On a Unified View of Modeling and Programming Position Paper

Ed Seidewitz

nMeta LLC
14000 Gulliver's Trail
Bowie MD 20720 USA
ed-s@modeldriven.com
ed@nmeta.us

Abstract.
In the software community, modeling and programming are generally considered to be different things. However, some software models specify behavior precisely enough that they can be executed in their own right. And all programs can be considered models, at least of the executions that they specify. So perhaps modeling and programming are not actually so different after all. Indeed, there is a modeling/programming convergence going on right now in the Unified Modeling Language (UML) community, with a recent series of specifications on precise execution semantics for a growing subset of UML. But the language design legacy of UML is largely grounded in the old view that sharply separates models and programs, complicating the new convergence. It is perhaps now time to move forward to a new generation of unified modeling/programming languages.

Keywords. Programming languages. Modeling languages. UML. fUML. Alf. Action language. Modeling tools.

I think it is safe to say that most software developers consider models to be something quite different from programs. However, let's consider what a "model" really is.

A model is always *about* something, which I term the *system under study* (SUS). For our purposes here, we can consider a model to consist of a set of statements about the SUS expressed in some *modeling language.* These statements make assertions about certain properties of the SUS, but say nothing about other properties that are not mentioned.

A model thus *abstracts* from the SUS it models by the selection of statements it makes. The model is useful to the extent that the properties not considered by the model are simply not important for the purpose of the model or can be chosen or determined independently of the model. (For further discussion of this view of modeling for software, and it's relation to how modeling is done in other fields, see [6].)

A modeling language can be textual, graphical or a combination of the two. Depending on the expressivity of the language, it may be possible to make statements about an SUS that range from very precise to quite loose. A precise model simply makes more detailed assertions that place tighter, less ambiguous constraints on the SUS.

Consider the simple class model shown in **Fig. 1**, in which the Unified Modeling Language (UML) [2] is used as the modeling language. What does this model mean? That depends on how you *interpret* it.

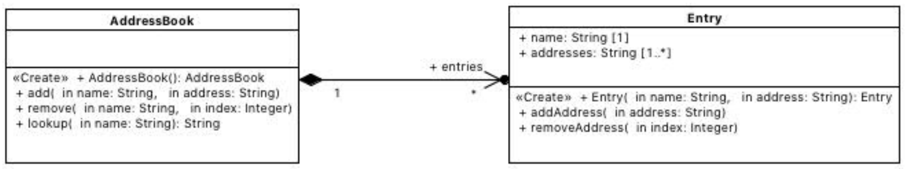
AddressBook
«Create» + AddressBook(): AddressBook
+ add(in name: String, in address: String)
+ remove(in name: String, in index: Integer)
+ lookup(in name: String): String
+ entries
1
*
Entry
+ name: String [1]
+ addresses: String [1..*]
«Create» + Entry(in name: String, in address: String): Entry
+ addAddress(in address: String)
+ removeAddress(in index: Integer)

Fig. 1. UML Class Model of an Address Book

One interpretation is that this is a problem domain model of address books. Under this interpretation, the model states that an address book can have zero or more entries; that each entry must include one name and at least one address; that addresses can be added to, removed from and looked up in an address book; and so forth.

But now suppose that I add behavioral models for the operations specified in the classes in **Fig. 1**. For example, **Fig. 2** shows a UML activity model for the behavior of the AddressBook lookup operation. What is this a model of? Effectively, it is a model of the *computation* to be carried out in order to perform the lookup operation.

Indeed, this activity is within the so-called *Foundational UML* (fUML) subset of UML, for which there are precisely-defined, standard execution semantics [5]. If behavioral models are provided for all the operations shown in **Fig. 1**, then the result is a completely *executable* UML model, with standard semantics.

Now, the diagram in **Fig. 2** may seem to be a somewhat awkward way to specify behavior at this level of detail. However, there is also a standard *Action Language for Foundational UML* (Alf) [1], which provides a fully textual notation for writing such activity models. For instance, the activity drawn in **Fig. 2** can be written as follows in Alf:

```
namespace AddressBook;
activity lookup(in name: String): String[0..*] {
  return this.entries->
           select e (e.name == name).addresses;
}
```

This now looks a lot like code. However, the *meaning* of this textual notation is, in fact, defined by mapping it to executable UML models. The Alf text given above essentially maps to the UML activity model shown in **Fig. 2**, and execution of the Alf text has exactly the semantics of executing the corresponding UML activity. Thus, Alf is not a separate programming language, but, rather, a textual notation for writing UML models.

