Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. V (2010), No. 5, pp. 775-782

Tool Support for fUML Models

C.-L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, I.-G. Czibula

Codruţ-Lucian Lazăr, Ioan Lazăr, Bazil Pârv,
Simona Motogna and István-Gergely Czibula
Department of Computer Science
Babeş-Bolyai University, Cluj-Napoca, Romania
Romania, 400084 Cluj-Napoca, 1 M. Kogălniceanu
E-mail: {clazar, ilazar, bparv, smotogna, istvanc}@cs.ubbcluj.ro

Abstract: In this paper we present a tool chain that aids in the construction of executable UML models according to the new Foundational UML (fUML) standard. These executable models can be constructed and tested in the modeling phase, and code can be generated from them towards different platforms. The fUML standard is currently built and promoted by OMG for building executable UML models. The compatibility of the executable models with the fUML standard means that only the UML elements allowed by fUML should be used for the abstract syntax and the extra constraints imposed by the fUML standard should be considered. The tool chain we propose is intregrated with the existing UML tools of Eclipse modeling infrastructure.
Keywords: Class Diagram, fUML, Action Language, Code Generation, Eclipse.

1 Introduction

The executable models are models that can be executed and tested without having to generate code from them and test them in a specific platform. Creating executable models in the process of developing an application is considered to be a good approach, because the business model and functionality can be implemented in the modeling phase, while the decisions regarding the implementation in the specific platform can be delayed to the phase for code generation from the model. The executable models have the advantage of not being poluted with code that is not related to the business logic, keeping the model and functionality much more compact and clear. And they also have the advantage of not being tied to a specific platform or technology.

The Foundational UML (fUML) [1] is a computationally complete and compact subset of UML [2], designed to simplify the creation of executable UML models. The semantics of UML operations can be specified as programs written in fUML. We introduced in a previous paper [3] an action language based on fUML, with a concrete syntax that follows the principles of the structured programming, which is supported by the modern languages like Java and C++.

In this paper we describe a tool chain that is aimed at building and testing executable UML models, as well as generating code towards different target platforms. The generated code is meant to be complete, with no code placeholders for the developer to fill out.

The tools are built on top of the Eclipse Modeling Framework Project (EMF) and some other projects from Eclipse that are part of the Eclipse Modeling Tools distribution. These tools are integrated with the Eclipse modeling infrastructure and with each other.

The remainder of the paper is organized as follows: section 2 presents the infrastructure needed to build executable models and section 3 presents the research problem. Then, section 4 describes the tool chain we propose. Section 5 presents the existing work related to ours and section 6 gives the conclusions of this paper.

2 Background

The UML Class Diagrams are widely used to create the structure of a model. They are intuitive and easy to use. However, the UML behavior diagrams (Activity Diagrams and State Machines) are not easy to use for larger models. 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 may be applied. 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.

However, creating executable fUML models is difficult, because the UML primitives intended for execution are too low level, making the process of creating reasonable sized executable UML models close to impossible.

A concrete textual syntax is needed, because it enforces a certain way of constructing models. This means that a lot of elements that need to be created explicitly in the graphical UML Activity Diagram can be implicitly derived from the syntax and created automatically.

OMG issued an RFP for a concrete syntax for an action language based on fUML [4]. Because there is no standardized action language at this moment, we proposed an action language of our own [3] as part of our framework: ComDeValCo (Framework for Software Component Definition, Validation, and Composition) [5–7].

The fUML standard also specifies how to create a virtual machine that can execute fUML executable models.

To generate code from MOF compliant models, OMG created MOF Model to Text Transformation Language [8], which is suitable to generate code from fUML models.

3 Research problem

The research problem is to investigate the creation of a tool chain that aids in the construction of fUML models that can be created and tested in the modeling phase, and from which code can be generated towards different platforms.

The tehniques mentioned below represent the core of a tool chain that can change the development experience for the better, by simplifying the process and allowing the developer to take decisions in the proper stage of development.

Model creation. To create the structure of the model, the usual UML Class Diagrams should be used. These Class Diagrams, however, must restrict the elements that can be used to those included in the fUML standard.

