Investigation And Evaluation Of UML Action Languages

Omar Badreddin, Timothy C. Lethbridge, Andrew Forward
University of Ottawa, 800 King Edward, Ottawa, Ontario, Canada
obadr024@uottawa.ca, tcl@eecs.uottawa.ca, aforward@gmail.com

Keywords**:** Action Language, Alf, OMG, OCL, UAL, UML.

Abstract: We analyze the current direction of UML Action Language development and provide a classification of the proposed action language structure and statements. We also present a functioning implementation of an action language and discuss our platform for experimenting with Action Language based systems. We propose a novel approach that adopts a bottom-up technique to defining an action language. This approach embeds the action language into a textual environment that includes the UML modeling elements. Unlike current action languages that typically address class diagrams only, our proposal includes class and state machine modeling abstractions as part of the action language. We tackle the problem of modeling-in-text not by introducing yet another programming language, but instead by providing an increased level of abstraction to existing high level programming languages. Both, a textual representation of the model and its visual view represent the same underlying model but from different vantage points.

1 INTRODUCTION

A UML action language (UAL) describes elements of a system, such as actions, algorithms, and navigation paths, which are not readily described by UML diagrams. Snippets of languages like C++ and Java can be used as a UAL, but such languages are unaware of UML abstractions, resulting in mixed levels of abstraction and 'boilerplate' code.

Current directions in creating UML action languages (UALs) adopt a top-down approach, where a new language is defined forming an additional layer of abstraction. We propose an alternative approach: iteratively discovering what is necessary in an action language starting with a pre-defined object-oriented language, like Java or Php, and adapting it by adding abstractions.

This paper covers background of UALs and investigates limitations of existing programming languages if used as action languages. We give a classification of UAL constructs, and compare Alf to our language Umple, which merges action language with textual models.

2 BACKGROUND

In 2008, the OMG published an RFP for a concrete syntax for a UML Action Language (UAL) (OMG, 2012). Responses were required to define a textual language for representing the UML subset defined in the Foundation Subset for executable UML Models (fUML).

The OMG required that the UAL be suitable for use in executable UML models. A proposed UAL had to meet a number of objectives including:

1. It must be computationally complete, meaning it must include standard arithmetic and logical capabilities supported natively or by the use of libraries.
2. The UAL must allow the invocation of user-specified external code such as legacy code.
3. It must allow embedding of native code. For example, if the target platform is Java, the UAL must allow the embedding of java statements and constructs.

The OMG received two proposals and merged them to form the Action Language for Foundational UML (Alf) (Planas et al, 2012).

2.1 Textual and Visual Modeling

Action languages allow for computational processes (such as navigation and algorithms) to be expressed at a similar level of abstraction to the declarative modeling elements found in UML such as classes, associations, and state machines.

Both actions and declarative aspects of a model can be described as diagrams or in text; however, in general, diagrams are used for the declarative aspects and text for actions.

Manipulating visual models can be time consuming and less efficient than manipulating text. Effort can be wasted on mouse-centric tasks to refine layout. This is reflected in increasing number of textual modeling environments and standards.

In 2004, OMG proposed Human-Usable Textual Notation (HUTN) (OMG, 2013), which defined a textual notation for class diagrams. However, HUTN has not seen significant adoption, and development has been discontinued. Other textual UML modeling tools have emerged (Bock, 2003), (Steel & Raymond, 2001) (Harris, 2012). For example, TextUML (Chaves, 2012) is a tool that allows the modeler to create and edit models in the same manner as one would write code.

Visual models can be appealing to the reader; they work well as a communication medium since a diagram can represent the spatial qualities of a model, whereas text linearizes the view. UML modeling tools like IBM Rational Software Modeler (IBM, 2013), or Papyrus (Papyrus, 2013), fall under this category. Visual modeling tools typically provide source code generation from models, and support for reverse engineering to mange the synchronization of modeling and coding artifacts, an approach that is not without challenges (France & Rumpe, 2007).

Because action languages for UML are textual, and due to the reasons described earlier, it is our perspective that textual UML modeling provides added value to traditional visual representations.

