Tool Paper: Combining Alf and UML in Modeling Tools – An Example with Papyrus –

Ed Seidewitz

Model Driven Solutions
14000 Gulliver's Trail
Bowie MD 20720 USA
ed-s@modeldriven.com

Jérémie Tatibouet

CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
P.C. 174, Gif-sur-Yvette, 91191, France
jeremie.tatibouet@cea.fr

Abstract. The Unified Modeling Language has been used largely in the software community to draw pictures for designing and documenting software written in other languages. The real executable semantics of a program are determined by the programming language, while the UML models themselves do not have a precise enough meaning to fully specify the executable functionality of the system being developed. Recently, however, there has been a great deal of work toward the standardization of precise, executable semantics for UML models – the "meaning" behind the pictures: Foundational UML (fUML) adopted by the Object Management Group in 2008, the Action Language for fUML (Alf) adopted in 2010, the recently completed Precise Semantics for UML Composite Structures (PSCS) and the Precise Semantics for UML State Machines (PSSM), now in progress. Together, these standards effectively provide a new combined graphical and textual language for precise, executable modeling. In particular, the Alf language goes beyond simply providing a textual "action language" for detailed behavioral code within graphical models, by including textual notation for fUML structural object-oriented modeling constructs (e.g., packages, classes, associations, etc.). This opens up the possibility of tooling allowing various parts of a UML model to be represented both graphically and textually (while preserving the same semantic level), with bidirectional synchronization between the two representations. This paper presents the achievement of an initial integration of UML and Alf in the context of the Papyrus tool for the specification of executable models.

1   Introduction

To most in the software community, "modeling" is something much different than "coding". On the other hand, there has also been a long standing minority in the software development community that has created models precise enough that they can be executed in their own right. Indeed, commercial executable modeling tools based on the Shlaer-Mellor method [5,6], Real-Time Object-Oriented Modeling (ROOM) [4] and Harel statecharts [2] all predated UML. However, such approaches converted over to UML notations,[1] and executable UML has been used for significant and critical applications, including fighter aircraft flight software, launch vehicle flight software and telecommunication switches [1,7].

Nevertheless, executable modeling has remained a niche approach dependent on divergent, proprietary tooling. One crucial issue with creating precise, standard UML models has been the imprecision of semantics specification in the UML standard. This issue was finally addressed with the adoption by OMG of the Foundational UML (fUML) specification[2]. This specification provides the first precise operational and axiomatic semantics for a Turing complete, executable subset of UML. The subset encompasses most of the object-oriented and activity/action modeling constructs of UML, which cover not only features commonly found in an object-oriented programming language, but also more advanced modeling features found in UML such as first-class associations and asynchronous signals.

But there has been a second crucial issue with executable UML modeling: the lack of a good surface notation for specifying detailed behavior and computation. UML is a largely graphical modeling language whose legacy is the unification of earlier graphical modeling languages. This is a great strength of UML for traditional, largely informal "big picture" analysis and design modeling, but it does not work well for representing detailed computations.

The fUML specification does not provide any new concrete surface syntax, tying the precise semantics solely to the existing abstract syntax model of UML. UML does provide a concrete notation for activities and actions that can be used to model, say, the method for an operation, but this requires one to draw a very detailed, graphical activity diagram.

This issue was addressed with the adoption by OMG of the Action Language for fUML (Alf)[3]. Alf is basically a textual notation for UML behaviors that can be attached to a UML model anyplace that a UML behavior can be. Together, these standards effectively provide a new combined graphical and textual language for precise, executable modeling.

Such a combination of graphical and textual notation is being implemented in practice in the Eclipse-based open-source UML/SysML modeling tool Papyrus[4]. In addition to the usual diagrams, the tool now provides the user with a textual editor sup-

---

[1]   See, for example, http://www.kc.com/XUML/ and http://www.xtuml.org/.
[2]   http://www.omg.org/spec/FUML/
[3]   http://www.omg.org/spec/ALF
[4]   https://eclipse.org/papyrus/

porting Alf. When developing an executable model, one can easily switch between the different editing views. The cohesion between the different views is ensured in a transparent way for the user.