To create the part of the model corresponding to the behavior of the operations, an Action Language based on fUML needs to be used. This is because it is close to impossible to use UML Activity Diagrams for this task, as the user will need to create, configure and relate too many elements.

The Class Diagram editor needs to be integrated with the Action Language textual editor, which is used to create the behavior for each operation. The integration refers to the ability to select a class or an operation from the model and, with a simple action (double-click or some key shortcut), the action language editor for the behaviors can be opened and used for the selected element. The action language editor must be able to load the existing behavior under the selected element and display it properly. Also, on save action, it should properly update the model under the selected element.

An action language that follows the principles of the structured programming is to use by many programmers familiar with structured programming languages. Thus, we consider this

an important factor in choosing the action language, even though the fUML standard allows non-structured control flow.

Because many programmers are familiar with object-oriented languages like Java or C++, having a similar syntax will make the language much easier to be used. Also, the effort for learning the new language will be very much reduced. The action language will need to create the abstract representation for the statements and control structures as provided by the programming languages mentioned above, it must support complex expressions and easy access to parameters and variables.

Model execution. After creating the model, or parts of it, there is a need to simulate the execution from certain starting points. This can be achieved with a virtual machine that knows how to execute fUML models. This means that it can execute the activities built with the action language editor described above.

Code generation. After creating and testing the model, the only thing left is to generate code to a specific platform. A code generation tool should take into considerations any tags (stereotypes) placed on the model elements, and adjust the code generation process accordingly.

If the resulted action language has a well structured abstract representation, it should be possible to apply model transformations and generate code to structured programming languages with little effort. Thus, the executable models for which the behavior is created with such an action language can be converted to a multitude of platforms.

4 Tool Chain

In this section we present the tools we use to create fUML based executable models. The tools are integrated in the Eclipse modeling infrastructure. The UML Class Diagram Editor and the fUML Execution Framework exist and they only need to be integrated with the other tools. The fUML based Action Language Editor is a tool proposed by us. The Code Generation Utility exists, but the templates used to generate code are written by us.

We use a Point-of-Sale (POS) example model, presented in fig. 1. The example model consists of a `POS` class, which contains a list of `Products`. The user can make a new `Sale` (stored in `POS` as the `currentSale`) by invoking the `makeNewSale` operation and by adding `SaleItems` to the current sale in the form of product code and quantity. The `POS` component finds the `Product` associated with the given product code and, if present, passes the product and quantity to the `currentSale`. The `Sale` creates a new `SaleItem` instance and adds it to its list of sale items.

Fig. 2 and 3 present the implementation of the operations, as it is written using our action language. Fig. 4 shows the abstract representation in fUML of `Sale.addItem`'s behavior.

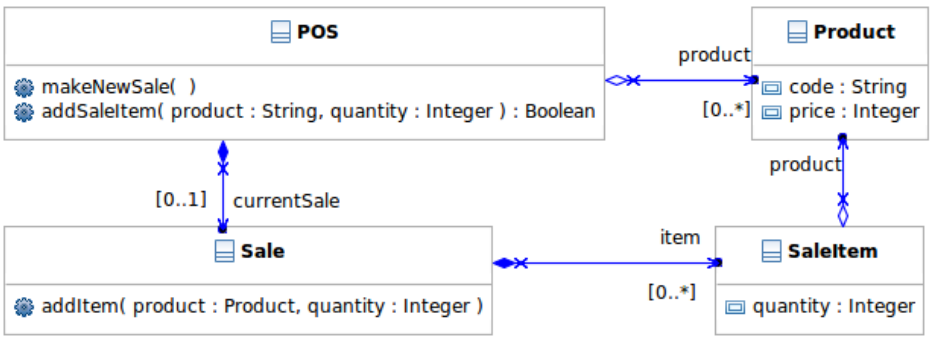

Figure 1: POS Example Model - Class Diagram Editor

4.1 UML Class Diagram Editor

The UML meta-model is provided by the Eclipse UML2 project [9], which is part of the larger Model Development Tools (MDT) project. UML2 is built on top of EMF(Core) [10] (which is part of the bigger Eclipse Modeling Framework project - EMF) with some adaptations, as UML has a structure that is not compatible with EMF.