2.2 Emergence of Action Languages

Action languages emerged to fill the gap between abstract and visual model notations to manage structure and relationships, with more algorithmic manipulation of the model's structure (i.e. programming language-like-statements). This gap, commonly referred to as 'execution semantics', has not yet been completely formalized. UML action languages (UALs) can help both modelers and coders to achieve the following goals.

2.2.1 Define the execution semantics of models

Models are an abstraction of a system, where details are purposely left out. To execute the model, missing details need to be defined using a Turing-complete language. Executing two versions of code generated from the same model should result in the same behavior, in the same way that different traditional compilers should result in systems with the same behavior.

2.2.2 Express actions that natively interact with UML constructs

UML introduces concepts that are more abstract than what is normally found in programming languages. This includes associations, state machines, preconditions, etc. A UAL should define constructs that interact with, and fill in missing details of, such modeling constructs. For example, an action language should define statements to add or remove objects in an association, execute state machine actions, and define executable checks for pre- and post-conditions where appropriate.

2.2.3 Express algorithmic details in a usable and maintainable way

To support an executable modeling environment, the need to unambiguously define algorithmic computations is imperative. A UAL should enable the modeler to define such algorithmic computations at a level of abstraction that is as high as possible and which builds on and complements modeling elements in a simple and elegant way.

2.2.4 Avoiding, or delaying, commitment to an execution platform

A UAL should allow modelers, and developers, to produce an executable system and, at the same time, to delay commitment to an execution platform. For example, a modeler should be able to define state machine actions in the UAL, and later in the development life cycle, a developer can choose to generate or embed Java code (or both), after committing to a Java execution platform. This is desirable in a model driven environments, where the same model may be eventually implemented on more than one execution platform.

2.2.5 Early verification and enhancement of reuse

Because a UAL would be defined over an executable subset of UML, it must be possible to execute the UML models, along with the associated action language, early in modeling

activities. Modelers can then see an executable prototype of their system, and refine their model accordingly.

2.3 Why not use a programming or constraint language?

Reasons for not using an existing programming language can be summarized in the following four points. These mirror the points expressed by Mellor et al (Mellor et al, 1999):

2.3.1 Programming languages provide more than what an action language needs

Java console I/O statements, and UI frameworks for Java are examples where the programming language is too powerful for what is needed from an action language. A programming language provides a large number of statements and libraries to accomplish the tasks like displaying output. They also provide freedom regarding how instance variables and methods can be used to represent and manipulate attributes and associations. Such concepts therefore have many concrete mappings, and when presented with implementation code, the developer has a hard time seeing the abstractions. A UAL can abstract the most commonly used concepts and make the algorithmic elements in models easier to understand.

2.3.2 Commitment to implementation

When programming an abstraction such as an association in a language like Java, one is forced to choose the low-level details, such as the names of methods and the algorithms. It is hard to change these later. As another example, when implementing a state machine one may choose to use a string attribute, but one may later on decide to change to an *enum* and hence have to change the code considerably. On the other hand, if using a UAL, this decision would be made by the compiler or code generator, and could be changed simply by changing the some configuration option, if a need arises.

2.3.3 Programming languages do not support concepts such as associations or states

As mentioned, a language like Java does not have constructs for the representation of associations or state machines, and consequently does not promote abstract thinking on the part of programmers.

2.3.4 Declarative constraint languages lack support for algorithms

OCL-like languages do a good job in navigating associations and defining pre and post conditions, but do not support implementation of algorithms.

3 MOTIVATING EXAMPLE

Our example is comprised of the class and state machine models illustrated in Figures 1 and 2. The class diagram describes a simple shopping system. Class Order has a deliveryAddress attribute, and an optional one-to-one association with ShoppingCart. Figure 2 shows the state machine diagram for instances of the class Order.

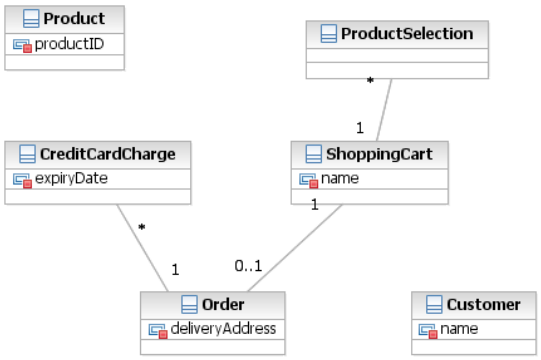

Figure 1: UML Class Diagram.

