STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 3, 2013

FORMAL DEFINITION OF FUML IN $\mathbb{K}$-FRAMEWORK

SIMONA MOTOGNA, FLORIN CRĂCIUN, IOAN LĂZAR, BAZIL PÂRV [(1)]

Abstract. The Alf language was introduced as a simpler, textual definition of fUML executable models. The operational semantics of Alf is defined by mapping the Alf concrete syntax to the abstract syntax of fUML. The operational semantics of fUML is described in a semi-formal way, focusing on its implementation in Java. Our paper addresses two problems regarding this issue: i) semantic integration, namely semantics mappings should be defined using platform independent constructions, and ii) the correctness of the execution engine must be guaranteed. We propose an approach to give a formal definition of Alf in the $\mathbb{K}$-semantic framework. Executable $\mathbb{K}$-definitions will specify a reference virtual machine that can gain access to $\mathbb{K}$'s tools for formal analysis and verification.

1. Introduction

Software systems have recorded a spectacular evolution in the last years: they become more complex every day, and are used in a lot of domains. This evolution puts a lot of pressure on the development cycle of software systems, such that we would like to automate the development process as much as possible. Constructing software automatically from high-level models is one of the challenges in software engineering nowadays.

In this context, models should be easy to built, but should also encapsulate a complete and precise behavior description. An executable model, in addition, has an associated formal action semantics such that the model can be executed an tested in this early stage of development. fUML [15] is the OMG proposal for such an approach.

Received by the editors: April 15, 2013.

2010 *Mathematics Subject Classification.* 68N30, 68Q60.

1998 *CR Categories and Descriptors.* D.2.4 [**Software/Program Verification**]: Subtopic – *Validation*; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Subtopic – *Logics of programs*.

Key words and phrases. fUML, K-Framework, Formal Methods.

This paper has been presented at the International Conference KEPT2013: Knowledge Engineering Principles and Techniques, organized by Babeş-Bolyai University, Cluj-Napoca, July 5-7 2013.

In a visionary article, Harel & Marron [8] argue that the abstraction level provided by models and specifications will make them more expressive and intuitive and the focus in software systems development will shift to model construction, composition and adaptivity. Considering the important role of models in the description of the system's behavior, it is clear that such an approach must assure at least some essential attributes of the system, such as correctness, safety, completeness.

Formal methods represent the mathematical instrument that can be used to assure these attributes, as they offer the appropriate mechanism to detect if a model has errors or how they can be avoided. The main drawback in the use of such methods is the fact that they are difficult to be understood by software engineers.

The main goal of this paper is to propose an approach that will take advantage of the characteristics of formal methods in order to construct a complete semantical definition of Alf and to provide an execution engine for it. The tool that we have chosen is the $\mathbb{K}$-framework proposed in [19], based on its main features. The $\mathbb{K}$-framework provides the necessary constructions to define the Alf semantics and its features can be used for reaching our purpose [20]:

- executability: the definitions are directly executable in order to be experimented with and analysed;
- unique definition: there is only one definition for a language, and several analysis tools that are sound with respect to this definition;
- program logic: the framework serve as a program logic with which the programs can be verified and analysed.

The rest of the paper is organized as follows: the next section gives an short introduction to fUML, Alf and $\mathbb{K}$-framework discussing related work concerning these subjects. Section 3 is dedicated to the presentation of our approach to constructing a virtual machine for fUML, while Section 4 deals with conclusions and future research directions.

2. Background

2.1. **fUML and Alf.** Approaches in which modeling is at the core of the development activities also simplify the component construction process [5]. One of the main component based development's challenge is to provide a general, flexible and extensible model, for both components and software systems. This model should be language-independent, as well as programming-paradigm independent, allowing the reuse at design level. Well-known such approaches are based on UML and MDA.