The UML2 project provides the Java classes that correspond to the UML model, so that a UML model can be represented in Java. The model can be saved in files with the "uml"extension in XMI format, as it is standardized by OMG.

The UML2 project comes with a tree-based editor that allows the user to manipulate an "uml" file and build the UML models using a tree. This editor is not quite easy to use, but, for building only the structure of classes of a model, it is enough.

The Eclipse UML2 Tools [11], which is also part of MDT, provides a Class Diagram editor. This project allows the user to attach UML Class Diagrams to the UML models and build the UML models using a Class Diagram editor. The editor is quite mature and easy to use, and most programmers are used to building models using Class Diagrams, so this editor is the best choice for a tool to build the class structure of a model (fig. 1).

Because we want to build fUML based models, it is important that only the elements allowed by fUML are used. The Interface, for instance, was excluded from fUML, so the user should not include interfaces in the model. To simulate an interface, the programmer can use abstract classes with public abstract methods. A specialized editor for Class Diagrams that allows only fUML elements to be used, is considered for the future.

4.2 fUML based Action Language Editor

We introduced in a previous paper [3] an action language based on fUML, with a concrete syntax similar to the concrete syntax of the modern programming languages like Java or C++. This action language follows the principles of the structured programming.

We have also built an Eclipse textual editor for this action language using the Xtext project [12], which is the only remaining project from the bigger Textual Modeling Framework (TMF) project from Eclipse. This textual editor can take the textual representation for the functionality of an `Operation` (fig. 2 and 3) and convert it to an `Activity` with all the UML `Actions` and other elements necessary to provide the same functionality in UML (fig. 4). This `Activity` is added to the `Operation`'s classifier (of `Class` type) and set as the behavior of the `Operation`. Only the elements allowed by fUML are used to create this UML model of the `Activity`.

The textual editor for the Action Language is integrated with the Class Diagram editor, so that when the user wants to edit the behavior of an Operation, s/he only needs to double-click, or use a context menu item of the Operation, or use a key shortcut to open the textual editor. After the behavior is created, the user needs to save it by pressing Ctrl+S, at which point the main "uml" model file is updated to contain the model for the Operation's behavior.

4.3 fUML Execution Framework

The fUML standard specifies how a virtual machine for fUML models should work, and there is a reference implementation in progress from ModelDriven.org [13]. We managed to integrate this tool in the Eclipse workbench, so that we can test our fUML models. We can either pass the activities created with our action language, along with the needed parameters, directly to the execution framework to be executed, or we can write test activities with our action language and execute the tests.

```
public makeNewSale() {
    self.currentSale := new Sale;
}

public addSaleItem(code:String, quantity:Integer) :  Boolean {
    def product:Product := null;
    foreach (prod in self.product) {
        if (code = prod.code) {
            product := prod;
        }
    }
    if (product = null) {
        return false;
    } else {
        self.currentSale.addItem(product,quantity);
        return true;
    }
}
```

Figure 2: POS::makeNewSale and POS::addSaleItem Activity Concrete Syntax

4.4 Code Generation Utility

To generate code, we used the Acceleo project [14], which is part of the bigger Model To Text (M2T) project from Eclipse. This project implements the MOFM2T standard from OMG. It allows the user to create templates, which can later be used to generate code from the models.

In our case, these templates need to work on the elements included in fUML. Generating the structure of classes is straightforward. However, generating code for the behavior of the `Operations` is more complex, because the structure of the elements and the way the actions are connected needs to be considered. Because we took the decision to follow the structured programming principles, the UML model resulted for the `Activities` is well structured, so we are able to generate code with ease to languages like Java or C++. If we wouldn't have taken this decision, then the code generation step would have been a real problem, as fUML allows the user to create interactions between actions that might be impossible to represent in the languages mentioned above. Also, due to the resemblance in concrete syntax between the action language and the target languages, we are able to generate compact code in the target languages.

An important note is that the templates are specifically written for `Activities` constructed with an editor compatible with our Action Language. This is because the way the elements are structured and the way they interact is very important. If these aspects are not followed, the templates will produce a bad output.

