Modeling Meets Programming: A Comparative Study in Model Driven Engineering Action Languages

Maged Elaasar[1] and Omar Badreddin[2]

[1]Modelware Solutions,
La Canada Flintridge, CA 91011, USA
melaasar@gmail.com

[2]Electrical Engineering and Computer Science Department
University of Texas at El Paso
obbadreddin@utep.edu

Abstract. Modeling and programming have often been considered two different activities. While this may be true when modeling is primarily meant for human communication and early design explorations, it is not the case when modeling is meant for execution. Some approaches have been specifically developed to address this latter case with variable successes. In this paper, we discuss two such approaches, namely ALF and Umple. ALF has evolved from the modeling community to provide a textual syntax for an executable subset of UML called Foundation UML (fUML). Umple has evolved from the academic community to introduce the abstractions of modeling into programing languages. We compare both approaches, highlight their critical differences, and discuss their contribution to the evolution of model oriented programming languages.

Keywords: Modeling, Programming, UML, ALF, Umple, Model Driven Engineering, Action Language.

1 Introduction

A model is a simplified representation of a more complex system. It is frequently used to abstract and analyze a system by focusing on one or more aspects. Models are used to understand, communicate, simulate, calibrate, evaluate, test, validate and explore alternatives for system development. Modelers use a wide variety of models to explore different aspects of the system such as requirements, structure, behavior, event, time, security, flow, process, activity, performance, quality, usability, etc. These models can be expressed in many forms including textual and visual representations.

Model Driven Engineering (MDE) [1] is a well-accepted engineering approach, where models are used to understand and comprehend parts of a complex system under development. Modeling languages raise the level of abstraction for the specification of a system to help manage system complexity and evolution. Modeling languages are typically defined in terms of their abstract syntax (metamodel), concrete syntax (or notation) and semantics (or rules). While some modeling languages are executable, like the foundational subset of UML (fUML) [2], others are not. This, according to some, is one criterion that distinguishes modeling languages from programming languages.

Programming, on the other hand, is the activity of developing executable software. Programs are written in a programming language, which is a set of rules for expressing computations in a human-readable form that can be translated unambiguously to a machine-readable form. A programming language is defined in terms of its, typically textual syntax, defined with some grammar formalism like BNF [3], and its semantics defined by translation to some mathematical formalism.

There is often confusion in distinguishing between modeling and programming languages [4]. For example, there is a misconception that textual languages are always programming languages and that graphical languages are always modeling languages. This distinction becomes more challenging when domain-specific languages (DSLs) are considered [5]. DSLs can be either visual, textual, or both; and may or may not be executable. It quickly becomes evident that a single criterion is not sufficient for distinguishing modeling and programming. However, there have been some criteria suggested in the literature [24] that could potentially be used to give a more accurate classification including: a) whether the language has a textual or visual notation, b) how the language syntax and semantics are defined, c) the extent to which the language is executable, d) the language's level of abstraction, e) the underlying fundamental concepts of the language, f) how the language is used in each phase of development, and g) whether the language supports multiple viewpoints. Such criteria will result in a number of categories of languages.

One category embeds code into modeling languages. An example of this approach is the *Rational Rose RealTime* [6], whereby one can specify snippets of code in some programming language (like C) as body of UML operations, derivation of UML properties or specification of UML actions. The resulting model is translated into code in that programming language, with the code snippets being integrated in the generated code. A key drawback of this approach is that the code snippets are at a different semantic level than the model elements, making the specification and debugging difficult. Moreover, synchronizing changes between the model and the code become challenging. Finally, the platform independence of the models is compromised (a workaround is to use different UML profiles to add code in different languages).

Another category embeds modeling concepts into programming languages. A notable example here is the Umple language [7]. Umple allows embedding modeling concepts from UML (e.g., class and state machine concepts) into native code of a programming language (e.g., Java, C++). The mix is translated into code before being executed. This approach allows developers to incrementally adopt modeling practices without giving up on code. It also blurs the line between modeling and programming, since an Umple module can be all-code, all-model or anything in between. A key drawback, like the previous category, is that platform independence is lost.

A third category gives modeling languages the appearance of a programming language. An example here is the ALF language [8], which is an OMG standard. ALF is a textual syntax for the foundational subset of UML (fUML), which has well-defined execution semantics. ALF's textual syntax is parsed into instances of its abstract syntax, which maps to the abstract syntax of fUML. ALF allows the textual expression of the structure (e.g., class diagram) as well as the behavior (e.g., state machines) of fUML models. ALF programs can be executed by fUML engines, which come with UML modeling tools, such as *Papyrus* [9] and *MagicDraw* [10].