This paper has two objectives. The first one is to introduce the reader to Alf both from syntactic and semantic standpoints. The second objective is to demonstrate the coupling between Alf and UML through an example built using the tooling integrated into Papyrus.

The remainder of this paper is organized as follows. Section 2 provides some additional background on Alf as a textual modeling language. Section 3 then introduces a simple example UML model, and Section 4 shows how this model can be updated with executable Alf code using Papyrus. Section 5 then makes some additional points about the synchronization of model changes occurring in different views. Section 6 identifies the limitations of the current Alf tooling and Section 7 concludes the paper.

2　Background

Semantically, Alf maps to the fUML subset. In this regard, one can think of fUML as effectively providing the "virtual machine" for the execution of the Alf language. However, this grounding in fUML also provides for seamless semantic integration with larger graphical UML models in which Alf text may be embedded. This avoids the semantic dissonance and non-standard conventions required if one where to instead, say, use a programming language like Java or C++ as a detailed action language within the context of an overall UML model.

However, the Alf language actually also includes a notation that goes beyond just behavioral modeling constructs. This additional textual notation includes all the *structural* modeling constructs included in the fUML subset. For example, suppose we have a UML class model that has an association between a `Customer` class and an `Account` class. This simple model can be represented textually in Alf:

```
package CustomerAccounts {
  public class Customer {
    public name : String;
  }

  public class Account {
    public balance : Integer;
  }

  public assoc CustomerAccount {
    public customer : Customer;
    public accounts : Account[*];
  }
}
```

Now, given a certain customer, we want to navigate across the association to the sum up the balances of all the customer's accounts. We can use Alf to define a UML activity to do this:

```
private import CustomerAccounts;
activity SumBalances(in customer : Customer) : Integer {
  totalBalance = 0;
  for (balance in customer.accounts.balance) {
      totalBalance += balance;
  }
  return totalBalance;
}
```

Syntactically, Alf looks at first much like a typical C/C++/Java legacy language. This is the result of a conscious compromise on the part of the submission team. Since, despite the issues involved, it is currently not uncommon practice to use Java or C++ as a UML action language, there was a strong desire to have a subset of Alf that would be familiar to such practitioners, to ease their transition to the new action language.

But the notational similarity can also be a bit deceptive. For example, association ends in UML are not semantically collection objects, but, rather, multi-valued properties with specified multiplicities (such as [*] used above, meaning "0 to many"). So, while `customer.accounts.balance` may look like a regular Java field access expression, what it really does is navigate from `cutomer`, across the association to the opposite `accounts` end, return all the `Account` objects at that end, and get the `balance` of each one. Alf adopts the notational convenience introduced in the already standard Object Constraint Language (OCL)[5] that navigation across an association to a multi-valued end automatically collects all the values at that end, so it is not necessary to have an explicit for loop to do this.

Note also that it is not necessary to explicitly declare the type of `totalBalance` or `balance`. The types of these local names are inferred from the result types of the expressions assigned to them – a convenience familiar to users of any modern scripting language.

Further, beyond simple syntactic conveniences, Alf also includes constructs that leverage the inherently concurrent, flow-oriented nature of the underlying fUML activity semantics. These include very powerful capabilities like filtering and mapping similar to those seen in many of the recently popular functional languages. So, for example, the body of `SumBalances` above can be more compactly written as:

```
return customer.accounts.balance->reduce '+';
```

Here, the functionality of an entire loop has been collapsed into a single expression, which maps directly to a UML reduce action. On an appropriate platform, this could be implemented as a highly concurrent operation, rather than as a sequential loop.

---

[5] http://www.omg.org/spec/OCL/

Further, suppose that the customer was to be selected based on email address from the extent of existing customers. This can be simply written:

```
myCustomer =
  Customer->select c (c.email == myCustomerEmail);
total = SumBalances(myCustomer);
```

The `select` notation here maps to a fUML parallel expansion region that, again, could be implemented as a highly concurrent search – or even translated into a database query. And, while the `=` looks like a traditional variable assignment, what it really maps to is a data flow in the underlying fUML activity – so local assignments do not actually introduce mutable state, which again allows much greater flexibility in the translation to implementation.