MDA framework [12] provides an approach for specifying systems independently of a particular platform and for transforming the system specification

into one for a particular platform. The most important benefits are higher abstraction level in program specification and increase of automation in program development. The availability of such tools and the easiness of their use has contributed to the success of MDA. But development processes based on MDA are considered heavy-weight processes since they cannot deliver (incrementally) partial implementations to be executed as soon as possible.

In this context, executing UML models became a necessity for development processes based on extensive modeling. For such processes, models must act just like code, and UML 2 and its Action Semantics [13] provide a foundation to construct executable models. In order to make a model executable, it must contain a complete and precise behavior description. Unfortunately, creating such a model is a tedious task or an impossible one because of many UML semantic variation points. Executable UML [9] means an execution semantics for a subset of actions sufficient for computational completeness. Two basic elements are required for such subsets: an action language and an operational semantics. The action language specifies the elements that can be used while the operational semantics establishes how the elements can be placed in a model, and how the model can be interpreted. Again, creating reasonable sized executable UML models is difficult, because the UML primitives from the UML Action Semantics package are too low level.

The Executable Foundational UML (fUML [15]) is a computationally complete and compact subset of UML, designed to simplify the creation of executable UML models. The semantics of UML operations can be specified as programs written in fUML. The fUML standard provides a simplified subset of UML Action Semantics package (abstract syntax) for creating executable UML models. It also simplifies the context to which the actions need to apply. For instance, the structure of the model will consist of packages, classes, properties, operations and associations, while the interfaces and association classes are not included.

The complete static and operational semantics of fUML is still in its early stages, and although several proposal have been issued, this problem is still open. In our opinion, the difficulties risen in the complete semantical definition of fUML lie in the following three aspects:

- The fUML standard enforces a data flow abstract representation for the behavior of the methods. For example, accessing the values of parameters or variables from certain reserved locations, the values (or the references) of these types of entities will flow as tokens on edges. If a parameter or variable is used in multiple places, its value is copied using a fork node and sent to each action that needs it. To assign a

new value to the entity, a new point that provides the value is created (with a new fork node).

- The number of fUML constructs and the relations between them; most of the research carried in this field concentrate on a subset of fUML, especially on actions and activities, but the integration of all syntactical elements in the formal specification is not an easy task.
- fUML suffers from the same problem as several well-known programming languages, namely they are not designed or analyzed using formal semantics. The current implementation of fUML uses a Java virtual machine, and the correctness of the semantical constructions and their corresponding behavior cannot be guaranteed. In a formal environment, using a mathematical mechanism we can prove different properties regarding model execution.

The language Alf has been adopted as Action Language for fUML in 2010 [14] providing a concrete syntax for describing fUML models. Alf semantics is mapped to fUML abstract syntax metamodel. Alf syntax is inspired from well known programming languages such as C++ and Java, and "acts as the surface notation for specifying executable behaviors within a wider model that is primarily represented using the usual graphical notations of UML" [14]. In this way, creating reasonable sized executable UML models is much easier based on Alf constructs, instead of low level UML primitives.

There has been a lot of reaserch in the field of verification an semantic specification for fUML and Alf, based on diverse mathematical mechanisms. Relevant results have been obtained in using CSP (Communication Sequential Processes) [3, 1, 2], considering a lightweight verification method for strong executability [18, 17], or using Petri Nets [22]. However, all these approaches fail to offer a complete specification, since they impose restrictions on used UML diagrams and notations. The fUML virtual machine is under investigation in [21].

The execution for fUML activities is concurent, the nodes within activities may be executed concurrently according to the control and data flow model defined by the UML specification. Figure 1 (a) shows an fUML activity fragment which computes the value 7. According to fUML, an action may begin execution when it has been offered control tokens on all its incoming control flows and all its input pins have been supplied (via object flows) object tokens sufficient for their multiplicities. Figure 1 (a) does not contain control flows. When this fragment is executed, three control tokens are offered for the value specification actions, because these actions do not have incoming control flows. These actions may be executed concurrently and they provide the values 1, 2, respectively 3 to the other actions. Because the next two call behavior actions

(+) require the object token offered by the fork node a, they will wait until that token will be offered by that fork node (synchronization mechanism). When the fork a offers the object token, the next two + actions may be executed concurrently and their results will be offered (via the fork nodes b and c) to the last + action which will provide the final value.