Approaches in the first category have been well explored for a long time, and have not had detrimental success in bridging the gap between the communities of modeling and programming. Therefore, we focus on approaches in the relatively newer second and third categories, most notably: Umple and ALF. The goal of both Umple and ALF is to provide an executable modeling language, with which an engineer can design and specify system behavior. Both approaches allow for visual and textual modeling and development. Umple has been developed in Academia following an evidence-based approach [18], where it has been driven by the need to improve comprehension of code [4]. ALF has been developed by the OMG, as a result of merging two proposals from IBM and BridgePoint [19], where it has been driven by the need to have a model-based action language.

Both programming and modeling have their unique strengths and weaknesses. Programming typically results in an executable artifact that can be tested, while models can sometimes be ambiguous and incomplete. Hence, there is potentially significant value in merging the two approaches and harness their strengths. In this paper, we focus on such approaches that provide formalisms that reduce or eliminate the distinction between modeling and programming. These formalisms can be either textual, visual, or both. Our contribution is to compare these two previously mentioned approaches, highlight their critical differences and discuss ideas for their evolution.

The rest of this paper is organized as follows: in section 2, we give an overview of related works; we define a running example in section 3; in section 4, we describe the ALF language approach; we also describe the Umple language approach in section 0; in section 6, we compare the two approaches and highlight their critical differences; finally, we conclude and outline future work in section 7.

2 Related Works

Adoption of MDE approaches in practice remains very limited [12]. Outside of a few niche domains (such as safety critical and embedded systems), MDE has not witnessed broad adoption. Open Source development for example has remained entirely code-centric [13]. In education, MDE pedagogies' results have been mixed, and students' perception of the role of UML in software development has been declining [29]

Challenges with adoption of MDE can be attributed to many factors, including: 1) synchronization of modeling and coding artifacts. Forward and reverse engineering technologies face many open challenges [14]. As a result, engineers find it increasingly difficult and time consuming to maintain the synchronization between model and code artifacts. 2) Challenges related to versioning and merging of large modeling artifacts. By contrast, versioning and merging of code artifacts has become widely adopted [15][16]. 3) Challenges in applying MDE practices in agile and small size projects, and 4) perceived failure of educating future engineers and convincing them of the value of MDE methodologies [17] [29]. To mitigate some of those challenges, a number of approaches have been proposed that merge both coding and modeling into the same development environment.

textUML provides an API and a library for creating UML diagrams using Java syntax [20]. Systems developed using *textUML* run over JVM, and the tool provides debugging facilities. *BridgePoint* is an executable modeling tool based on Shlaer-Mellor methodology [21]. It provides a visual editor only for UML constructs, and supports a proprietary syntax for action definitions. Both *textUML* and *BridgePoint* provide an executable platform mixing code and modeling concepts.

tUML is another approach whereby both models and code are defined in the same artifact [22]. *tUML*, like ALF, defines a subset of UML for which it provides a textual syntax. The main goal of *tUML* is to facilitate the fixing of sketchy models, by providing a familiar textual syntax.

PlantUML is yet another textual modeling and coding environment [23]. The distinctive feature of *PlantUML* is that it is very permissive and verbose. Actions in *PlantUML* can be defined using unstructured text. As a result, *PlantUML* does not strictly follow the UML metamodel.

3 Running Example

The running example is an Ecommerce ordering system that is loosely based on the Online Bookstore Domain case study given in appendix B of [26]. The example, shown in (**Fig. 1**) with a UML class diagram, describes a set of classes involved in an Ecommerce transaction. *Order* is a class responsible for handling ordering functionality. It has two attributes: *totalAmount* and *datePlaced*. It also composes instances of class *OrderLineItem*, which has the attributes *quantity*, *amount* and a reference to the product ordered. The Product class has the attributes *description* and *unitPrice*. The *Order* class is referenced by a *CreditCardCharge* class, with the attributes amount, *authorizationCode*, and a reference to a *CreditCard* class. The latter has attributes *accountNumber*, *expirationDate*, and *billingAddress*. Finally, class *Customer* has attributes name and *shippingAddress*, and references to set of *Card* and *Order* instances.