The point is that Alf provides an essentially complete notation for writing programs at the level of UML modeling semantics. Indeed, the open-source Alf Reference Implementation[6] is distributed un-integrated with any graphical tool, allowing executable models to be written completely textually in Alf in exactly this way. Nevertheless, a particular benefit of Alf is its close relationship to standard UML, which allows it to be integrated readily as a textual notation into existing graphical UML tools. But the availability of the extended Alf notation for structural modeling opens up the possibility of integration beyond just including Alf snippets for behavioral functionality within a graphical UML model. We turn next to consideration of tooling that realizes this possibility.

3 Papyrus support for Alf

Implementation of Alf in the context of Papyrus is the result of a collaboration between Model Driven Solutions (language implementation) and CEA LIST (integration in Papyrus). As shown in **Fig. 1**, this implementation is structured as a number of Eclipse plugins, which can be grouped into two parts, a "back end" and a "front end".

---

[6] http://alf.modeldriven.org
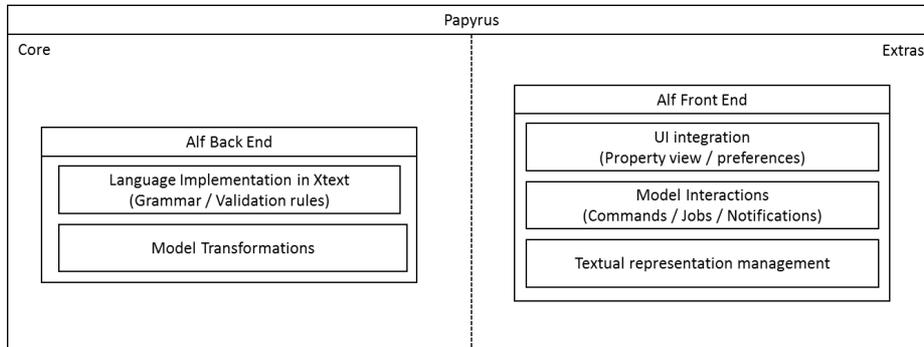
Fig. 1. Alf support architecture in Papyrus

The "back end" provides a complete implementation of the Alf language. The syntax of the language is defined as an Xtext[7] grammar. This grammar is used to parse Alf text into an Ecore metamodel based on the normative Alf abstract syntax. Semantic validation rules are given as OCL constraints that annotate this metamodel, which are executed during automatic validation as part of the Xtext framework. In addition, the "back end" also implements the mapping of the Alf abstract syntax to the UML abstract syntax using QVTo[8] transformations. Finally, a reverse QVTo transformation is also provided for mapping UML to the Alf abstract syntax, which may be then serialized to Alf text, in order to allow bidirectional synchronization between Alf and UML representations.

The "front end" enables the end user to use the Alf language implementation in the context of a UML model designed in Papyrus. It contributes to the property view and proposes an additional tab "ALF" (see, for example, **Fig. 3**) containing the required elements to specify and propagate Alf specifications entered by user in an existing UML model. The "front-end" handles all the interactions with the edited model through a set of commands and jobs. Additionally it provides experimental developments to maintain a synchronization between graphical and textual views of a single model.

The next section illustrates the use of Alf tooling integrated into Papyrus through an example.
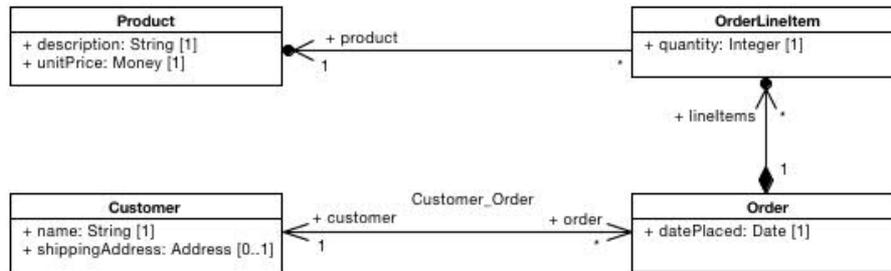
4 A Simple (but Representative) Example

To show how graphical/textual model integration can work in UML tooling, it is easiest to use an example. We will use an example from the domain of e-commerce, a simple model of an order. The first thing to understand is what information needs to be kept on an order and how this is related to information on the customer placing the

---

[7] https://eclipse.org/Xtext/documentation/
[8] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

order. This can be well represented using a UML class diagram, such as the one shown in **Fig. 2**.