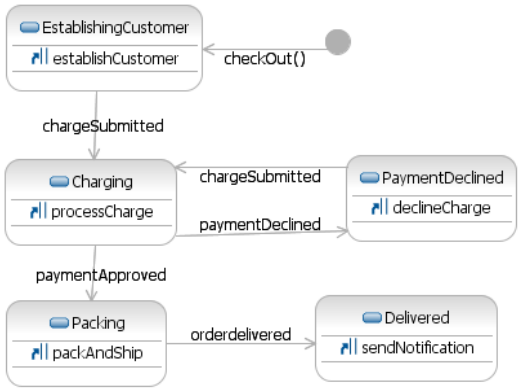

Figure 2: State machine diagram for the order class.

The state machine defines the behavior of the instances of the class Order. When event

CheckOut() occurs, the order object becomes instantaneously in *EstablishingCustomer* state. Upon entering that state, the entry action is executed. Additional actions and transitions are illustrated in Figure 2.

4 CLASSIFICATIONS OF ALF STATEMENTS

To analyse UML action language, we focus on the scope, statements, language structure, and assess the abstraction level. In what follows, and without loss of generality, we routinely refer to Alf as a representative UML Action Language.

Alf statements add a level of abstraction to widely adopted high level programming languages. Alf aims to maintain similar look and feel to high level programming languages like Java to enhance adoptability. For example, comments and blocks are formed in the same manner.

We present two classifications of Alf statements; one is based on modeling elements, and the other on the abstraction level compared to a common object oriented programming language like Java.

4.1 Modeling element classification

In this classification, Alf statements are classified into four categories: 1) statements for manipulation of local variables; 2) statements for manipulation of attributes; 3) statements for manipulation and navigation of associations, and 4) statements for specifying behavior

The following subsections give an example for each category and are not meant to be exhaustive.

4.1.1 Manipulation of local variables

Alf allows the modeler to create variables for intermediate computation. Variables can be created, assigned, and reassigned in a fashion similar to programming languages. Expressions in this category include assignment and comparison expressions.

In that sense, there is a little difference between Alf and any other programming language, except that Alf differentiates between modeling attributes and local variables.

4.1.2 Manipulation of attributes

To manipulate attributes in a class diagram, Alf must provide means for navigation of the class diagram, as well as namespaces. The statements and expressions in this category bear some similarity with OCL expressions. For example, the following statement navigates from *CreditCardCharge*, to *Order*, to *ShoppingCart*:

```
CreditCardCharge::Order::ShoppingCart
```

Because many existing languages use the dot notation, Alf considers the following to be semantically equivalent:

```
CreditCardCharge.Order.ShoppingCart
```

In addition, Alf supports common high level programming statements, like:

- Boolean operations (And, Or, etc)
- Increment and decrement of attributes
- Additions and subtractions

4.1.3 Manipulation and navigation of associations

Expressions for manipulating associations are performed similarly to navigation expressions in OCL. One difference is that collections in Alf are flat; a collection cannot itself contain collections.

UAL defines link statements to add and remove objects to and from association ends. For example, to add Credit Card to the Order-*CreditCardCharge* association, the UAL syntax is similar to:

```
order ->add(new_creditCard)
```

4.1.4 Specifying Behaviour

This category encompasses a wide range of the statements. All statements for manipulation of local variables and attributes also fall under this category, since the definition of behavior inevitably includes manipulation of variables and attributes.

Similar to high level programming languages, a curly bracket can be used to group a number of statements into a block. Alf statements and blocks can be associated with annotations that define execution semantics using the keyword @. For example, such an annotation can define whether the statement, or block, is executing in the same or separate thread.

Most importantly, Alf supports the so called in-line statement, where a native code of a programming language can be embedded. Alf passes the in-line statements to the underlying platform for execution., Platform independence is compromised as soon as an inline code is inserted.

Lastly, under this category is Alf control structures (if, switch, while, do, for, break and return). The syntax and semantics for such

statement is very similar to traditional programming languages.