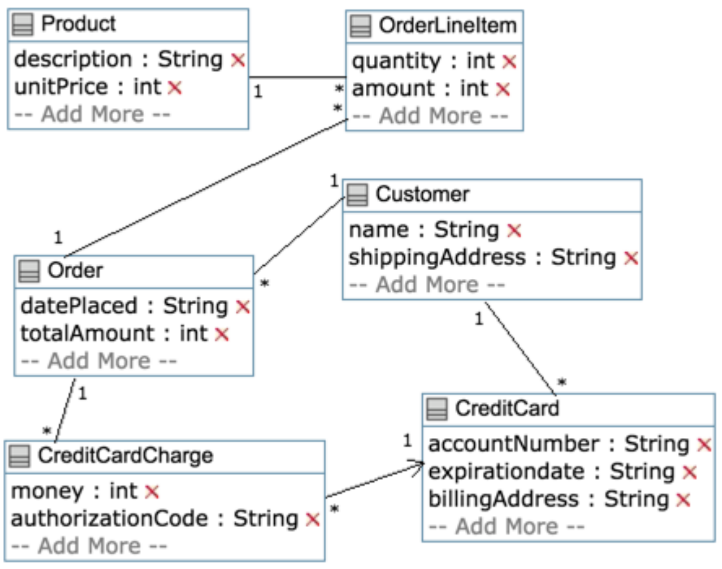

Fig. 1. Structural and Behavioral Models

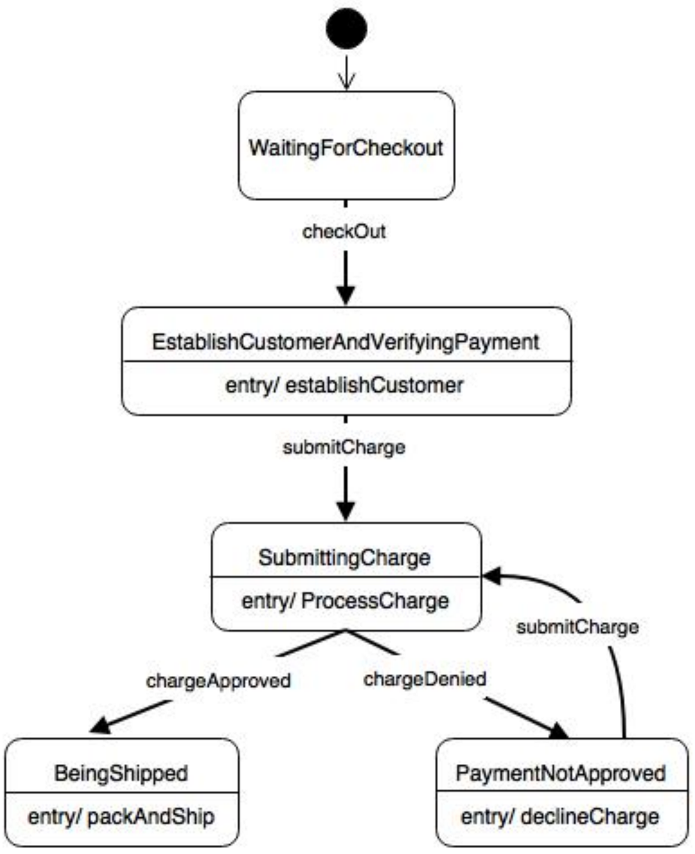

Fig 2. Behavioral Model

Out of all the above classes, *Order* is an active class (i.e., has its own thread of control), whose behavior is defined using a UML state machine (shown in Fig 2). The machine starts by waiting for a *Customer* to checkout the order (state). Once checked out (trigger), it establishes the customer (state) by invoking the (entry) behavior *EstablishCustomer*. Then, once the charge is submitted (trigger), it performs the submitting charge (state) by invoking the (entry) behavior *ProcessCharge*. At that point, if the charge is approved (trigger), it prepares for shipping (state) by invoking the (entry) behavior *PackAndShip*. On the other hand, if the charge is declined (trigger), it indicates that the payment has not been approved (state) and invokes the (entry) behavior *DeclineCharge*. This may prompt the customer to re-enter the payment information and resubmit the charge (trigger), which goes back to being processed (state).

Each one of those behaviors invoked by each state can itself be a behavior specified using a UML state machines, a UML activity, or an expression in some action language. For example, the *EstablishCustomer* behavior is specified as a UML Activity diagram (**Fig. 3**). The activity creates an instance of class *Order*, populates its collection of *OrderLineItem* instances, and sets its *datePlaced* to current date and its *totalAmount* to the sum of the amounts for each line item in the order. The reader may notice that this visual activity modeling may not be the most convenient way of specifying this activity. This is further discussed in subsequent sections.