Fig. 2. Order example class diagram

This diagram was entered graphically using Papyrus. It models an order as recording the date it was placed and having a set of line items, each of which specifies the quantity of a certain product included in the order. It also shows that an order is placed by a single customer, who may have many orders.

Models such as this are particularly useful in discussions with problem domain stakeholders. They are straightforward to understand and a lot of detail can be presented in a well-laid-out, compact diagram. For most people, this is far easier to understand than large blocks of text or written descriptions such as the previous paragraph.

Of course, there is also behavior associated with the classes shown in the diagram. Suppose, for instance, that you would like to add a `totalAmount` attribute to the `Order` class, along with an `addProduct` operation that adds a new line item for a given product and updates the `totalAmount` appropriately. To do this, the `addProduct` will use a new `getAmount` operation on `OrderLineItem.`

Rather than doing this by adding elements in multiple steps on the diagram, one can often specify them more efficiently by just typing text. We will show next how this can be done in Papyrus.

5   Adding Alf Text

In the context of Papyrus, the Alf editor is only available when you select a model element that is in the scope of fUML (i.e., a Class, a Package, a Signal, an, Enumeration, a Datatype, an Association or an Activity). As an example, if you click on the `OrderLineItem` class either on the diagram (cf. **Fig. 2**) or in the model explorer, the textual specification corresponding to this element is rendered in the editor, as illustrated in **Fig. 3**.
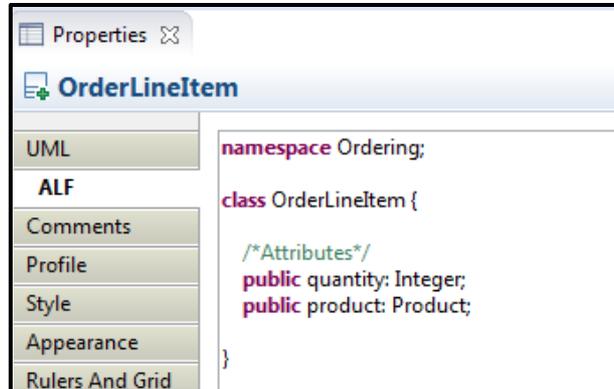
Fig. 3. Alf specification of `OrderLineItem`

Now you can simply type the new `getAmount` operation directly into this textual representation, as shown in **Fig. 4**.
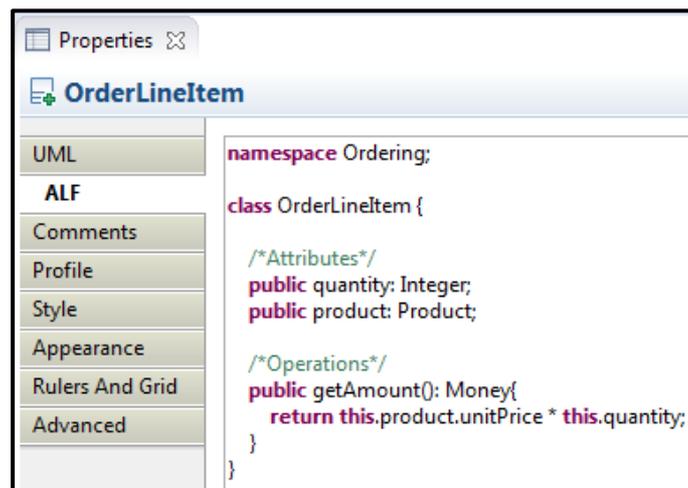

Fig. 4. A new operation with its implementation

As soon as something new is added to the Alf specification, you can *compile*[9] it, in order to propagate the changes to the UML model. The compilation feature is only available if the model is correct both syntactically and semantically. The validation process is triggered each time a modification is made in the specification.