For a fUML `Activity` that does not respect these constraints and which might be constructed using a different fUML based Action Language, a special set of templates need to be written. This is not necessarily an issue, because the Action Languages can be used in conjunction to construct UML models and because the templates only need to be written once. Our Action Language is a general purpose language and might be a bit hard to specify and build a proper editor for it. But a different Action Language that is specialized on a more concrete domain could have a simpler syntax and the code generation templates might be easier to be written.

Fig. 5 shows a snippet from the Acceleo templates we used to generate Java code. It shows how the statements are iterated, considering the `StructuredActivityNodes` that contain the

```
1  public addItem(product:Product, quantity:Integer) {
2      def newItem :  SaleItem := new SaleItem;
3      newItem.product := product;
4      newItem.quantity := quantity;
5      self.item.add(newItem);
6  }
```

Figure 3: Sale::addItem Activity Concrete Syntax

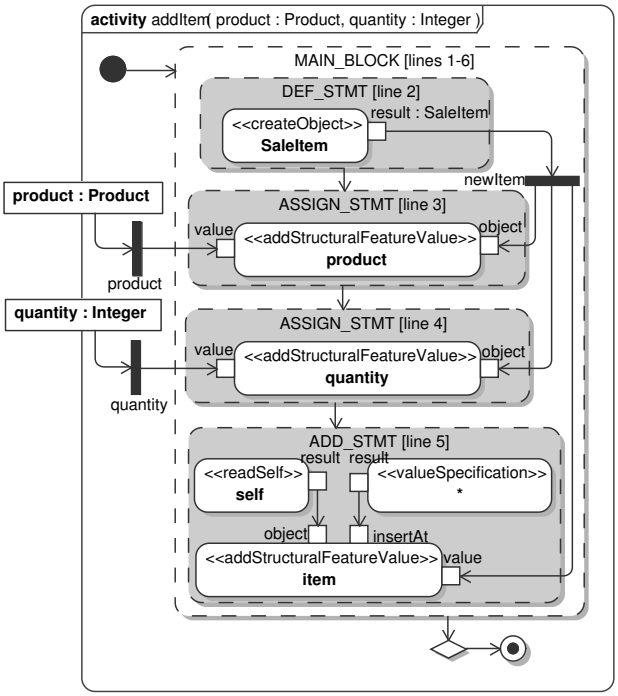

Figure 4: Sale::addItem Activity - fUML Abstract Syntax

actions for each statement, and the edges that enforce the sequential flow between these nodes.

The `operationActivity` template selects the only `StructuredActivityNode` it contains, which is the node containing all the statement nodes (*MAIN_BLOCK* node from fig. 4). The `activityBlock` template selects the statement node with no incoming edges, which represents the node corresponding to the first statement (*DEF_STMT [line 2]* from fig. 4).The `iterateBlockStatements` template prints the text for the statement node by calling the `blockStatement` template and, if the node has an outgoing edge to the next statement node, calls recursively the `iterateBlockStatements` for the next statement node.

5 Related Work

Only a few frameworks for building executable models exist and some of them are proprietary: Mentor Graphics' BridgePoint product with its Object Action Language (OAL) [15], Kennedy Carter's iUML product with its Action Specification Language (ASL) [16] and others. Some of these frameworks have their action languages based on the Action Semantics package from UML, but none is based strictly on fUML, as the standard is still in Beta version.

```
[template public operationActivity(a : Activity)]
[a.node->any(oclIsTypeOf(StructuredActivityNode)).oclAsType(StructuredActivityNode).activityBlock('    ')/]
[/template]

[template private activityBlock(blockNode : StructuredActivityNode, ident : String)]
{
[let firstNode : ActivityNode = blockNode.node->any(incoming->isEmpty())]
[blockNode.iterateBlockStatements(firstNode.oclAsType(StructuredActivityNode), ident)/][/let]
[ident/]}
[/template]

[template private iterateBlockStatements(parentNode : StructuredActivityNode,
                          stmtNode : StructuredActivityNode, ident : String)]
[stmtNode.blockStatement('    '.concat(ident))/]
[if (stmtNode.outgoing->notEmpty())]
[let nextNode : ActivityNode = stmtNode.outgoing->any(true).target]
[if (parentNode.node->includes(nextNode))]
[parentNode.iterateBlockStatements(nextNode.oclAsType(StructuredActivityNode), ident)/][/if]
[/let]
[/if]
[/template]
```