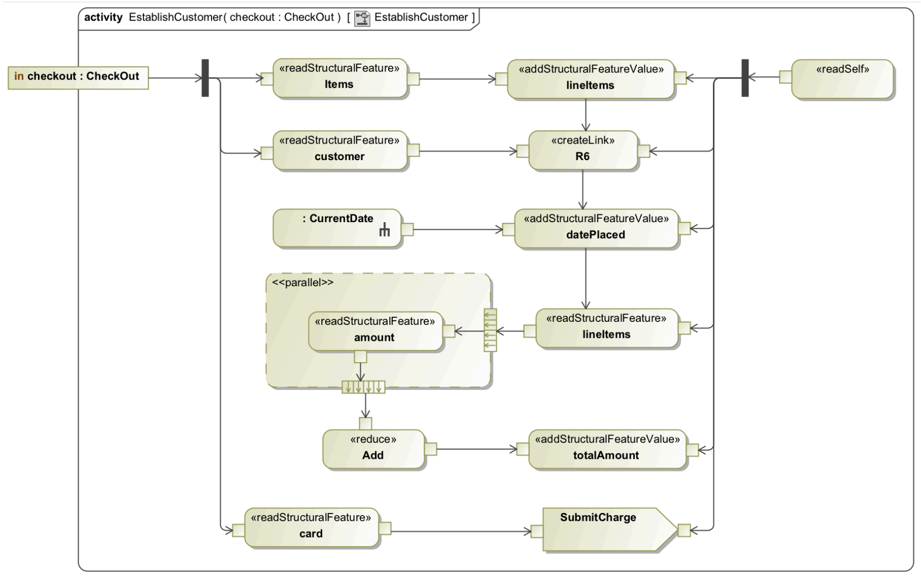
activity EstablishCustomer(checkout : CheckOut) [EstablishCustomer]
in checkout : CheckOut
«readStructuralFeature» Items
«addStructuralFeatureValue» lineItems
«readSelf»
«readStructuralFeature» customer
«createLink» R6
: CurrentDate
«addStructuralFeatureValue» datePlaced
<<parallel>>
«readStructuralFeature» amount
«readStructuralFeature» lineItems
«reduce» Add
«addStructuralFeatureValue» totalAmount
«readStructuralFeature» card
SubmitCharge

Fig. 3. Activity Modeling for *EstablishCustomer*

4 ALF: UML Action Language

ALF [2][8] is a standard action language, defined as a textual surface syntax for foundational UML (fUML) [25]. fUML is an executable subset of standard UML that can be used to define, in an operational style, the structural and behavioral semantics of systems. fUML is a computationally complete language. It has a Kernel module that provides basic object-oriented capabilities, a Common Behavior module that provides general behavior and synchronous communication capabilities and an Activities module that provides activity-modeling capabilities. Other modules being standardized include one for Composite Structure modeling [28] capabilities and one for State Machine modeling capabilities [27].

The motivation for ALF as a textual syntax for fUML is clear: the graphical notation of fUML (e.g., the activity diagram in **Fig. 3**) is not convenient for detailed programming. The textual syntax on the other hand is more suitable for the specification of detailed computations and algorithms. However, what distinguishes ALF from other action languages (e.g., regular programming languages) is that it is at the same semantic level as the rest of the model. This is achieved by mapping the ALF textual syntax directly to the abstract syntax (metamodel) of fUML. Alf also provides standard libraries for primitive types (e.g., string, boolean, integer, real, etc.) and primitive behaviors (boolean, string, arithmetic, and collection functions). It also provides basic asynchronous communication support based on the concept of channels.

ALF is defined by its concrete syntax, abstract syntax and semantics. The concrete syntax is specified with a BNF grammar. The abstract syntax is defined with a MOF metamodel that is synthesized during parsing of an Alf text and that includes additional derived attributes and constraints that specify the static semantic analysis of that text. Finally, the semantics of ALF are defined by mapping the ALF abstract syntax metamodel to that of fUML.

ALF is primarily envisioned as a textual language for specifying executable (fUML) behaviors within an overall graphical UML model. This mainly includes activity and action modeling. However, it is capable of being used to specify the structural part of a fUML model, as well as asynchronous communication. This is achieved by ALF's structural modeling features.

4.1 Structural Modeling

Structural modeling with ALF involves specifying the class diagram of the system. The code in Listing 1 provides the class diagram of the running example specified in ALF. Classes are defined separately from associations. As shown, the syntax is Java-like to facilitate adoption and comprehension.

```
active class Order {
   public datePlaced: Date;
   public totalAmount: Money; }

class OrderLineItem {
   public quantity: Integer;
   public amount: Money; }

class Product {
   public description: String;
   public unitPrice: Money; }

class CreditCardCharge {
   public amount: Money;
   public authorizationCode: String[0..1] }

class CreditCard {
   public accountNumber: String;
   public expirationDate: Date;
   public billingAddress: Address; }

class Customer {
   public name: PersonalName;
   public shippingAddress: Address[0..1] }

assoc R1 {
   public product: Product;
   public orderlineItem: OrderLineItem[*]; }

assoc R2 {
   public lineItems: compose OrderLineItem[*];
   public order: Order; }

assoc R3 {
   public order: Order;
   public charge: CreditCardCharege[*]; }

assoc R4 {
   public card: CreditCard;
   public charge: CreditCardCharege[*]; }

assoc R5 {
   public card: Card [*];
   public customer: Customer;}

assoc R6 {
   public customer: Customer;
   public order: Order[*]; }
```
Listing 1. Structural modeling in ALF

4.2 Behavioral Modeling

ALF provides textual syntax for the specification of behavior models that are supported by fUML. Up to version 1.2.1, the fUML specification supports activity models but not state machines (or interactions). However, ongoing work at the OMG on the specification of the precise semantics of state machine will change that in an upcoming revision of fUML and consequently ALF. Meanwhile, ALF can be used for specifying standalone activity models and in specific areas of state machines, most notably to specify guard conditions for transitions, and *entry*, *exit* and *do* behaviors of states, in the form of activities.