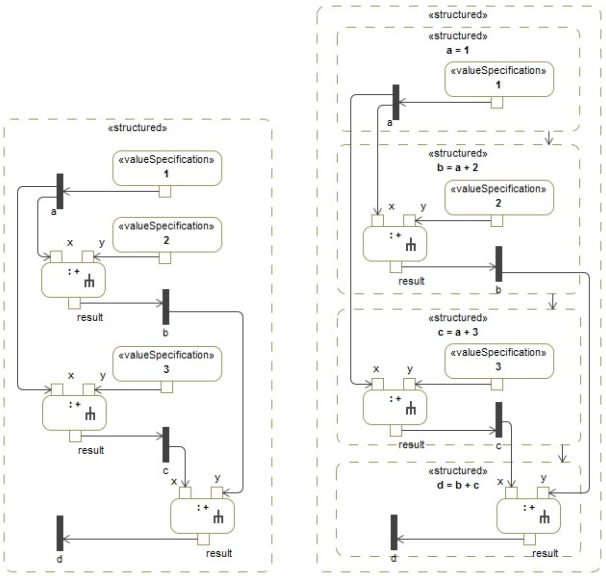


FIGURE 1. (a) fUML activity fragment; (b) Alf to fUML mapped activity fragment.

Using a textual notation, the activity fragment from Figure 1 (a) may be written as shown in Figure 2 (a). Taking into account the concurent execution semantics of fUML, after a is computed, the statements *2* and *3* may be executed in parallel, then d will be computed after the previous executions have been completed. We may notice that the statements presented in Figure 2 (a) are common to the languages that use a sequential model of computation. So, if this textual representation would be compiled into a parallel fUML representation without explicit mechanisms for specifying the concurrency aspects, then it would be difficult to write and control the concurrency aspects.

```
a = 1;        //Statement 1
b = a + 2;    //Statement 2
c = a + 3;    //Statement 3
d = b + c;    //Statement 4
```

```
a = 1;        //Statement 1
//@parallel
{b = a + 2;   //Statement 2
c = a + 3; }  //Statement 3
d = b + c;    //Statement 4
```

FIGURE 2. (a) Block of statements written using a textual notation; (b) Alf textual notation

Alf defines a textual representation inspired from such languages, but also adds features which allows users to write parallel programs using very simple textual constructs mapped to the powerful abstract syntax of fUML. For example, Figure 1 (b) shows the fUML activity corresponding to the textual representation from Figure 2 (a). Because the textual representation contains a sequence of statements (sequential model of computation), all statements are mapped to *structured* activities having control flows between them. But, the control flow introduced between the structured activities corresponding to the statements *2* and *3* will prevent the concurrent execution of these two statements.

In order to indicate parallel execution, Alf provides the *parallel* annotation. Figure 2 (b) explicitly shows that the statements *2* and *3* must be executed in parallel, and after both will be completed, statement *4* will be executed. The compiled fUML model of this textual notation contains a *structured* activity containing the compiled statements *2* and *3*, but without control flows between them.

Alf allows us to use the *parallel* annotation for the block written in Figure 2 (a). In this case, the compiled model would be the model from Figure 1 (b) but without control flows between structured activities. The synchronization between actions follows the control and data flow model. (There are some restrictions for parallel blocks, e.g. a local name used in the lhs of assignments may be part of only one assignment.)

```
a = 1; //Statement 1
//@parallel
{
    b = a + 2; //Statement 2
    c = a + 3; //Statement 3
}
d = b + c; //Statement 4
```

FIGURE 3. Alf textual notation

2.2. **K-framework.** Developed as a collaborative effort between several research groups, $\mathbb{K}$ is a rewrite-based framework supporting definition and execution of programming languages. The $\mathbb{K}$-semantics can be executed and tested, and the underlying matching logic and language Maude can be used for program analysis and verification.