4.2 Abstraction level classification

Action languages are, by definition, at a higher abstraction level compared to common object oriented programming languages. However, as we have illustrated, some Alf statements are at the same abstraction level as a classic programming language. We therefore use two categories for this classification; *common abstraction level*, an abstraction level common to modern programming languages, and a *high abstraction level*, an abstraction level higher than this. For example, when a UAL statement can be implemented in one object-oriented programming language statement, the UAL statement is categorized under common abstraction level. However, when the UAL statement requires more than one programming language statement, the UAL statement is categorized under high abstraction level.

4.2.1 Common abstraction level

Alf statements concerned with manipulation of local variables and attributes, as well as Alf's control structure statements, largely fall under this category. A language such as Java makes available syntax for manipulating attributes as well as logical and comparison operations similar to that of Alf. At times, the syntax of Alf is identical to that of Java, in an attempt to enhance the action language familiarity and adoption.

4.2.2 High abstraction level

Three main aspects of Alf's syntax fall under this category, namely:

1. Manipulation of Associations
2. Annotation
3. Mixin of inline native code

These three aspects are not typically available in an object oriented programming language, without the use of libraries. In that sense, Alf's syntax is of a higher level of abstraction.

4.3 Challenges in the current Alf approach

The current top-to-bottom approach to defining an action language gives rise to the following issues.

Firstly, a considerable number of constructs in the action language are indeed identical to programming languages. This might be desirable but does raise the question about to what extent an Action Language is different than a programming language? Does the difference justify the overhead of creating a yet another programming language?

Secondly, there is no evidence that the scope and depth of the current statements are sufficient to satisfactorily produce executable systems. Any action language needs to support wide variety of domains and be able to sufficiently support the development of wide variety of applications. An action language should support constructs that are most valuable in a modeling environment, and do so in a way that has been shown to be usable by programmers.

Thirdly, do the newly-defined constructs blend well in a modeling environment? An action language should eventually generate executable artifacts. It is still unclear to what extent the existing UAL statements will help in generating error-free artifacts. In a typical modeling environment, models generate different patterns of code; action languages should be able to deal with this complexity by generating code that overall behaves as expected.

Finally, the design of the action languages, we observe, is based on best guess effort to define what constitutes an action language, and what not.

We are not aware of any empirical evidence that Alf statements actually reflect existing patterns in software development sources across platforms and domains. Moreover, because an action language will execute in a modeling environment, the evidence needs to be based on patterns prevalent in an executable modeling environment, which to date, is not widely adopted.

In our efforts to avoid some of the challenges in the top-bottom approach of defining action languages, we have built a platform that supports incremental definition of an action language in a modeling environment. Our bottom-up approach compliments the effort to formalize and standardize action languages and avoids some of those challenges.

5 THE UMPLE ACTION LANGUAGE PLATFORM

The Umple approach to implementing a UML action language is distinct from the official OMG approach in three aspects. First, Umple makes a textual representation available for UML modeling elements and integrates the textual action language

with the textual modeling constructs. This is done without loss of the visual representation of UML models. Modelers can create and edit models diagrammatically or textually, and can embed the action language textually. This allows modelers and the developers to reason uniformly about models and action language statements. Second, Umple's bottom-up approach attempts to raise the abstraction level of the widely adopted programming languages to include modeling abstractions and action semantics, effectively overcoming limitations associated with programming languages use as action languages in UML models. Such an approach enabled the team to continuously use the UML and the Action Language in building real systems of considerable complexity.

We raise the abstraction level of programming languages by iteratively executing the following language refinements (LRs).

L.R-1. Make available additional, and more abstract, language constructs.

L.R-2. Restrict and modify statements so they become language independent

L.R-3. Within our modeling and action language environment, and by building complex systems, we iteratively identify new language constructs for inclusion in our Action Language.

Umple is a complete development platform. The discussion in this paper is limited to its relevance to UML action languages. Other publications on Umple should be referred to for more information (Badreddin et al, 2014), (Badreddin et al, 2014), (Badreddin, 2013), (Badreddin & Lehtbridge, 2013), (Badreddin et al, 2012), (Badreddin & Lethbridge, 2012).

5.1 Overcoming limitations with existing programming languages for use as an action language

Umple, as we show in the remainder of this paper, addresses the limitations in programming languages for use as an action language as follows.