In the running example, the behavior of the Order class is specified with a state machine, so it cannot itself be specified in ALF. However, some of its entry actions, e.g., *EstablishCustomer*,(**Fig. 3**) are activities, hence can be expressed more concisely in ALF code. The code in Listing 2 creates an instance of the class *Order* and populates it based on the input parameter *checkOut*. This shoes that a textual syntax is more concise and comprehensible than the corresponding visual notation.

```
activity EstablishCustomer(checkOut: CheckOut) {
   this.lineItems = checkOut.items;
   R6.addLink(checkout.customer, this);
   this.datePlace = CurrentDate();
   this.totalAmount = this.lineItems.amount->reduce Add;
   this.SubmitCharge(checkOut.card); }
```

Listing 2. Behavioral modeling in ALF

5 Umple: UML Programming

Umple is a model oriented programming language. It integrates modeling and programming by introducing model-level concepts and embedding them textually as part of modern Object Oriented programming languages [7]. The result is a language that blurs the distinction between code and modeling abstractions. The key premise of this approach is to enable software engineers that are mostly accustomed to textual coding environments to take advantage of modeling abstractions and MDE methodologies.

To demonstrate the approach, we represent the running example presented in section 3 encoded in the Umple language.

5.1 Structural modeling

Listing 3 shows the structural part (class diagram) of the running example.

```
class Product {
  1 -- * OrderLineItem;
  description;
  int unitPrice; }

class OrderLineItem {
  * -- 1 Order;
  int quantity;
  int amount; }

class Order {
  1 -- * CreditCardCharge;
  datePlaced;
  int totalAmount;  }

class CreditCardCharge {
  * -> 1 creditCard;
  int money;
  authorizationCode; }

class Customer {
  name;
  shippingAddress; }

class CreditCard {
  * -- 1 Customer;
  accountNumber;
  expirationdate;
  billingAddress; }
```

Listing 3. Structural modeling in Umple

Like ALF, Umple also adopts a Java-like syntax to promote adoption and comprehension. Associations are defined within either of the participating classes as shown in Listing 3. Alternatively, associations can be grouped under 'association' header (as shown in Listing 4). Associations can be named, and associations role names are optional on both ends.

```
association {
  1 Product -- * OrderLineItem;
  * OrderLineItem -- 1 Order;
  1 Order -- * CreditCardCharge;
  * CreditCardCharge -> 1 creditCard;
  * CreditCard -- 1 Customer;
}
```

Listing 4. Grouping of associations in Umple

Umple supports primitive types (such as int, string, date, etc). Umple defaults attributes to string type if a type is not provided.

5.2 Behavioral Modeling

In Umple, active classes, those that own their thread of execution, can have a state machine associated with them. In our running example, the class *Order* is an Active class. Hence, its state machine can be defined as in Listing 5.

```
class Order {
  1 -- * CreditCardCharge;
  datePlaced;
  int totalAmount;

  OrderStatus {
    WaitingForCheckOut {
      checkout -> EstablishCustomerAndVerifyingPayment;
    }
    EstablishCustomerAndVerifyingPayment {
      entry/ {establishCustomer();}
      submitCharge -> SubmittingCharge;
    }
    SubmittingCharge {
      entry/{ProcessCharge();}
      chargeApproved -> BeingShipped;
      chargeDenied -> PaymentNotApproved;
    }
    BeingShipped {
      entry/{PackAndShip();}
    }
    PaymentNotApproved {
      entry/{declineCharge();}
      submitCharge -> SubmittingCharge; } }

  // methods to implement state machine actions
    private void establishCustomer () {
      this.lineItems = checkOut.items;
      order.addOrderLineItem(aOrderLineItem);
      this.datePlace = CurrentDate();
      this.totalAmount = this.lineItems.amount += Add;
      this.SubmitCharge(checkOut.card); }

    private void ProcessCharge() {
  // method action code here } }
```

Listing 5: Behavioral Modeling in Umple

5.3 Execution Semantics for Umple

Umple has been developed to be an executable language, just like ALF. However, since it is not a standard, we like to shed some lights on its execution semantics. Recall that an Umple module can mix modeling concepts with some programming language. Umple currently has integration with Java, C++, C#, Ruby, and PhP. For purposes of this paper, we use Java as an example language. When an Umple module is compiled, the compiler first translates the modeling parts of the module to corresponding code in the same programming language. Umple supports both an Eclipse-based and a web-based development platform. The code provided in this paper can be inserted into the web-based platform for a quick demonstration of the approach.

In Umple, model and code parts can be mixed in the same module or in separate modules. In this paper, we present the structural code and behavioral code separately, but they can also be combined in the same module. As shown in Listing 5, the state machine defines the behavior of the order instances, as represented by the *Order-Status* state variable. In this class, you can see an association between *Order* class and *CreditCard* class, in addition to the state machine.