A $\mathbb{K}$ definition consists of configurations, computations and rules. Several notable results, such as the formalization of C [6], Scheme [10], and Verilog [11], type checking [7] and symbolic execution [4] have contributed to the confidence in the $\mathbb{K}$-framework capabilities. Recent approaches [20] suggest that $\mathbb{K}$ may be a suitable tool for formalization and analysis of fUML and Alf.

3. Virtual Machine for fUML

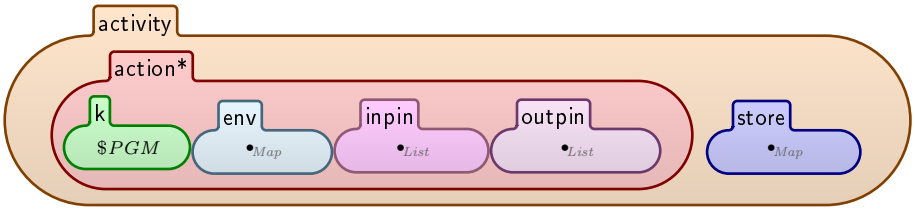
In this section we illustrate our formalization of fUML in K-framework. The fUML standard includes class and activity diagrams to describe a system's structure and behaviour respectively. In this paper we mainly focus on the formalization of the activity diagrams. A formalization of class diagrams has already been given in [20] and can be later integrated with our current approach.

We start with the syntax of ALF simple expressions which are going to be mapped to actions nodes in fUML.

```
MODULE ALF-SYNTAX
  SYNTAX  AExp ::= Id
                 | Int
                 | AExp + AExp
                 | AExp * AExp
  SYNTAX  Ids ::= List{Id,","}
  SYNTAX  Stmt ::= Id = AExp
                 | Stmt ; Stmt
  SYNTAX  AExps ::= List{AExp,","}
END MODULE
```

In order to define the K-semantics of fUML we introduce the main K-configuration. A configuration consists of a pool of actions ($cell_{action^*}$) which forms an activity ($cell_{activity}$). Each action has a list of input ($cell_{inpin}$) and output pins ($cell_{outpin}$) and the program that is going to be executed ($cell_K$). Input and output pins are named. Action pins names are mapped to global names through the action environment ($cell_{env}$). The global names are kept in an activity store ($cell_{store}$). The activity store allows the actions to communicate in a asynchronous way. An action can read an input pin only if its name is not mapped to an undefined value ($\perp$). After the action execution is completed the action output pins are mapped to values. For the moment we do not make a distinction between data and control pins (since they can have the same formalization). The action nodes are executed concurrently.

CONFIGURATION:



ALF program is stored in one of the action nodes and is translated into one or more fUML action nodes. Note that ALF action nodes and fUML action nodes are treated in a similar way and their execution runs in parallel.