---

[9] The compilation is a user-triggered operation that starts when the user clicks on the "compile" button available below the Alf editor. The compilation consists in taking an Alf specification and building the corresponding fUML model.

In the current example, the result of the compilation is that a new operation is added to the class `OrderLineItem.` After compilation, the textual specification is updated, as shown in **Fig. 5**.
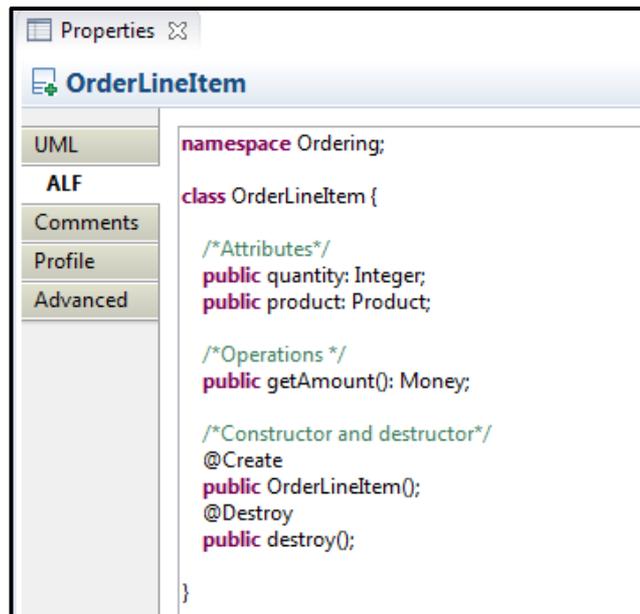

Fig. 5. Class `OrderLineItem` after compilation

Notice that the body of the operation `getAmount` no longer appears in the class definition. However, it has not really disappeared. As part of the compilation process, a UML activity was added to the model to hold the implementation of the operation. In UML terminology, this is known as the *method* of the operation. Clicking on this element in the model shows the textual representation in **Fig. 6**, which does, indeed, have the body of the operation, as originally entered.
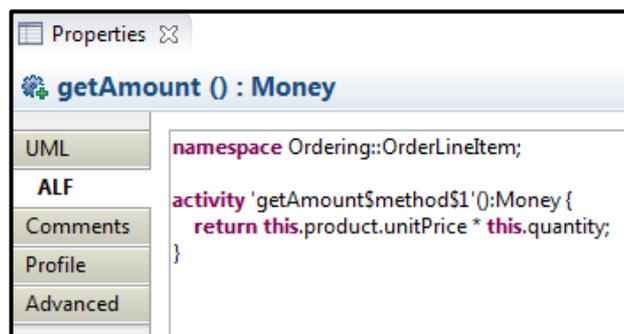

Fig. 6. Implementation of the `getAmount` operation

Note also that the compiler has automatically added default constructor and destructor operations to the `OrderLineItem` class, annotated with `Create` and `Destroy` stereotypes. In particular, you will need to use the `OrderLineItem` constructor to create a new `OrderLineItem` in the `addProduct` operation. And it would be more useful if the constructor was first updated to take line item `product` and `quantity` arguments, shown in **Fig. 7**.

```
@Create
public OrderLineItem(in product: Product, in quantity: Integer){
    this.product = product;
    this.quantity = quantity;
}
```