5.1.1 Programming languages provide more than what an action language needs

This limitation is overcome in Umple by limiting the scope of the programming language into the subset required in the action language. The Umple compiler handles this by marking statements that are outside of a limited set with a warning in the editor view. Those warnings do not prevent Umple from compiling and executing the model and the action language, because the underlying Umple platform supports all programming language statements. We find this flexibility highly useful in building full systems using Umple. In addition, the scope of an Action Language that is powerful enough to build complete systems is bigger than we first anticipated.

5.1.2 Commitment to implementation

Umple no longer requires the programmer to implement many abstract concepts; as in ordinary compilers, the many implementation decisions are left to the compiler designers. The compiler will select a suitable implementation based on the target environment.

Take for example a 'for loop' in a typical high-level language compiler. The for loop is implemented in a machine language in a number of different ways, all are deemed acceptable as long as the semantics of the for loop is maintained. Taking the same concept to the modeling abstraction, consider a state machine. There are a variety of approaches to the implementation of state machine behavior (Gurp & Bosch, 1999), from an action language perspective, all are acceptable as long as the semantics of the state machine is maintained.

5.1.3 Programming languages concepts such as association or states

This is one core aspect of Umple. Umple makes available those UML constructs in the language itself. This becomes evident when we present the language syntax.

5.1.4 Declarative constraint languages lack support for algorithms

Because Umple is based on object-oriented programming languages, this limitation is not applicable to Umple. In addition, Umple supports aspects of the OCL, an example being the *pre* and *post* conditions we present in the following sections.

5.2 Umple Syntax

We illustrate Umple syntax by implementing our motivating example (explained in Figures 1 and 2) using Umple syntax. Umple models a Class and its associated state machines in the same or separate

artifacts. In this paper, we model the class and state machine models in the same artifact. The class diagram and state machine diagram in Figure 1 and 2 can be represented in Umple, in part, as follows:

```
Class Order {
  deliveryAddress;
  1 -- * CreditCardCharge;
  0..1 – 1 ShoppingCart;

  orderStatemachine {
    EstablishingCustomer {
      entry / {establishCustomer();}
      chargeSubmitted -> Charging; }

    Charging {
      entry / {processCharge();}
      paymentApproved -> Packing;
      paymentDeclined ->
        PaymentDeclined;  } } }

class CreditCardCharge {
  expiryDate; }
```

This illustrates the textual representation of UML models in Umple. In addition to the UML elements present in the motivating example, Umple has similar syntax for nested states, entry, exit and transition actions, guards, events, and do activities. The complete Umple grammar and syntax is maintained on the Umple home page (Lethbridge et al, 2012).

5.3 Umple modeling abstractions

In this section, we define the execution semantics of Umple's modeling and algorithmic elements. Umple defines an executable subset of UML for which Umple generates executable artifacts that implement this semantics. Examples of the modeling elements are the following.

5.3.1 Associations

Umple supports all possible multiplicity combinations, and generates code that maintains multiplicity and referential integrity at run time. For our motivating example, Umple makes available the following statements for the one-to-many association with *CreditCardCharge*:

```
getCreditCardCharge(int index)
```

This interface returns the *creditCardCharge* matching the index.

```
getCreditCardCharges()
```

This returns a list of all *creditCardCharges*.

```
numberOfCreditCardCharges()
```

This returns the number of *creditCardCharges* associated with the order object.

```
hasCreditCardCharges()
```

This returns true if the object order has at least one *creditCardCharge* associated with it.

```
indexOfCreditCardCharge(aCreditCardCharge)
```

This returns the index of the *creaditCardCharge*. Umple also generates interfaces to manipulate associations by adding and removing objects to either side of the association. Those interfaces maintain the integrity of association multiplicities at run time. This is an example of additional, and more abstract language constructs (L.R-1).

5.3.2 Attributes

Umple generates setter and getter interfaces for all attributes, and allows the user to insert his own pre and post conditions for the setters and getters. Various properties such as immutability can also be specified.

5.3.3 State machine

UML state machines define the behavior of instances of a class. Umple generates artifacts to implement state machine semantics. Events become part of the system interface and the state transitions are executed in response to events.