The state machine defines the states, transitions and actions. Umple also support an arbitrary number of nesting levels and concurrent regions.

By default, the first state specified in a module is considered the start state. Transitions' trigger events are specified as methods that return a Boolean value; true if the event was processed and false otherwise. Transitions' entry, exit and do actions require definition using an action language. Since we are using Java as an action language, such actions are defined in Java. Listing 5 shows two methods; *establishCustomer* and *ProcessCharge*. The method *establishCustomer* is implemented in Java (syntactically it looks very similar to ALF).

Action methods can interplay with the structural and behavioral modeling constructs by calling the interface generated by Umple. For example, to manipulate an association membership, the action method can call the *add* or *remove* interface made available by the language. In our example, the method *addOrderLineItem* is generated from Umple's structural model and is implemented in the generated code in Java as follows.

```
public boolean addOrderLineItem(OrderLineItem aOrderLineItem) {
    boolean wasAdded = false;
    if (orderLineItems.contains(aOrderLineItem)) { return false; }
    Order existingOrder = aOrderLineItem.getOrder();
    boolean isNewOrder = existingOrder != null
    && !this.equals(existingOrder);
    if (isNewOrder) {
      aOrderLineItem.setOrder(this); }
    else {
      orderLineItems.add(aOrderLineItem); }
    wasAdded = true;
    return wasAdded; }
```
Listing 6: Implementation of *addOrderItem*

Umple language supports both bi-directional, Uni-directional and composition associations. Associations can be declared in either of the two participating classes, or alternatively can be grouped under one class named *Associations*. Umple allows all combinations of association multiplicities.

6 Comparison and Discussion

In this section, we compare both ALF and Umple with respect to their motivation, approach to integration with modeling languages, language artifacts, abstract syntax, textual syntax, graphical syntax and semantics. Table 1 summarizes those differences.

Table 1. Comparison between ALF and Umple

	ALF	**Umple**
Textual Representation	YES	YES
Textual Manipulation	YES	YES
Visual Representation	YES (fUML notation)	YES (UML notation)
Visual manipulation	YES, partially through the fUML visual notation	YES. Changes are automatically synchronized since both visual and textual representations are semantically equivalent.
Standalone	YES, but can also be used to specify parts of a UML model specified visually	YES.
Executable	YES	YES
Action Language Syntax	New textual Syntax	New textual syntax embedded in OO language, such as Java, C, C++, Ruby.
Syntax Origin	Industrial proposals from IBM and BridgePoint, refined and managed by OMG	Evidence based, where syntax evolves based on empirical user studies
Activity support	YES	YES through embedding programming language
State machine support	Under development	YES
Meta-Model Conformance	fUML	Umple meta-model that closely resembles a subset of UML meta model

Code Injection	YES but injected code is passed along unparsed	YES but injected code is natively part of the module
Platform	ALF execution engine	The same compiler/interpreter of the embedding Language with a front – end translation of modeling code to the language

6.1 Motivation.

Both ALF and Umple are motivated by the need to bridge the gap between programming and modeling. In the case of ALF, a need was identified for a language to specify actions in UML models while staying within the boundary of the UML abstract syntax. Historically, UML tools used to support specifying those actions using programming languages. However, this has proved hard and inconvenient due to the differences in syntax between UML and those languages.

Umple, on the other hand, came in response to the less than satisfactory adoption of modeling among the software development community. The traditional forward engineering (i.e., code generation from model) approaches were not enough to convince developers to model first. For example, a recent study on the adoption of modeling practices by the open-source community revealed that only 0.3% of commits were XML based [13], suggesting that model based commits are negligible. Even when applied, the produced models tend to be low in abstraction (as detailed as code).

6.2 Approach to Integration with Modeling Languages

ALF provides a textual grammar (specified in BNF) for the executable subset of UML (fUML), which covers mainly activity diagrams. However, instead of modeling those activities using the graphical notation, it proved much more convenient to specify them using the ALF textual syntax. An ALF program is a fUML model that can be executed according to the execution semantics of fUML. The idea is that developing ALF programs may appeal to software developers who are used to programming using textual languages (like Java or C++). However, unlike those languages, which have small specifications, fUML is still considered a large and complex specification.

On the other hand, Umple provides another strategy to help software developers gradually adopt modeling practices. It allows them to slowly replace parts of their code with more concise modeling abstractions that can be translated to equivalent code before execution. For example, bi-directional associations in UML can be considered higher in abstraction than two properties in the associated classes that are opposite to each other (have opposite cross references). By doing so, developers can control when and how fast they adopt modeling approaches. A valid Umple module can be made up of all code, all modeling concepts or anywhere in between.

6.3 Language Artifacts

ALF can be used either as a standalone programming language, specifying the structure and behavior of a system textually, or can be used to specify some behaviors textually in an otherwise visually specified structural fUML model. However, we suspect that the latter use case is still the predominant one.