Fig. 7. `OrderLineItem` constructor implementation

Now you are ready to add the new attribute and operation to `Order`. So, click on the `Order` class on the diagram, getting the textual representation shown in **Fig. 8**.
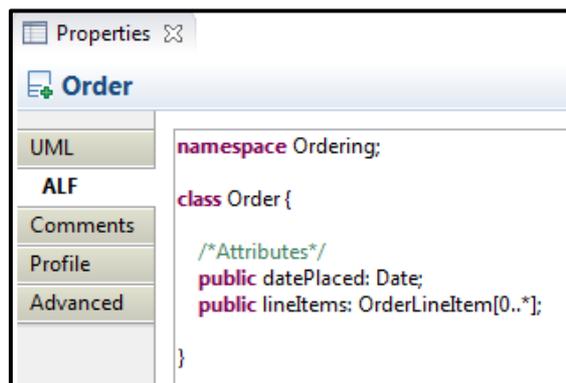
```
Properties ⊠

Order

UML          namespace Ordering;
ALF          class Order {
Comments
Profile         /*Attributes*/
Advanced        public datePlaced: Date;
                public lineItems: OrderLineItem[0..*];

             }
```
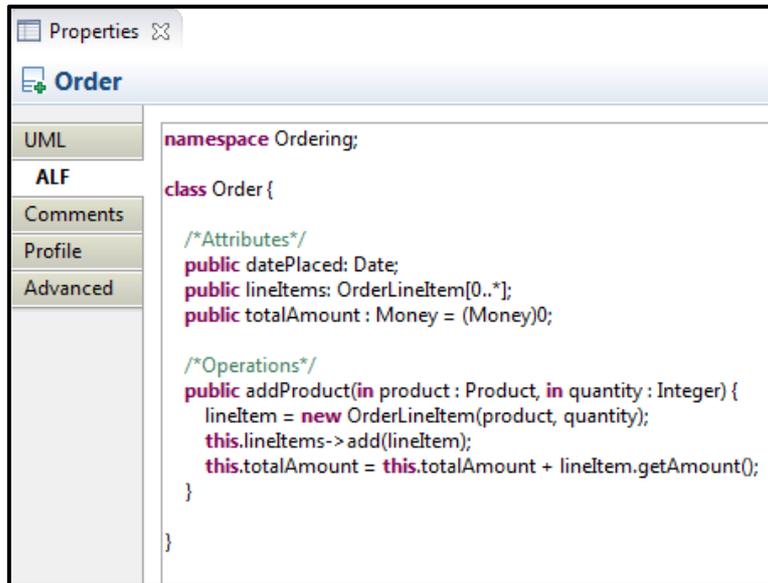
Fig. 8. The textual specification corresponding to the `Order` class

Then add the `totalAmount` attribute of type `Money` and the operation `addProduct`, with its implementation, as shown in **Fig. 9**. Finally, compile the result, in order to propagate your changes within the model.

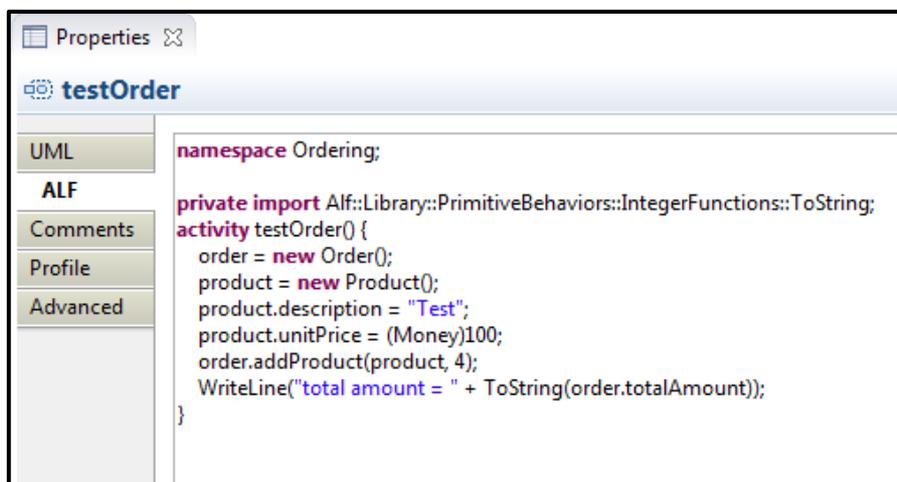**Fig. 9.** `Order` class updated with a new attribute and a new operation

We can now end this example by creating a little test program for our simple model, entering it as the UML activity shown in **Fig. 10**.