State machine guards are an example of limiting the scope of a programming language (L.R-2). Early releases of Umple allowed arbitrary statements as guards. We later restricted guard code to be only simple Boolean expressions, or a function call that returns a Boolean value.

Umple events illustrate where a construct is modified to make it language independent (L.R-1). Umple events are represented by a name, rather than a function call statement, making the event name language-independent; the syntax is unchanged regardless of the target language.

5.3.4 Umple algorithmic elements

To fully support UML executable environment, Umple enables modelers to include algorithmic

elements in the model. Algorithmic elements can make use of Umple's generated interface. Modelers can embed their natively-defined algorithmic elements in the language of their choosing. Let's take the example of navigating from *CreditCardCharge* to *Order*, to *ShoppingCard*. Because Umple generates automatically a number of interfaces, the navigation can be performed as follows:

```
getCreditCardCharge(index).getOrder(
).getShoppingCart()
```

Algorithmic code can be embedded within state machine entry, exit, and transition actions. Blocks of code from inline algorithmic statements can be referenced by name within any state machine element.

We are iteratively adding additional restrictions to Umple-based object-oriented languages (L.R-2 and L.R-3). For example, we restrict manipulation of the model attributes to only the setters and getters. We also disallow statements that manipulate internal representation for state machines and associations.

5.4 Umple in practice

Because Umple is a fully executable action language environment, we are able to use it in building a variety of applications, both model-intensive and/or algorithmic-intensive. We came to the realization that the subset of a programming language to satisfy the action language requirement is larger than we first anticipated (L.R-3). We currently limit statements that violate modeling integrity, for example, statements that result in updating state machine internal representations. Because there are a wide variety of systems where Umple is used, limiting the scope of the action language results in unintended hindrance to modelers and developers alike.

We have built Umple using Umple itself, a practice commonly referred to as 'eating your own dog food'. This guarantees robustness.

6 COMPARISON OF ALF AND UMPLE

Before making the comparison, it is imperative to note the following core differences:

1. Alf is an action language added to UML, while Umple is an action language in a fully executable platform for experimenting and developing action languages.
2. While both Alf and Umple target an unambiguous execution of UML models, Umple takes the approach of raising the abstraction levels of object programming languages, while Alf defines a new language that will then be executed on some platform.

6.1 Representation of the UML modeling and execution artifacts

Alf is to be embedded in the visual elements of UML models. The supporting tool should enable the modeler to manipulate both textual and visual elements in the same view. UML models, in particular large models, may become overloaded by the number of textual elements. In addition, some macro textual editing features may inevitably be compromised by embedded the textual artifacts across a number of visual elements.

Umple assumes the visual model and the textual representation are two faces of the same coin. Umple attempts to blur the lines between the model and the action language, where the visual model becomes merely an editable view.

6.2 Approach for raising the abstraction level

Alf implies a language-independent language. In other words, a file containing UAL statements can generate virtually any implementation language code, whether that be Java, or Php. This conforms to the common need in model driven engineering projects, where the implementation language and platform need to be determined in a lazy fashion.

Umple's bottom-up approach takes the stand of starting from a full-fledged object-oriented programming language. This approach enabled us to:

1. Build real and fully functional systems using Umple, and learn from how an action language is used in a modeling environment.
2. Iteratively add refinements to enhance the programming language.
3. Quickly assess the impact of limiting scope, or adding new abstractions, to systems and users.
4. Significantly reduce barriers to adoption. Using a familiar syntax means Umple users require minimal training to be able to start using and building systems using Umple.

This does mean that for each base language we have to create a parser that extends the base

language with Umple concepts. We have done this for Java PHP and C++ and other languages that allow the {} notation for blocks. The Umple constructs would not need to change.

6.3 Platform Independence

In a Model Driven Arechitecture, a Platform Independent Model (PIM) is a model that has no platform dependencies, while a Platform Specific Model (PSM) is a model optimized for execution on a specific platform (France & Rumpe, 2007). Alf is a PIM, since there is should be no dependency on the language side for execution on any specific platform. However, as soon as native code is embedded in-line, the platform independence is compromised, since the model becomes tide to the embedded language platform.

