is based on *modeling languages* and *transformations*. However, development processes and tools need to be adjusted to the MDE paradigm.

The Unified Modeling Language (UML) is a general purpose, widely used modeling language used for describing the different aspects of software systems. It offers several modeling notations to express not only the structure but the behavior of the modeled system (e.g. with state machines, activities). But UML is criticized because of the lack of a precise definition of the language [3]. For this reason, the Object Management Group (OMG) created the *Semantics of a Foundational Subset for Executable UML Models (fUML)* [12] and *Action Language for Foundational UML (Alf)* [11] specifications. Together these specifications offer a precise semantic definition of the basic static and dynamic language elements and a textual notation, which are needed to have an executable system model. Thus, fUML can be used as a basic building block in a UML-based MDE approach [5].

Some commercial MDE tools already offer an executable variant of UML. Some of these are using fUML (e.g. No Magic's Cameo Simulation Toolkit for

MagicDraw) or are using other approaches (e.g. xtUML in the BridgePoint modelling tool of Mentor Graphics). These tools are able to execute, debug and transform UML behavioral models.

Fortunately, more and more open source tools have started to include some kind of executable UML functionality, and support for fUML or Alf. However, due to the nature of open source, these developments are sometimes overlapping and redundant, some projects are not active any more, and their technology readiness level and documentation is uneven. Thus, if someone wants to engage in a UML-based MDE project the following questions naturally arise.

1. What open source tools are available for fUML and Alf?
2. What are their maturity level?
3. How can the different tools be connected to solve a complex problem?

We are currently working on a tool chain to provide verification of UML behavioral models in the communication domain by transforming them to the input of formal model checkers and back-annotating the results. The goal of the verification is to prove the deadlock freedom of protocols. When developing the initial prototypes, we tried to base our work on open source components for fUML or Alf. This paper primarily aims at summarizing our experience of using these open source components for model analysis purposes.

Section 2 introduces our modeling and analysis approach, then collects the fUML and Alf tools we are aware of. Section 3 details the avalilable transformation technologies, and presents our Eclipse-based transformation prototype implementation. Section 4 summarizes our experiences and some open issues with the specifications and the tooling.

2 Overview

The goal of our project is to provide formal analysis for behavioral UML models by transforming these models to the input languages of formal verification tools (e.g. model checkers). We aim to use a combination of UML diagrams as input to capture static and dynamic aspects. In order to facilitate the adaptation of the tool chain to different back-end tools, we intend to use intermediate models between the source engineering model and the target verifier model, which is proposed in numerous model analysis approaches [4,6]. In this paper, we propose to use existing standard UML behavioral specifications (namely, fUML and Alf) and open source tools to realize this approach.

2.1 Open Source Tools for fUML and Alf

The goal of the fUML specification of the OMG is to give a precise semantic definition for a subset of UML. This subset includes parts of the classes, behaviors, activities and actions packages of UML Superstructure, but does not include state machines or sequence diagrams. The standardization including composite structures (such as components, ports...) is ongoing [10]. The specification

Table 1: Open source fUML and Alf tools as of 2014-07

	Tool	Version	Modification	Platform	Goal
fUML	fUML Ref. Impl.	1.1.0	2013-06	Java/console	Interpreter
	Moka	1.0.0	2014-06	Eclipse	fUML execution engine
	Moliz	1.0	2014-06	Eclipse	fUML execution engine
Alf	Alf Ref. Impl.	0.4.0	2014-05	Java/console	Interpreter
	Papyrus Alf Editor	-	2014-07	Eclipse	Editor

contains the definition of (1) an *abstract syntax* for the subset of UML Infrastructure/Superstructure, (2) an *execution model* for the operational semantics, (3) Java language *semantic definitions* for the behavior of the elements, (4) a *model library* (functions for primitive types, channels), and (5) axiomatic semantics for a base UML.