Umple, on the other hand allows the specification of programs as standalone textual artifacts. The artifacts can include only model abstractions, only code abstractions, or a mix of both. As programmers get more comfortable with modeling abstractions, and understand the corresponding generated code, they may switch (refactor) their code to add more model-based abstractions. It is interesting to mention that the Umple codebase itself is writing in Umple, i.e., in a mix of model and code fashion.

6.4 Abstract Syntax

Although ALF's grammar is defined in BNF, the specification provides a full mapping from that grammar to the abstract syntax of fUML. This means that the AST implied by the BNF grammar is fully mapped to the abstract syntax of fUML, specified as a MOF metamodel. This mapping made it possible, for example, to develop an ALF support for the Papyrus tool using Xtext [30], a technology that allows developing textual syntaxes that map to Ecore (Eclipse implementation of Essesntial MOF). fUML supports structural modeling in addition to some behavioral modeling (e.g., using activity diagrams). There is undergoing work at OMG to add more coverage of the UML behavioral models. ALF also supports code injection (injecting code of other languages, like Java or C).

On the other hand, Umple's abstract syntax is a mix of the BNF grammar of the programming language and one that is defined for the modeling concepts. The latter corresponds to an Umple-specific metamodel that resemble the UML metamodel. In other words, the modeling parts of an Umple program are parsed into an AST that conforms to a BNF grammar which maps to the Umple metamodel. Before compilation, that AST is traversed and converted to code in the same embedding programming language. Umple currently supports structural modeling in addition to behavioral modeling using state machines (activity modeling is achieved by embedding a programming language). Umple supports code injection only for the target language.

To support a close interplay between the code and the model, Umple generates a number of 'methods' from the modeling elements that can be called by the coding elements. For example, the state machine model in Umple will generate methods to support querying the current state, or first state, or the number of states in a particular state machine. Similarly, in the class diagram, the associations will generate methods to return the member instances on either end of the association, and methods to add and remove members. These methods can then be called by the code to implement a specific behavior.

6.5 Textual Syntax

ALF's textual syntax for action definitions uses Java-like conventions. The rationale is to be familiar to OO languages. However, it is not parsed with a Java parse; it has its own parser (for its grammar); and hence stay platform independent.

Umple uses the target language syntax for the non-modeling parts. The rationale is to improve adoption and comprehension (this has also educational benefits where students are likely to already know Java for example). Furthermore, Umple syntax has been driven by comprehensive empirical studies (grounded theory, subject interviews, controlled experimentations, surveys, etc.).

6.6 Graphical Syntax

ALF is just a textual concrete syntax for the fUML abstract syntax, which has a visual concrete syntax (notation) as well. This means that ALF programs can be visualized using the fUML graphical notation, and an fUML model specified visually can also be edited through the ALF textual notation.

Umple code can be manipulated visually or textually. Textual editors include both the modeling abstractions as well as the programming language code. The visual editors show only the modeling abstractions. Since the code can be of any arbitrary object oriented language, this code does not have corresponding visual elements. Edits on the textual or visual side are automatically synchronized, so that both remain in synch.

6.7 Execution Semantics

ALF uses it's a separate execution engine that interprets its programs based on the fUML execution semantics. Despite the fact that fUML's, and more generally UML's, abstract syntax implementation might be different between modeling tools (which is what is happening in practice), the problem may not transcend to ALF, if those tools supported the same BNF grammar, at least as far as parsing is concerned. There is always a risk of differences in execution semantics interpretation.

Umple, on the other hand, relies mostly on the native compiler/interpreter of the embedding programming language; after all modeling code has been translated into that language. This gives it consistency in execution semantics. This is also helped by the fact that only one team is managing the implementation of Umple. Of course, exposing Umple to different implementations may make it suffer similar problems to fUML and ALF.

7 Conclusion

Action languages are motivated by the need to bridge the gaps between programming and modeling. These languages have the potential to raise the abstraction level, and enable software developers to effectively adopt design level abstractions introduced by modeling languages, such as UML.

This paper discussed two such action languages, ALF and Umple. ALF has emerged from the modeling community and is managed by the OMG. Umple has emerged from academia in an effort to bring modeling closer to the mainstream programming community. Both languages have a concrete textual syntax, but they differ in key points.

ALF is designed to function either independently, or within the context of a UML structural model. Umple is designed to function as a complete executable language where structural modeling is part of the language. Moreover, Umple supports two concrete syntaxes, textual and visual, although they are not standard On the other hand, ALF also supports textual and graphical notations, albeit both are standardized,.

ALF adopts a Java-like syntax, where in the case of Umple, the action language syntax is identical to the embedding OO language. The motivation in both cases is to facilitate adoption, and ease the learning curve for new comers.