Figure 5: Acceleo Template Snippet for Java

6 Conclusions and Further Work

In this paper we presented a tool chain that can be used to create, execute and generate code from fUML executable models. The tool chain integrates an UML Class Diagram editor, a fUML Action Language editor, a fUML Execution Framework and a Code Generation Framework.

We plan to further work on this tool set, to better integrate the tools, in order to improve the user experience with the tool set as a whole. A specialized editor for Class Diagrams that allows only fUML elements to be used, is considered as future work. Also, we plan to investigate the possibilities of applying transformations to the models created by our action language, and to generate code in other programming languages.

Acknowledgment

This work was supported by the grant ID 546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

Bibliography

[1] *Semantics of a Foundational Subset for Executable UML Models*, Object Management Group Standard, Rev. 1.0, Beta 2, October 2009. [Online]. Available: `http://www.omg.org/spec/FUML/1.0/Beta2/PDF/`

[2] *UML Superstructure Specification*, Object Management Group Standard, Rev. 2.2, February 2009. [Online]. Available: `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`

[3] C.-L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, and I.-G. Czibula, "Using a fUML Action Language to construct UML models," *Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2009.

[4] *Concrete Syntax for a UML Action Language*, Object Management Group Request For Proposal, 2008. [Online]. Available: `http://www.omg.org/docs/ad/08-08-01.pdf`

[5] B. Pârv, S. Motogna, I. Lazăr, I.-G. Czibula, and C.-L. Lazăr, "ComDeValCo - a Framework for Software Component Definition, Validation, and Composition," *Studia Universitatis Babeş-Bolyai, Informatica*, vol. LII, no. 2, pp. 59–68, 2007.

[6] I. Lazăr, B. Pârv, S. Motogna, I.-G. Czibula, and C.-L. Lazăr, "An Agile MDA Approach for Executable UML Structured Activities," *Studia Universitatis Babeş-Bolyai, Informatica*, vol. LII, no. 2, pp. 101–114, 2007.

[7] C.-L. Lazăr and I. Lazăr, "On Simplifying the Construction of Executable UML Structured Activities," *Studia Universitatis Babeş-Bolyai, Informatica*, vol. LIII, no. 2, pp. 147–160, 2008.

[8] *MOF Model to Text Transformation Language (MOFM2T)*, Object Management Group Standard, Rev. 1.0, January 2008. [Online]. Available: `http://www.omg.org/spec/MOFM2T/1.0/PDF/`

[9] *UML2*, The Eclipse Foundation, 2010. [Online]. Available: `http://www.eclipse.org/modeling/mdt/?project=uml2`

[10] *Eclipse Modeling Framework (Core)*, The Eclipse Foundation, 2010. [Online]. Available: `http://www.eclipse.org/modeling/emf/?project=emf`

[11] *UML2 Tools*, The Eclipse Foundation, 2010. [Online]. Available: `http://www.eclipse.org/modeling/mdt/?project=uml2tools`

[12] *Xtext - a programming language framework*, The Eclipse Foundation, 2010. [Online]. Available: `http://www.eclipse.org/Xtext`

[13] *Foundational UML Reference Implementation*, ModelDriven.org, 2009. [Online]. Available: `http://portal.modeldriven.org/project/foundationalUML`

[14] *Acceleo*, The Eclipse Foundation, 2010. [Online]. Available: `http://www.eclipse.org/modeling/m2t/?project=acceleo`

[15] *Object Action Language Reference Manual*, Mentor Graphics, 2009. [Online]. Available: `http://www.mentor.com/products/sm/techpubs/object-action-language-reference-manual-38098`

[16] *UML ASL Reference Guide*, Kennedy Carter Limited, 2003. [Online]. Available: `http://www.ooatool.com/docs/ASL03.pdf`