The Action Language for Foundational UML (Alf) is a standardized textual language for specifying executable behavior. It can be used to extend a graphical UML model (e.g. to define actions or guards). However, it can also express structural model elements, thus an entire UML model can be described in a textual language. The specification consists of (1) an EBNF grammar for the Alf language, and (2) a modeling library for primitive types, functions and collections. UML state machines are not in fUML, thus Alf does not cover it; but Annex A of the specification gives an example for such a mapping.

Table 1 summarizes the tools with fUML or Alf support we are aware of. The following is a short description of the fUML and Alf tools in Table 1.

fUML Reference Implementation [9] The Reference Implementation was created along the draft specification to validate the semantics and to help further tool development. The reference implementation is a console Java application that takes a model in XMI as an input, executes an activity given as a parameter, and provides a detailed trace about the execution. It was developed by Model Driven Solutions.

Moka [14] Moka is an incubation component of the open source modeling environment Papyrus. It contains an fUML execution engine, and is able to animate and debug UML activities. Moreover, it includes a PSCS-based [10] engine. Breakpoints can be set in the code, and the activity can be started with an Eclipse debug configuration. The execution is animated (object and control flow). Moka is developed by CEA List.

Moliz [7] The moliz project is concerned with the execution, testing and debugging of UML models. It provides an fUML execution engine with step by step simulation and breakpoint support. Extension projects based on the Moliz engine provide non-functional analysis or testing of models. Moliz is developed by the BIG group of Vienna University, and supported by LieberLieber Software GmbH (a company developing extensions to Sparx's Enterprise Architect).

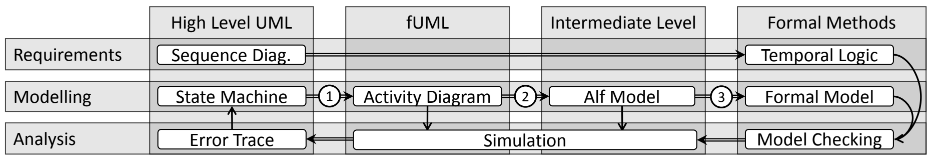


Fig. 1: Overview of the modeling and analysis approach

Alf Reference Implementation [8] The Alf Open Source implementation is a reference implementation similar to the fUML Reference Implementation. It handles the full Alf syntax at the "extended compliance level", which level is specified in the Alf language specification. It uses the fUML Reference Implementation execution engine to provide an interpreter for Alf code units. It is provided as a console Java application and was developed by Model Driven Solutions.

Alf editor in Papyrus [13] The goal of this component is to offer a textual editor in Papyrus to specify operations and activities in the Alf language. The editor would offer syntax highlighting and validation of Alf code. However, the current implementation is still in development with several limitations.

2.2 The Proposed Modeling and Analysis Approach

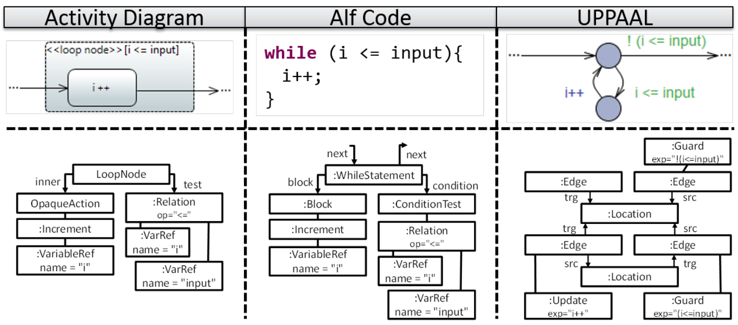
Fig. 1 presents how fUML and Alf fit into the approach sketched above. The behavioral model is first defined by UML state machines, which are then transformed to an fUML activity diagram (Step 1). An fUML model can be simulated and debugged then with one of the existing fUML execution engines.