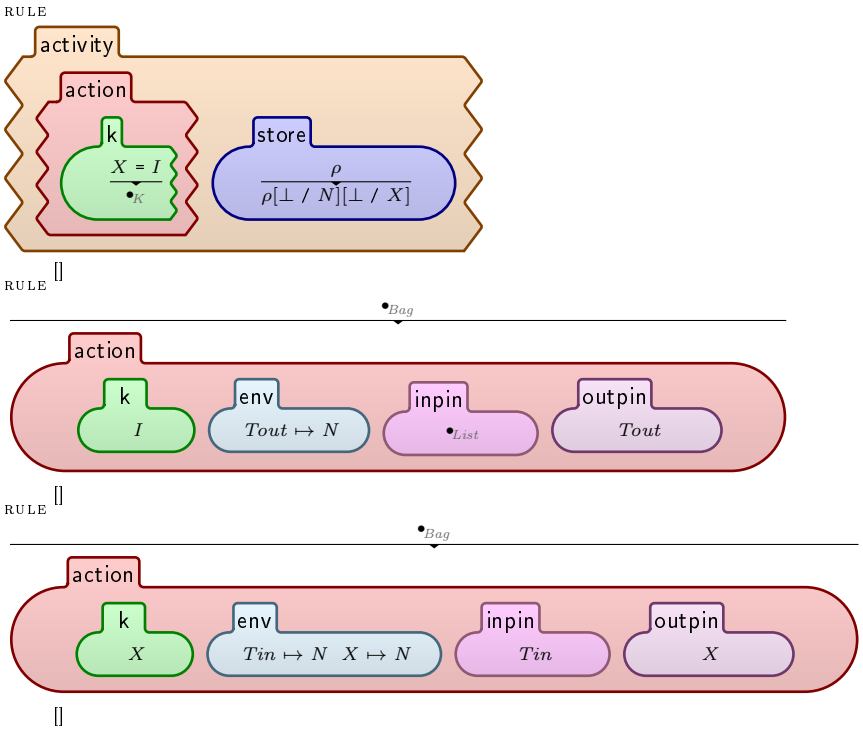
The execution of an ALF action node generates a fUML activity diagram (a pool of interconnected fUML action nodes), while the execution of the fUML action nodes do the *model execution* namely the propagation of object flow and control flow through the fUML model.

We adopted a small-step operational semantics for the translation of ALF to fUML. Therefore we introduced the following intermediate operations to be executed inside a fUML action node (in $cell_K$): *read* to allow the fUML action to wait for the input pins (data and control input flow) to have valid values; *write* and *writeVar* to propagate the values on the output pins (data and control output flow); *lookup* to find the value assigned to a global name which usually denotes a variable name or an input pin name; and *callBehaviour* to invoke a fUML behaviour (For simplicity in this paper we consider only the simple arithmetic operations to illustrate the concepts of our formalization). The undefined value $\perp$ is used to model the fUML diagram edges on which data or control flow has not been propagated yet. Thus the activity global names (corresponding to fUML activity edges) are mapped to $\perp$ in the activity store.

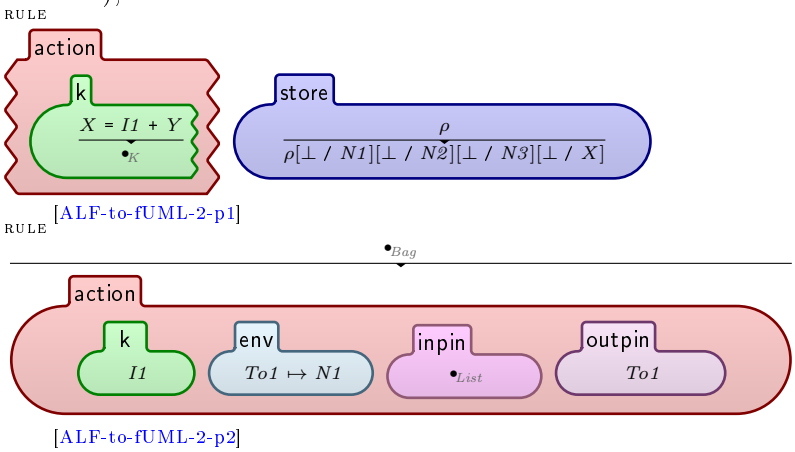
SYNTAX $K ::=$ `read` (Ids)
$\mid$ `write` (Int, Ids)
$\mid$ `writeVar` (Id)
$\mid$ `callBehaviour` $(+, Id, Id)$
$\mid$ `lookup` (Id)
$\mid$ $\perp$

The following rules illustrate the translation of ALF action nodes into fUML action nodes. The graphical presentation of the rules is directly generated by $\mathbb{K}$-framework compiler. Note that in this paper we split a rule on many lines due to the page width limitation such that the notations *p1...pn* denote the components of a splitted rule.