Pure models in Umple are platform independent, since executable semantics can be generated for any platform. Umple action language is as platform-independent as the underlying language execution platform is.

6.4 Lines of Code Comparison

While lines of code (LoC) is a simple measure, it is considered to be a good indicator for complexity (Gold et al, 2005). Alf, at the time of writing, does not support state machine constructs. We therefore make the comparison based on class diagram, associations, and attributes. Because attributes are defined in similar fashion in Alf and Umple, we focus on classes and association. For the sake of demonstration, let's consider the association between the classes *ShoppingCart* and Product. This association is represented in Alf as follows:

```
public active class ShoppingCart;
public active class Product;
public assoc R4 {
  public : ShoppingCart[0..*];
  public :  Product[1..*]; }
```

In Umple, this association is defined as follows

```
class ShoppingCart {
  0..* -- 1..* Product; }
```

In the case of Umple, there is no need to explicitly define a class Product because Umple identifies Product as a class as it is participating in an association. In addition, the association can be defined in one end class, or both, or in a separate entity.

Our motivating example has four associations, which can be implemented in Alf in (4*5 = 20 LOC), while in Umple, the same associations are implemented in (4*2 = 8 LOC).

7 RELATED WORK

There is a consensus in the research and professional communities that UML models are, by themselves, incomplete with regard to executability. UML models can have a number of varying interpretations (Evans, 1998), (France et al, 1997), (Evans, 1998). Action languages, or methods for formalizing execution semantics, are referred to as a way to provide such formalism.

For different types of models, researchers and practitioners have identified the need for explicit and unambiguous execution formalism. At the meta-model level, Muller et al (Muller et al, 2005) proposed a language for precise action specification at the meta level. Sunyé (Sunyé, 2001) illustrates how an action language can be applied at the meta-model level to maintain behavior-preserving transformation, implement design patterns, and achieve design aspect weaving (Keller & Schauer, 1998). Action language usages extend to formally defining model transformations. Varro and Pataricza (Varro & Pataricza, 2003) propose an executable action language for formally defining model transformations. Their language generates model transformation scripts for a number of existing off-the-shelf software tools.

Alvarez et al (Alvarez et al, 2001) proposes an action semantics language for UML where actions are defined as computational procedures with side-effects.

A Java-like action language called JAL is proposed by (Dinh-Trong et al, 2005). JAL is a simple language that they used for defining the actions in the activity diagrams with the goal of automated test generation for class and activity diagrams.

8 CONCLUSIONS

We have defined a technology called Umple that has some advantages over Alf as a UML action language. Alf is a new textual language designed to be embedded in UML constructs, whereas Umple allows any language to be used as an action language.

We also have developed a process whereby we have incrementally developed Umple bottom-up to provide action-language capabilities. It has been tested in practice on various systems, including Umple itself. This contrasts with Alf, which has been developed top-down. Umple can be incrementally be adopted by developers who are used to using standard languages and want to move towards modeling; Alf, on the other hand requires a complete rewrite of action code.

REFERENCES

Alvarez, J. M., Clark, T., Evans, A. and Sammut, P. "An Action Semantics for MML". 2001. Lecture notes in computer science, Springer. pp. 2-18.

Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Exploring a Model-Oriented and Executable Syntax for UML Attributes." Software Engineering Research, Management and Applications. Springer, 2014. 33-53.

Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity." Software Engineering Research, Management and Applications. Springer, 2014. 129-149.

Badreddin, Omar. "Empirical evaluation of research prototypes at variable stages of maturity", User Evaluations for Software Engineering Researchers (USER), 2013 2nd International Workshop , 10.1109/USER.2013.6603076. 2013 , Pages: 1- 4.

Badreddin, Omar, Lethbridge, Timothy C., "Model Oriented Programming: Bridging the Code-Model Divide". ICSE Workshop on Modeling in Software Engineering, 2013, Modeling in Software Engineering (MiSE), 2013 5th International Workshop , 10.1109/MiSE.2013.6595299. 2013 , Pages: 69 - 75.

Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Model oriented programming: an empirical study of comprehension." 2012 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., 2012.

Badreddin, Omar. ; Lethbridge, Timothy C. "Combining experiments and grounded theory to evaluate a research prototype: Lessons from the umple model-oriented programming technology", User Evaluation for Software Engineering Researchers (USER), 2012. 10.1109/USER.2012.6226575 , 2012 , Page(s): 1- 4.