We propose to use Alf as an intermediate modeling formalism. fUML constructs can be directly mapped to Alf languages elements. (Note we use "Alf model" to refer to the parsed, abstract syntax representation, and "Alf code" to the concrete textual syntax). This transformation is depicted as Step 2 on Fig. 1, which takes a model containing class definitions and activity diagrams enriched with Alf code for the definition of the basic actions, and transforms the model to a full Alf description. Then Step 3 starts from this Alf model and transforms it to the target formal modeling language of Uppaal (after abstracting from some details irrelevant from an analysis point of view).

Up to now, we created a prototype that implemented Step 2 and 3 using open source technologies to identify the open issues. The prototype was developed in Eclipse over models represented in EMF. Papyrus was used for handling UML models. A crutial part in the approach is to implement the transformations between the different models, which will be elaborated in the next section.

3 Transformation Technologies

A model transformation (MT) defines a mapping from a source language to a target language. Model transformations are specified in a language which is



```
while (i <= input){
  i++;
}
```

Fig. 2: Transformation chain example: from Papyrus activity diagram through Alf specification to Uppaal real-time system model

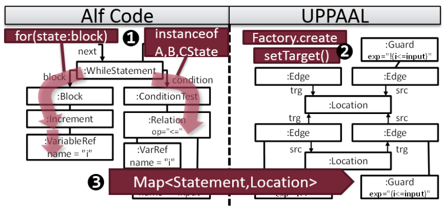
executed by a transformation engine on a source model to create a target model. A complex MT can be split into multiple steps to form a transformation chain.

Figure 2 shows a simple iterator example of the transformation of Step 2 and 3 described in Figure 1 with concrete syntax in the top row and model representation in the bottom row. Activity diagrams define guards and actions by Alf code, but the control flow is defined by explicit transition arrows and composite nodes. First, the activity diagram is translated to full Alf code, where the actions are mapped to statements and the flow is defined by structures (while statements). Then the Alf code is mapped to an Uppaal formal model where the statements are represented by atomic state transitions, and the control is managed by explicit edges with guard expressions.

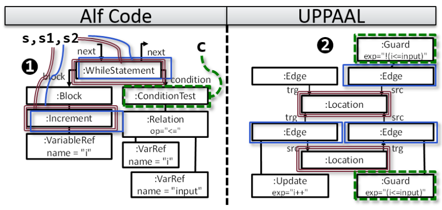
We show how this MT problem can be solved using different MT approaches using the Alf-to-Uppaal transformation. We also highlight the differences and the benefits of the techniques from the following aspects: source model processing, target model construction, trace handling and debugging possibilities.

Transformation program A first approach to implement a MT is to create a problem specific transformation program written in a general-purpose imperative programming language like Java. It is easy to start development without relying on any third-party MT tool. However, such MT programs are error-prone and hard to maintain resulting in a long development process.

Figure 3c shows the skeleton of an example Java transformation code, where both the source and target models are handled as arbitrary Java (EMF) objects. The source model is traversed by complex control structures, where navigation is available only through direct references (see Mark 1)in Figure 3a). The transformation code is responsible for finding the required relations in the model, i.e. possible state transitions in between Alf statements. Objects of the target EMF model are created via factory methods, attributes and references are defined by setter methods as in common Java objects (Mark 2)). Traceability management is ad hoc, and usually implemented with temporally maps (Mark 3)).



(a) Transformation program illustration



(b) Model transformation program run

```
Location transform(WhileStatement w) {
  for(Statement s: w.block.statements) {
    /* map the inner states */}
  for(Location l : locations) {
    /* Link the locations to a circle */}
  // Map the condition
  // Set the guards of the outgoing edges
}
```

```
w.block.increment.variableRef.name                    ❶
```

```
l = factory.createLocation();                         ❷
```

```
e.setSource(locationMap.get(
            edgeMap.get(w.condition)));               ❸
```