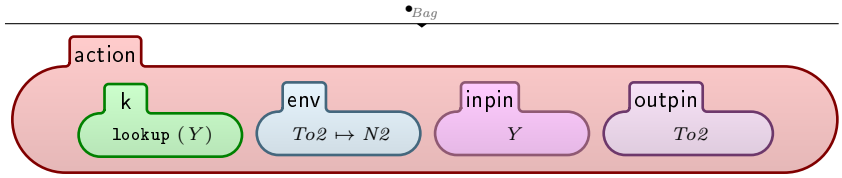
The first rule generates two new fUML actions from an ALF assignment $X = I$ (where X is a variable name and I is a constant) as follows: a value specification action (*ALF-to-fUML-1-p2*) and a variable action (*ALF-to-fUML-1-p3*). Note that new fUML action nodes are added to the existing action nodes in that activity. The rule assumes that inside the activity exists an ALF action node (*ALF-to-fUML-1-p1*) that has that assignment as the current instruction in its $cell_K$. The rule consumes that assignment of the ALF action node and let the ALF action $cell_K$ to continue with the next instruction. The fUML actions input and output pins are kept separately and are linked together through the local action environments and global activity store. Those two new generated fUML actions and ALF initial action are executed concurrently. fUML actions communicate in an asynchronous manner through the store variable N.



The following rule generates four new fUML action nodes from an ALF assignment $X = I1 + Y$ (where X is a variable name and $I1$ is a constant) as follows: a value specification action, a waiting actions (lookup for the variables valid values), one call behaviour action and a variable action.

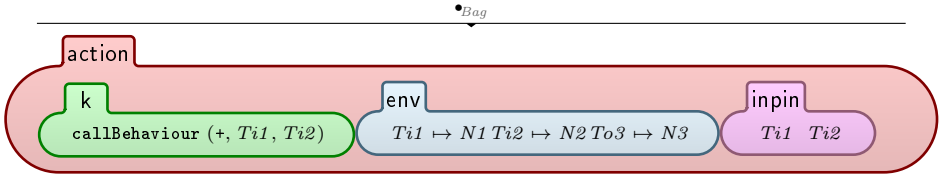


RULE



[ALF-to-fUML-2-p3]

RULE



[ALF-to-fUML-2-p4]

A fUML action node is destroyed when its executable code is completed (namely $\mathbb{K}$-cell is empty). We also provide rules for simplification of ALF expressions like the following:

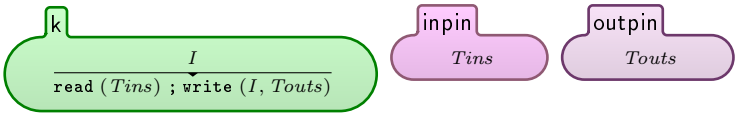
RULE

$$\frac{X \texttt{ = } E1 \texttt{ + } E2}{Y \texttt{ = } E1 \texttt{ ; } X \texttt{ = } Y \texttt{ + } E2}$$

when $E1$ =/=K $Id \wedge_{Bool} E1$ =/=K Int

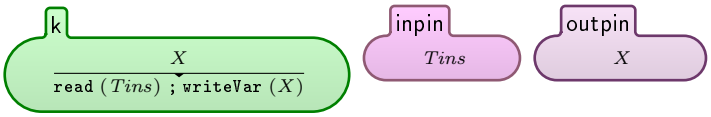
Next rules define the execution of the fUML node actions: value specification, variable and call behaviour. Each rule first waits for the valid values on its input pins and then writes an appropriate value on its output pins.

RULE



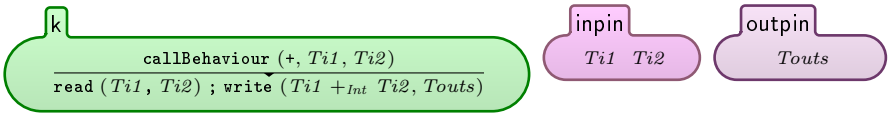
[fUML-Action-ValueSpec]

RULE



[fUML-Action-Var]

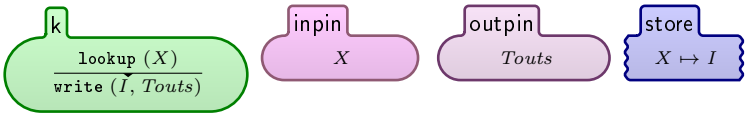
RULE



[fUML-Action-CallBehaviour]

The rule for *lookup* is waiting for a global variable to have a valid value in the store. When the assigned value is valid, that value is written on the output pins.