Fig. 10. A test of the example model

Since Alf compiles to fUML, the compiled activity can be executed using Moka, the Papyrus model execution framework.[10] When interpreted, the model provides the expected result:

```
total amount = 400
```

This section presented a simple executable model combining UML notation and Alf. What we have seen is that textual representation introduces an additional view of the underlying model, by which the user can actually modify the model. Maintaining cohesion between multiple views (such as graphical and textual) and a model is a challenging task. The next section briefly discusses the existing support for multiple synchronized views, and their usefulness.

6   Keeping text and model synchronized

In the context of Papyrus two behaviors are supported in order to synchronize what is in the model with what appears in the graphical representation (i.e. the diagrams).

1. A model element can be partially synchronized with its graphical representation. For example if a class "A" containing two properties has a representation in a diagram, perhaps only one of these properties may be shown. If something changes about this property, its representation is of course updated. However if something else is added to the class, its representation remains the same.
2. Full synchronization can be maintained between the model and the graphical representation. This means that for a model element that appears on a diagram, any information about its owned elements is shown. **Fig. 11** illustrates the full synchronization between the `Order`[11] class and its view in the diagram. Properties operations, methods and other elements all appear in the graphical representation.

---

[10] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[11] Note the naming convention used for methods of operations. These are introduced by the compilation process.

```
                        Order
+ datePlaced: Date [1]
+ lineItems: OrderLineItem [*]
+ totalAmount: Money [1]
# Order$initializationFlag$1: <Undefined> [0..1]

+ addProduct( in product: Product,   in quantity: Integer)
# Order$initialization$1()
«Create»  + Order(): Order
«Destroy»  + destroy()

addProduct$method$1
Order$initialization$1
Order$method$1
destroy$method$1
```
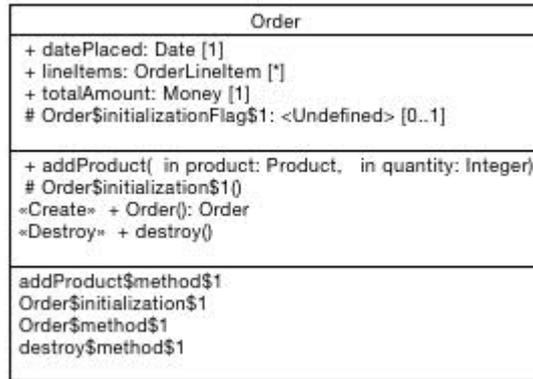
Fig. 11. Full `Order` class graphical synchronization

The integration of the Alf tooling allows a user to edit the same model through two different views in Papyrus. The main problem here is to ensure that when a modification of the model is performed through a specific view, then other views are synchronized according to these changes.

In the context of the Alf tooling, we take advantage of EMF notifications to determine which textual representation have to be re-computed when the model changes. Say we have a class "B" that inherits from the class "A" and both are located in Package "SimplePackage". If class "A" is renamed to "C", then this implies the textual representation of B has to be computed as well has the one of "SimplePackage" that contains both classes.

There are, of course, cases in which the synchronization cannot be maintained between the views, and this is mainly related to the feature proposed by the tooling. In the context of Alf tooling, the user is allowed to modify the textual specification of a model element but is not forced to compile (i.e. to propagate the change to the model) right away. Indeed, the user has the ability to keep an incomplete or invalid Alf specification, which can be completed later on. However, this also introduce the possibility that the user may try to change the model from a different view, while ongoing changes remain pending for the textual view. This possibility is prevented by the tooling, which asks the user what to do. Typically, the user has the choice to keep or override his ongoing changes. Tooling relying on the Eclipse compare framework to enable the user to resolve manually a conflict, as it is proposed for example in Git, is expected in future developments.

The next section identifies current limitations of the Alf tooling.

7 Tooling limitations

The current version of the Alf tooling supports the basic features required to specify and modify an existing model using Alf. However, there is still a long way to go to bring this tooling to a level comparable to other existing professional programming

environments. This section identifies three limitations that need to be addressed in future development to increase the level of maturity of our current tooling.