(c) Example Transformation program implemented in java

```
                  module Alf2UPPAAL; create OUT : Machine from IN : Block {

rule State2Location {                                rule Condition2Guard {            ❷ to = new objects
  from s: Alf!State                  ❶ from =          from c: Alf!Condition
  to   l: UPPAAL!Location}             objects +       to   positiveGuard: UPPAAL!Guard( location <- /*...*/
rule Transition2Edge {                 condition                                         expression <- /*...*/)
  from s1: Alf!State,                                       negativeGuard: UPPAAL!Guard( location <- /*...*/
       s2: Alf!State ( /* next state */ )                                                expression <- /*...*/)}}
  to   e:  UPPAAL!Edge( src <- /*...*/
                        trg <- /*...*/ )}
```

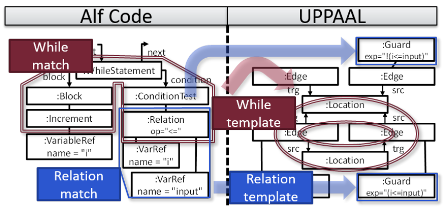
(d) Example model ATL transformation rules

Fig. 3: Illustration of transformation programs and model transformation tools

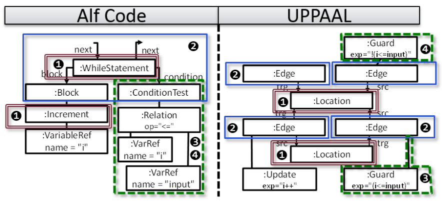
Model Transformation Tool There are several dedicated MT languages to specify transformation rules. Those rules are automatically executed by transformation engines on the source model to produce the output model. The transformation development process is scaling well with respect to the number of rules, However, there are also some constructs that are difficult to formulate in the transformation language, hard to integrate with other tools or transformation steps implemented in imperative code, and it has little debugging support.

The ATLAS Transformation Language (ATL) [1] provides a language to specify, and a toolkit to execute MTs. Figure 3d shows an example ATL code to map Alf specification to Uppaal model, which is illustrated in Figure 3b. A rule is defined by a `(from,to)` pair, where the `from` part declaratively selects objects from the source model instead of model traversal (at Mark 1) `s`, `s1` and `s2` are selected, where `s1` and `s2` have to be subsequent statements). The `to` part specifies the object to be created and the values of their attributes. During model generation, a trace model is automatically produced.

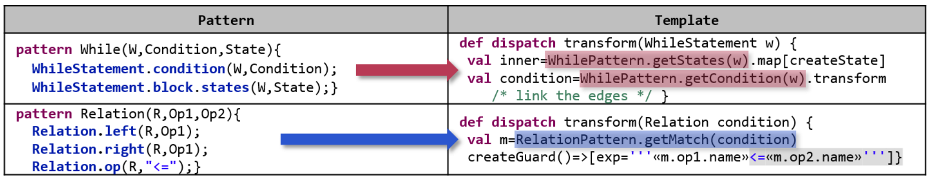
Model Queries and Functional Programs Declarative graph query frameworks like EMF-IncQuery[15] provide language for defining high level graph patterns



(a) Functional templates with model queries



(b) Query based model generation



Pattern	Template

`pattern While(W,Condition,State){`
`  WhileStatement.condition(W,Condition);`
`  WhileStatement.block.states(W,State);}`
	`def dispatch transform(WhileStatement w) {`
	` val inner=WhilePattern.getStates(w).map[createState]`
	` val condition=WhilePattern.getCondition(w).transform`
	`    /* link the edges */ }`

`pattern Relation(R,Op1,Op2){`
`  Relation.left(R,Op1);`
`  Relation.right(R,Op1);`
`  Relation.op(R,"<=");}`
	`def dispatch transform(Relation condition) {`
	` val m=RelationPattern.getMatch(condition)`
	` createGuard()=>[exp='''«m.op1.name»<=«m.op2.name»''']}`

(c) Example EMF-IncQuery patterns with Xtend functional program

Patterns on Source Metamodel	Trace	Generation Template on Target
❶ `pattern Statement(S)`		`Object(Location,l)` ❶

❷ `pattern nextState(From,To)`	`f = Statement(From) -> l`
	`t = Statement(To)   -> l`
		`Object(Edge,e)`
		`Reference(src,e,f)`
		`Reference(trg,e,t)` ❷