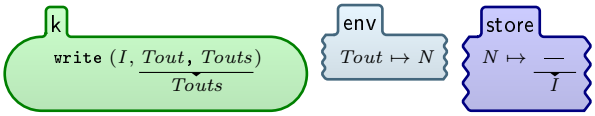
RULE



when $I =/=_{Int} \perp$
[fUML-Action-Lookup]

The next rules are writing appropriate valid values on the output pins.

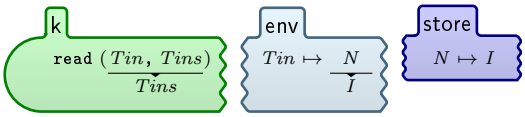
RULE



[fUML-Action-Set-Outpins]

The rules for *read* guarantee that all input pins have received valid values.

RULE



when $I =/=_{Int} \perp$
[fUML-Action-Wait-Inpins]

4. Conclusions and Future Work

We propose a novel formalization of ALF and fUML in $\mathbb{K}$-framework. The $\mathbb{K}$-framework allows us to directly define a concurrent semantics and to execute fUML models conform to OMG specification [15].

The main contributions of this paper are: i) the definition of $\mathbb{K}$-configurations corresponding to Alf syntax (fragment for simple arithmetic expressions) and ii) the specification of the $\mathbb{K}$-rules that simultaneously transform Alf artifacts into fUML constructs and execute them.

This paper is to be consider a proof of concepts for an approach of defining a virtual machine for Alf and fUML in the $\mathbb{K}$-framework, virtual machine that will be based on the executable $\mathbb{K}$-rules.

In contrast to the existing fUML virtual machine implemented in Java our approach is declarative and allows us to directly support a higher degree of genericity (specified by explicit semantics variation points in OMG specification [15]).

The main benefit consists in a higher level of platform and language independence, since the approach will not be based on a Java virtual machine, but on a more formal definition.

Another advantage is the extensibility. It is well known that Alf syntax, and implicitly its applicability, are quite restrictive [16]. Our approach may offer the necessary instrument to ease the extension of Alf for specific constructs.

Our future investigations will concentrate on a complete definition for Alf and fUML in $\mathbb{K}$. Next important step is to integrate class diagrams and OCL constraints [20] in our current proposal. We also plan to use the $\mathbb{K}$-framework tools to perform different forms of analysis and verification (e.g. inconsistency, deadlock free like in [3, 1]) on our $\mathbb{K}$-rules.

References

[1] Islam Abdelhalim, Steve Schneider & Helen Treharne (2011): *Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP*. In: *ICFEM*, pp. 33–48. Available at http://dx.doi.org/10.1007/978-3-642-24559-6_5.

[2] Islam Abdelhalim, Steve Schneider & Helen Treharne (2012): *An Optimization Approach for Effective Formalized fUML Model Checking*. In: *SEFM*, pp. 248–262. Available at http://dx.doi.org/10.1007/978-3-642-33826-7_17.

[3] Islam Abdelhalim, James Sharp, Steve A. Schneider & Helen Treharne (2010): *Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP*. In: *ICFEM*, pp. 371–387. Available at http://dx.doi.org/10.1007/978-3-642-16901-4_25.

[4] Irina Mariuca Asavoae, Mihail Asavoae & Dorel Lucanu (2010): *Path Directed Symbolic Execution in the K Framework*. In: *SYNASC*, IEEE Computer Society, pp. 133–141. Available at http://synasc10.info.uvt.ro/[SYNASC].