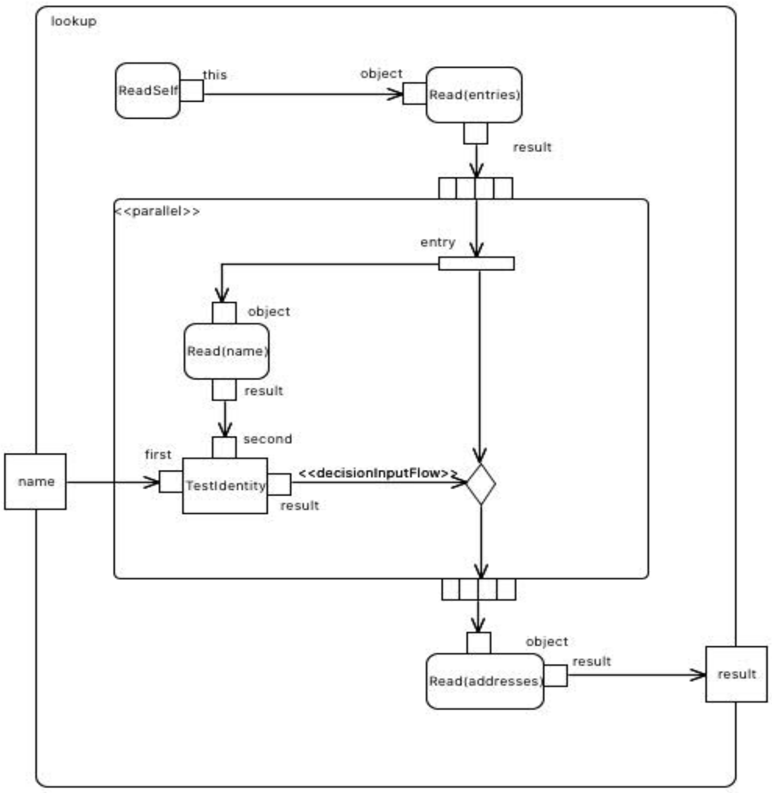

Fig. 2. UML Activity Model for the AddressBook lookup operation

Further, Alf also includes textual notation for structural UML modeling elements. For example, the AddressBook class can be notated in Alf as follows:

```
class AddressBook {

  public entries: compose Entry[0..*];

  @Create public AddressBook();
  public add(in name: String, in address: String);
  public remove(in name: String, in index: Integer);
  public lookup(in name: String): String[0..*];

}
```

So, it is clear that a modeling language does not have to be graphical, and that models can be executable. So how is this different than programming?

In fact, while they are not generally viewed in this way, programming languages are essentially *all* textual modeling languages. Programs written in these languages are precise models of *execution* (where, for simplicity, I consider both data and algorithmic aspects to be included in the term "execution"). Modern programming languages, in fact, allow a programmer to abstract away a great deal of the details that need to be handled to actually execute a program – from language processing, to the operating system, right down to the bare hardware. Indeed, the progression of programming languages from machine language, to assembly language to "higher order" languages can be seen exactly as a progression in increasing the abstraction possible for modeling execution in a program.

From this point of view *all* programs are actually models. And all *executable* models are actually programs. But there are, of course, software models that are *not* programs. Such models allow us to reason about software in ways other than through direct execution and testing.

For example, **Fig. 3** shows a UML object model for an instance of the AddresBook class that has a single entry. This object model can be *deduced* to be correct, based on the class model in **Fig. 1**. Thus, the class model for the Entry class requires that an entry object have a name an at least one address, and the object model in **Fig. 3** satisfies these constraints. One could still draw an object model in which the entry object had no addresses, but this model would be *invalid* relative to the class model given in **Fig. 1**.

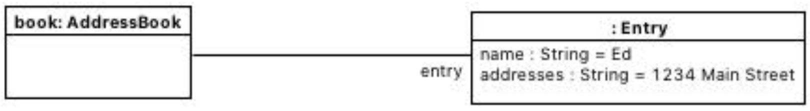

Fig. 3. UML Object Model for an AddressBook

Thus, UML object models can be given precise semantics that are not necessarily execution semantics. There can also be requirements models, architecture models and business models with precise semantics that may or may not be execution semantics. But a major advantage of considering execution in the context of a wider modeling language is that it allows deductive or inductive reasoning on models to be combined with execution and testing, within a single, consistent semantic framework. On the other hand, the semantics of most programming languages are *entirely* execution semantics, other than, perhaps, "static semantic" checks that may be provided by the language compiler (such as type checking).

To maximize the effectiveness of a combined modeling/programming language, one would like a language that takes the best experience with the design of both traditional modeling and programming languages. I would suggest that the following characteristics are particularly important for such a language.

- It should be designed to express both problem and solution domain models, not just as an abstraction of hardware computing paradigms.
- It should have a formal semantics that allows reasoning about models, but also provide a (consistent) execution semantics for models (or segments of models) in which execution behavior is fully specified.

- It should provide a textual notation for representing and reasoning on all types of models, but should also provide graphical notations where those are most appropriate, allowing multiple views of the same model.

With the adoption of the fUML and Alf specifications, as well as a growing set of additional "precise semantics" built on that foundation [3,4], such a convergence of programming and modeling language design is actually already taking place for UML. However, UML was not originally designed with executability in mind and it has become a very complicated language (especially since version 2.0). Furthre, even in the latest base UML standard [2], the specification of semantics is informal and largely imprecise. This makes it considerably more difficult to create separate precise semantics specification for UML, while maintaining general compatibility with the language as it has existed for many years. And it limits how far the community can practically go in achieving the goals I listed above.

It is, perhaps, time we moved on to something better – both for modeling and for programming.

References

1. Object Management Group. 2013. *Action Language for Foundational UML (Alf): Concrete Syntax for a UML Action Language, Version 1.0.1.* October 2013. http://www.omg.org/spec/ALF/1.0.1/
2. Object Management Group. 2015. *OMG Unified Modeling Language™ (OMG UML), Version 2.5.* http://www.omg.org/spec/UML/2.5/
3. Object Management Group. 2015. *Precise Semantics of UML Composite Structures (PSCS), Version 1.0.* http://www.omg.org/spec/PSCS/1.0/
4. Object Management Group. 2015. *Precise Semantics of UML State Machines, Request for Proposals.* http://doc.omg.org/ad/2015-3-2
5. Object Management Group. 2016. *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.2.1.* January 2916. http://www.omg.org/spec/FUML/1.2.1/
6. Seidewitz, E. 2003. What Models Mean. *IEEE Software.* September/October 2003, 26-32.