❸ `pattern PositiveWhileCondition(`
`StartLoop,Condition,Expression)`
	`incoming =`
	`nextState(Condition,StartLoop) -> e`
		`Object(Guard,g)`
		`Attribute(exp,g,"($Expression$)")`
		`Reference(guard,incoming,g)` ❸

❹ `pattern NegativeWhileCondition(`
`FinishLoop,Condition,Expression)`
	`outgoing =`
	`nextState(Condition,FinishLoop)-> e`
		`Object(Guard,g)`
		`Attribute(exp,g,"!($Expression$)")`
		`Reference(guard,outgoing,g)` ❹

(d) Query based views: Match the source model, create the target model

Fig. 4: Two model generation approaches based on EMF-IncQuery

and efficiently find their match in a model. Model queries provide an advanced language with high expressive power to navigate model structures with required properties, which can be evaluated on the source model. These queries can be used by the developer or a transformation program. The left column of Figure 4c shows two example patterns: the `While` pattern selects the while statements with its inner statements and loop conditions, the `Relation` lists the comparison expressions with the left and right operators. The left side of Figure 4a shows the matches on the example model.

Functional approaches proved their usefulness in the implementation of data transformations. Their main advantages are 1) compact representation 2) easy extendability and 3) composability. Template based code generation and model transformation widely uses functional approaches. The Xtend language adds functional elements to Java language like lambda expressions while keeps the imperative elements. The right column of Figure 4c shows the skeletons of the transformer functions: the first transforms a while statement, the second maps a relation condition.

Our experiences shows that a functional Xtend code using EMF-IncQuery queries is efficient for implementing model transformations. It can be used similarly to MT tools and it eliminates known navigation issues of Xtend. This approach allows the developer to 1) freely mix the two techniques (query the model from the template code, or use pure functions in a pattern), 2) directly call and debug templates and queries or 3) reuse rules by other transformations.

Query Based Views as Model Transformations IncQuery patterns can be used to define views to represent relations of the source model in a target model. Those query based view models are automatically and incrementally maintained. Views can be used as visualisation, or as model transformation. The visualiser aids the developer to correct bugs. A trace model is automatically derived.

Figure 4d shows four example for the use of query based views, and Figure 4b illustrate it with the running example, where the matches and their images are marked with Mark 1)- Mark 4). Mark 1) defines that each match of a `Statement` should be mapped to a new `l` Location in the Uppaal model. The trace of the mapping can be used in other rules, for example Mark 2) collects the subsequent statements from the control graph of the Alf model with the `nextState` pattern and creates an `e` edge between the images of the statements. Pattern Mark 3) gets the first statement of the cycle body and names it to `StartLoop` and the edge where the loop enters called `incoming`. Similarly Mark 4) selects the exiting statement first statement where the loop exits called `FinishLoop`, the edge is named outgoing. The model creation rules then maps the Alf loop conditions to Uppaal guard expressions and inserts it to the incoming and outgoing edges positively and negatively.

While query based views provide a restricted subclass of model transformations, this loss of generality is balanced by the fact that this class of model transformations is (source-)incremental by its nature, i.e. changes in the source model are propagated to the target model incrementally.

Summary From the above transformation approaches we choose the third one in our prototype implementation. EMF-IncQuery was used for the efficient and scalable queries of model elements. Xtend was used to map the source to the target model according to the defined EMF-IncQuery patterns. The lessons learnt during the development are summarized in the next section.

4 Experience and Open Issues

This section summarizes our experiences during the development of the previously presented transformation.

Experience with fUML and Alf:

– *No list of tools:* Although it seems trivial, but even collecting the available tools required considerable effort. This paper provides a snapshot, but it will become outdated. As fUML and Alf is still not yet widespread, OMG could promote them by listing the tools on its website along the specifications.