7.1 Auto-completion, cross liking and refactoring

The Alf editor needs to provide content completion, cross-linking and refactoring capabilities, just as expected in other Eclipse-based language editors. This will considerably accelerate the time required to obtain a valid specification.

7.2 Debugging

Papyrus provides the ability to execute fUML models, thanks to its model execution platform Moka. This also benefits from an integration with the Eclipse debugger, which makes it possible to interact with an execution and analyze manipulated values. The integration with the debugger does not yet allow the placement of breakpoints into Alf specifications and the propagation of debug information to the level of Alf source. Alf source-level debugging would make it a lot easier to debug complex behavior, which, when mapped to UML activity models, really look like compiled code.

7.3 Alf specification persistence

Although Alf specifications can be compiled into equivalent UML, it is usually desirable to persist the original Alf text for later retrieval. Indeed, it might not be possible to exactly reproduce the original text from the compiled models (e.g. user text formatting). Consequently, the text is for the moment stored in the model. The technical and standardized solution is to use a comment, stereotyped "TextualRepresentation" (with a tagged value "language = 'Alf'"), attached to the element mapped from Alf. The body of the comment contains the Alf code entered by the user.

The problem with this approach is that the model itself is modified to hold the persisted Alf text. A consequence of this is that, if the user tries to compare (for example with EMF Compare) the model with another version of the same model, there will be some differences (i.e., the comments containing the Alf text), which are not significant. Preliminary user feedback seems to indicate that it might be valuable in future versions of the Alf tooling to decouple the persistence of the UML model and the persistence of Alf textual specifications.

8   Conclusion

The Alf standard is new, and it will take some time for a new generation of tooling to be completed for it. But development of this tooling is progressing, with the vision of providing all the benefits of familiar IDEs for textual programming languages, along with the benefits of synchronized graphical views provided by a UML tool.

Unlike mainstream programming languages (e.g. Java, C++), Alf has been designed to smoothly integrate with UML, both syntactically and semantically. In this

way, both the graphical and textual representations have the same precise, UML, model-level semantics.

In this paper we wanted to demonstrate that UML and Alf could in practice be used jointly to specify correct-by-construction and easily refinable system models. To do so, we presented in Section 3 a simple UML model built in Papyrus, which was then completed in Section 4 using Alf tooling we developed. The result was a model of an order system ready to be tested (i.e. executable). Beside the purely functional aspect of the tooling, note the flexibility brought by Alf in the specification of a model. Modeling choices can be updated smoothly in the model by a single click.

Although significant production applications have yet to be developed using the latest graphical and textual standards for executable UML modeling, it has already be used in significant research activities. Indeed an early prototype version of the Alf integration into Papyrus was used to specify the normative test suite for PSCS. And there is research [8, 9] into using fUML and Alf as the basis for specifying the semantics of domain-specific modeling languages.

We have also identified some limitations of the current Alf tooling compared to professional programming language environments, or based on user feedback. Removing these limitations will be an important focus in the future development of the technology.

References

1. Corcoran, D. 2011. *Model Driven Development and Executable UML at Ericsson.* http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Corcoran.pdf
2. Harel, D., and Politi, M. 1998. Modeling Reactive Systems with Statecharts: The Statement Approach. McGraw-Hill.
3. Mellor, S. J. and Balcer, M. J. 2002. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley.
4. Selic, B., Gullekson, G. and Ward, P. 1994. *Real-Time Object-Oriented Modeling.* Wiley.
5. Shlaer, S. and Mellor, S. J. 1988.Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall.
6. Shlaer, S. and Mellor, S. J. 1991. *Object Lifecycles: Modeling the World in States.* Prentice Hall.
7. Shubert, G. 2011. *Executable UML Information Day Panelist Presentation. Lockheed-Martin Space Systems Company.* http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Shubert.pdf
8. Tatibouet, J., Cuccuru, A., Gérard, S., and Terrier, F. 2014. Formalizing Execution Semantics of UML Profiles with fUML Models. MODELS.
9. Mayerhofer, T., Langer, P., Wimmer, M. 2013. xMOF: A Semantics Specification Language for Metamodeling. Satellite Events of MODELS.