Action languages have not yet been widely adopted. We are only aware of a handful of courses that offer action language training. In the future, we plan to develop such courses. We also plan to use both languages to develop a large case study where we specify a system's structure and behavior and report on the findings. We also plan to carry an empirical study where we ask software developers to program certain problems in both languages (after some initial training) and report on our findings.

References

[1] France, Robert, and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap." 2007 Future of Software Engineering. IEEE Computer Society, 2007.

[2] Seidewitz, Ed. "UML with meaning: executable modeling in foundational UML and the Alf action language." ACM SIGAda Ada Letters 34.3 (2014): 61-68.

[3] Zaytsev, Vadim. "BNF was here: what have we done about the unnecessary diversity of notation for syntactic definitions." Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, 2012.

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

[5] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., & Völkel, S. (2014). Design guidelines for domain specific languages. arXiv preprint arXiv:1409.2378.

[6] Bichler, Lutz, Ansgar Radermacher, and Andreas Schuerr. "Evaluating UML extensions for modeling real-time systems." Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on. IEEE, 2002.

[7] Badreddin, Omar, and Timothy C. Lethbridge. "Model oriented programming: bridging the code-model divide." Proceedings of the 5th International Workshop on Modeling in Software Engineering. IEEE Press, 2013.

[8] Seidewitz, Ed. "UML with meaning: executable modeling in foundational UML and the Alf action language." ACM SIGAda Ada Letters 34.3 (2014): 61-68.

[9] Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., ... & Terrier, F. (2009, June). Papyrus UML: an open source toolset for MDA. In Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009) (pp. 1-4). .

[10] Neuendorf, David. "Review of MagicDraw UML® 11.5 Professional Edition." Journal of Object Technology 5.7 (2006): 115-118.

[11] Dévai, Gergely, Gábor Ferenc Kovács, and Ádám An. "Textual, Executable, Translatable UML." In OCL@ MoDELS, pp. 3-12. 2014.

[12] Tilley, Scott, Steve Murphy, and Shihong Huang. "5th international workshop on graphical documentation: determining the barriers to adoption of UML diagrams." Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information. ACM, 2005.

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

[14] CanforaHarman, Gerardo, and Massimiliano Di Penta. "New frontiers of reverse engineering." 2007 Future of Software Engineering. IEEE Computer Society, 2007.

[15] Oliveira, Hamilton, Leonardo Murta, and Cláudia Werner. "Odyssey-VCS: a flexible version control system for UML model elements." Proceedings of the 12th international workshop on Software configuration management. ACM, 2005.

[16] Badreddin, Omar, Timothy C. Lethbridge, and Andrew Forward. "A novel approach to versioning and merging model and code uniformly." Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on. IEEE, 2014.

[17] Laforcade, Pierre, and Christophe Choquet. "Next step for educational modeling languages: the model driven engineering and reengineering approach." *null.* IEEE, 2006.

[18] Badreddin, Omar, and Timothy C. Lethbridge. "Combining experiments and grounded theory to evaluate a research prototype: Lessons from the umple model-oriented programming technology." Proceedings of the First International

Workshop on User Evaluation for Software Engineering Researchers. IEEE Press, 2012.

[19] Badreddin, Omar, Timothy C. Lethbridge, and Andrew Forward. "Investigation and evaluation of UML Action Languages." Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on. IEEE, 2014.

[20] Chaves, R. 2009. "TextUML". http://abstratt.github.io/textuml/readme.html

[21] Fayad, Mohamed E., Louis J. Hawn, Mark A. Roberts, and Jerry R. Klatt. "Using the Shlaer-Mellor object-oriented analysis method." IEEE Software 10, no. 2 (1993): 43-52.

[22] Jouault, Frédéric, and Jérôme Delatour. "Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems." OCL@ MoDELS. 2014.

[23] PlantUML modeling tool. Available online: http://plantuml.com/

[24] Sun, Y., Demirezen, Z., Mernik, M., Gray, J., Bryant, B. "Is My DSL a Modeling or Programming Language?", Proceedings of 2nd International Workshop on Domain-Specific Program Development, Nashville, US, pp. 4, 2008.

[25] Lazăr, Codruţ-Lucian, Ioan Lazăr, Bazil Pârv, Simona Motogna, and István-Gergely Czibula. "Tool Support for fUML Models." International Journal of Computers Communications & Control 5, no. 5 (2010): 775-782.

[26] Mellor, S., Balcer, M., "Executable UML: A Foundation for Model-Driven Architecture", Addison-Wesley, 1st edition, 2002.

[27] OMG, Precise Semantics of UML State Machine RFP, ad/15-03-02.

[28] OMG, Precise Semantics Of UML Composite Structures v1.0, formal/2015-10-02.

[29] Badreddin, O. B., Sturm, A., Hamou-Lhadj, A., Lethbridge, T., Dixon, W., & Simmons, R. (2015). The Effects of Education on Students' Perception of Modeling in Software Engineering. In First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS (pp. 39-46).

[30] Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2010.