– *Few examples:* There are few example fUML models or Alf code available. Moreover, version and tool incompatibility further hinders the reuse of examples (e.g. a model exported in XMI could not always be loaded in an other tool, the UML specification changed in recent years, etc.).
– *Reference implementations:* The fUML and Alf reference implementations are fulfilling their purpose, they help to understand the specifications. Advanced functions are not in their scope, but they provide a good starting point for tool developers.
– *Semantic information:* Semantic information that could help the verification is not present in the Alf model (e.g. current compound state in a hierarchical or parallel activity). It can be added to the transformation tool as annotations.
– *Alf grammar:* The current version of Alf grammar uses complex structures of language elements to ensure unambiguity of the language. This structure is effectively handled by matching graph patterns during the transformation.

Open issues with the tools:

– In the released version of Papyrus, Alf code cannot be specified for the behavioral elements, thus we attached them as simple text in comment blocks.
– We had to create our own (limited) Alf metamodel for the transformations as the available ones were incomplete in the pre-Luna versions of the Eclipse-based tools.
– The tools would need a functionality to restrict the available elements (e.g. an Alf language element should not be used in a certain application domain).

Maturity of the tools:

– *Proven:* the Eclipse platform and its core services (e.g. EMF, Xtext) provided a solid base for tool development.
– *Incubation:* Although still in incubation phase, it was relatively easy to use EMF-IncQuery due to its easy-to-use tooling and examples. (Note, we consulted also with local developers of the tool).
– *Prototype:* The fUML engines (Moka, Moliz) are working and there are some examples and documentation available. However, they are still in prototypes phase (e.g. in a simple scenario we were able to produce NullReferenceException in Moka).
– *Alpha:* The Alf editor in Papyrus is still in active development (e.g. most of the code is in the sandbox repository, it supports only a limited part of Alf).

5 Conclusion

Support for the executable extensions of UML started to appear in open source modeling tools. This paper summarized these tools and our experiences with them, obtained during the development of a prototype transformation chain. Our findings showed that advancement in the tooling is promising, but there is still a long road ahead. As a summary, the questions raised in the introduction can be answered as follows:

1. There is a growing number of open source tools supporting fUML and Alf, they are summarized in Table 1.
2. These tools are mainly under development, their maturity level is summarised in Section 4.
3. There are several transformation approaches, which are summarized with their advantages and disadvanteges in Section 3.

Acknowledgments. This work was partially supported by Ericsson Hungary.

References

1. ATLAS Group: The ATLAS Transformation Language (2014), `http://www.eclipse.org/atl/`
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012)
3. Broy, M., Cengarle, M.: UML formal semantics: lessons learned. Software & Systems Modeling 10(4), 441–446 (2011)
4. Hu, Z., Shatz, S.: Explicit modeling of semantics associated with composite states in UML statecharts. Automated Software Engineering 13(4), 423–467 (2006)
5. Jouault, F., Tisi, M., Delatour, J.: fUML as an assembly language for MDA. In: Int. Workshop on Modeling in Software Engineering (MiSE). pp. 61–64 (2014)
6. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)
7. Mayerhofer, T., Langer, P.: Moliz: A model execution framework for UML models. In: Int. Master Class on Model-Driven Engineering: Modeling Wizards. pp. 3:1–3:2. MW '12, ACM (2012)
8. ModelDriven.org: Action language for UML (Alf) open source implementation (2014), `http://modeldriven.org/alf/`
9. ModelDriven.org: Foundational UML reference implementation (2014), `http://portal.modeldriven.org/project/foundationalUML`
10. Object Management Group: Precise semantics of UML composite structures RFP (2011), `http://www.omg.org/cgi-bin/doc?ad/11-12-07`
11. Object Management Group: Action Language for Foundational UML (Alf) (2013), formal/2013-09-01
12. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML) (2013), formal/2013-08-06
13. Papyrus: Alf support in Papyrus (2014), `http://wiki.eclipse.org/Papyrus/UserGuide/fUML_ALF`
14. Papyrus: Moka overview (2014), `http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`
15. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming (2014), in Press.