[5] Krishnakumar Balasubramanian, Aniruddha S. Gokhale, Gabor Karsai, Janos Sztipanovits & Sandeep Neema (2006): *Developing Applications Using Model-Driven Design Environments*. *IEEE Computer* 39(2), pp. 33–40. Available at http://doi.ieeecomputersociety.org/10.1109/MC.2006.54.

[6] Chucky Ellison & Grigore Roşu (2012): *An Executable Formal Semantics of C with Applications*. In: *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, ACM, pp. 533–544, doi:10.1145/2103656.2103719.

[7] Chucky Ellison, Traian Florin Şerbănuţă & Grigore Roşu (2009): *A Rewriting Logic Approach to Type Inference*. In: *Recent Trends in Algebraic Development Techniques*, *Lecture Notes in Computer Science* 5486, Springer, pp. 135–151, doi:doi:10.1007/978-3-642-03429-9. Available at http://dx.doi.org/10.1007/978-3-642-03429-9. Revised Selected Papers from the 19th International Workshop on Algebraic Development Techniques (WADT'08).

[8] David Harel & Assaf Marron (2012): *The quest for runware: on compositional, executable and intuitive models*. *Software and System Modeling* 11(4), pp. 599–608. Available at http://dx.doi.org/10.1007/s10270-012-0258-8.

[9] Stephen J. Mellor & Marc Balcer (2002): *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Patrick Meredith, Mark Hills & Grigore Roşu (2007): *A K Definition of Scheme*. Technical Report Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign.

[11] Patrick O'Neil Meredith, Michael Katelman, José Meseguer & Grigore Roşu (2010): *A Formal Executable Semantics of Verilog.* In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, IEEE, pp. 179–188, doi:doi:10.1109/MEMCOD.2010.555863.

[12] OMG (2003): *MDA Guide Version 1.0.1.* . Available at http://www.enterprise-architecture.info/Images/MDA/MDA20Guide20v1-0-1.pdf.

[13] OMG (2009): *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.2.* Available at http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/.

[14] OMG (2010): *Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF).* Available at http://www.omg.org/spec/ALF/1.0/Beta1.

[15] OMG (2011): *Semantics of a Foundational Subset for Executable UML Models (FUML).* Available at http://www.omg.org/spec/FUML/Current.

[16] Isabelle Perseil (2011): *ALF formal.* *ISSE* 7(4), pp. 325–326. Available at http://dx.doi.org/10.1007/s11334-011-0168-x.

[17] Elena Planas, Jordi Cabot & Cristina Gómez (2011): *Lightweight Verification of Executable Models.* In: *ER*, pp. 467–475. Available at http://dx.doi.org/10.1007/978-3-642-24606-7_37.

[18] Elena Planas, David Sanchez-Mendoza, Jordi Cabot & Cristina Gómez (2012): *Alf-Verifier: An Eclipse Plugin for Verifying Alf/UML Executable Models.* In: *ER Workshops*, pp. 378–382.

[19] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework.* *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012. Available at http://dx.doi.org/10.1016/j.jlap.2010.03.012.

[20] Vlad Rusu & Dorel Lucanu (2011): *A K-Based Formal Framework for Domain-Specific Modelling Languages.* In: *FoVeOOS*, pp. 214–231. Available at http://dx.doi.org/10.1007/978-3-642-31762-0_14.

[21] P. Langer T. Mayerhofer & G. Kappel (2012): *A Runtime Model for fUML.* Available at http://publik.tuwien.ac.at/files/PubDat_210111.pdf.

[22] Yann Thierry-Mieg & Lom-Messan Hillah (2008): *UML behavioral consistency checking using instantiable Petri nets.* *ISSE* 4(3), pp. 293–300. Available at http://dx.doi.org/10.1007/s11334-008-0065-0.

(1) Babeş Bolyai University
E-mail address: {motogna,craciunf,ilazar,bparv}@cs.ubbcluj.ro