Badreddin, Omar, Timothy C. Lethbridge, and Maged Elassar. "Modeling Practices in Open Source Software." Open Source Software: Quality Verification. Springer, 2013. 127-139.

Bock, C. "UML without Pictures". 2003. IEEE Software, vol 20, pp. 33-35.

Chaves, R. " TextUML", accessed 2012, http://abstratt.com/.

Dinh-Trong, T., Kawane, N., Ghosh, S., France, R. and Andrews, A. A. "A Tool-Supported Approach to Testing UML Design Models," in Proccedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2005.

Elkoutbi, M., Khriss, I. and Keller, R. K. "Automated Prototyping of User Interfaces Based on Uml Scenarios". 2006. Autom.Software.Eng, vol 13, Springer. pp. 5-40.

Evans, A. "Making UML Precise, OOPSLA'98". 1998. October.

Evans, A., France, R., Lano, K. and Rumpe, B. "Developing the UML as a Formal Modelling Notation," in UML, 1998, pp. 397–407.

France, R. and Rumpe, B. "Model-Driven Development of Complex Software: A Research Roadmap," in FOSE '07: 2007 Future of Software Engineering, 2007. pp. 37-54.

France, R., Evans, A., Lano, K. and Rumpe, B. "The UML as a Formal Modeling Notation". 1997. Computer Standards and Interfaces, vol 19, Citeseer. pp. 325-334.

Gold, N., Mohan, A. and Layzell, P. "Spatial Complexity Metrics: An Investigation of Utility". 2005. IEEE Trans.Software Eng., vol 31, pp. 203-212.

Harris, T. " YUML", accessed 2012, http://yuml.me/.

IBM. " IBM Rational Software Architect Modeling Tool", accessed 2013, http://www-01.ibm.com/software/awdtools/architect/swarchitect/

Keller, R. K. and Schauer, R. "Design Components: Toward Software Composition at the Design Level," in Proceedings of the 20th Int'l Conference on Software Engineering, 1998. pp. 302-311.

Lethbridge T.C., Forward, A. and Badreddin, O. " Umple Language Online.", accessed 2012, http://try.umple.org.

Mellor, S. J., Tockey, S. R., Arthaud, R. and Leblanc, P. "An Action Language for UML: Proposal for a Precise Execution Semantics". 1999. Lecture notes in computer science, Springer. pp. 307-318.

Muller, P. A., Fleurey, F. and Jézéquel, J. M. "Weaving Executability into Object-Oriented Meta-Languages". 2005. Lecture notes in computer science, vol 3713, Springer. pp. 264.

Object Management Group (OMG). " Concrete Syntax for a UML Action Language RFP", accessed 2012, http://www.omg.org/cgi-bin/doc?ad/2008-9-9.

Object Management Group (OMG). " Human-Usable Textual Notation", accessed 2013, http://www.omg.org/technology/documents/formal/hutn.htm.

papyrus, "The Papyrus UML", accessed 2013, http://www.papyrusuml.org.

Planas, Elena, et al. "Alf-Verifier: an eclipse plugin for verifying Alf/UML executable models." Advances in Conceptual Modeling, 2012. Springer Berlin Heidelberg, 2012.378-382.

Steel, J. and Raymond, K. "Generating Human-Usable Textual Notations for Information Models," in Fifth International Conference on Enterprise Distributed Object Computing (EDOC 2001), Seattle, Washington, USA, 2001. pp. 250-250.

Sunyé, G., Pennaneac h, F., Ho, W. M., Le Guennec, A. and Jézéquel, J. M. "Using UML Action Semantics for Executable Modeling and Beyond". 2001.Lec' notes in comp' sci', Springer. pp. 433-447.

Van Gurp, J. and Bosch, J. "On the Implementation of Finite State Machines," in Proceedings of the 3rd Annual IASTED Int'l Conference Software Engineering and Applications, 1999. pp. 172-178.

Varro, D. and Pataricza, A. "UML Action Semantics for Model Transformation Systems". 2003. Period Polytech Electr Eng, vol 47, Citeseer. pp. 167-186.