Position Paper for Campam '06

Ben Denckla*

April 20, 2006

Multi-paradigm modeling means different things to different people. For that matter, "paradigm" and "modeling" may mean different things to different people. Perhaps only "multi" would fail to spark controversy, since it is so generic.

1 Computer-assisted modeling

Let's start by discussing modeling. In particular, we are concerned with computer-assisted modeling. Here I have changed the "CA" in "CAMPaM" from "computer automated" to "computer assisted." This is a mild matter of preference. Although the tools we are concerned with certainly do automate many aspects of modeling, they do not automate all aspects. Therefore, overall, I consider them to assist in modeling through automation rather than automate modeling.

So, what is computer-assisted modeling? It could just mean the use of a computer to assist in the creation of something called a model. For our purposes, though, let's narrow it to the use of a computer to assist in the creation *and execution* of someting called a model. This would exclude use of a CAD drafting tool or even a tool that helps create UML diagrams but does not attempt to do something like generate code from them.

This leads us to the question of what constitutes executability. When a CAD drafting tool renders a document to screen or printer, it is in some sense "executing" the document, just as an interpreter for a traditional computer language might execute a program. In fact, to render to a printer, it is probably generating PostScript code, which will then be interpreted on the printer. So a CAD drafting tool is in fact a code generator!

Yet, let's not consider this the execution of a model. Let's use a more common-sense/domain-specific notion of "execution."

Usually, the CAD document is a model of something like a mechanical system. Therefore executing this model would be doing something like simulating the application of forces to it and seeing how it behaves over time. Printing it is not executing it.

Now, what is a model? A model is an abstract representation of another thing. The representation may be "lossy" or lossless, i.e. the abstraction may lose information or not. A set of ODEs might be a lossy model of a circuit. The regular expression "ab*" is a lossless model of all strings consisting of an 'a' followed by zero or more 'b' characters. In some cases the loss of information is the result of a tradeoff, e.g. "we ignored relativistic effects in this model, making it less accurate but more tractable." In other cases the loss of information is a pure win, e.g. "we ignored time in this model because for our purposes it is irrelevant."

Note that the execution of a model may be lossless or lossy as well. For example, although a set of ODEs may be a lossy model of a circuit, it is a lossless model of the function that is its solution. Yet, the execution of this set of ODEs with a numerical solver is lossy with respect to the function that is its solution. In constrast, the execution of this set of ODEs with a symbolic solver is lossless with respect to the function that is its solution.

When a model and/or its execution is lossy, we often refer to its execution as *simulation*. When a

*Denckla Consulting, 1607 S. Holt Ave., Los Angeles, CA 90035, USA. bdenckla@@alum.mit.edu

model and its execution are lossless, we usually don't refer to its execution as simulation. In this case the behavior during execution is what is being modeled. In this case the model may be referred to as a *program* (often in a *domain-specific language* (DSL)). We'll call this form of execution *interpretation* even though it may consist of compilation followed by interpretation of machine- or byte-code. From this perspective, simulation transforms a model of x into another model of x, whereas interpretation transforms of a model of x into x.

Whether something should be called a model or a DSL program, and whether its execution should be called simulation or interpretation (or just "execution") is a matter of taste and context. In my context, that of research and development of block diagram languages, I tend to think of block diagrams as programs, i.e. I think of their behavior during execution as the final goal or product. In contrast, a Simulink user might find it very strange to think of a diagram as a program, especially a continuous diagram. Rather, they would think of it as a model whose execution produces a simulation trace (another model) of a physical system that is the final goal or product.

2 Multi-paradigm modeling

Okay, now that we have some idea of what computer-assisted modeling is, let's discuss what multi-paradigm modeling is. A modeling paradigm describes a style or family of modeling languages. For instance, state transition diagrams are a modeling paradigm that encompasses languages like StateMate and StateFlow. Discrete block diagrams are a modeling paradigm that encompasses languages like discrete Simulink and Ptolemy SDF. Thus multi-paradigm modeling is modeling in multiple styles or sub-languages underneath one umbrella language. For example the combination of Simulink and Stateflow forms a multi-paradigm modeling language.

A major challenge (perhaps *the* major challenge) in multi-paradigm modeling is to define how the different paradigms interact. E.g., are all paradigms available at once, i.e. can syntactic elements of all paradigms be mixed freely, or is the use of a paradigm restricted to a syntactically bounded sub-model? Must a new a coordination language be invented to bridge existing paradigms, or can one of the existing paradigms serve this function? What language is used to define the underlying semantics (and the overlying syntax) of all the paradigms?

Just behind the surface of these (daunting) challenges is a more basic challenge, which is that in order to define how different paradigms interact, one must first have good definitions of the individual paradigms, which are rarely available. Many modeling languages are visual, so they are unable to benefit directly from the mature techniques that have developed for the definition of textual languages. Graph grammars have emerged as a promising technique for the definition of visual languages. Although many modeling languages like block diagrams have been used for decades for human communication, it is challenging to move their semantics from the informal level that humans require to the formal level that an executable implementation requires. An even higher level of semantic formality is needed if we want to be able to reason logically (e.g. prove things) about models.

Returning to the challenges of multi-paradigm modeling, I will conclude by citing some research I feel is relevant. As much as possible, modeling should try to draw upon the rich, mature research that has been done in traditional (textual) programming languages.

I cite [2] as an excellent example of a multi-paradigm programming language in which syntactic elements of all paradigms can be mixed freely, and the underlying semantics are defined in terms of a small kernel language with structured operational semantics.

I cite [3] to show Haskell to be an excellent example of a multi-paradigm (actually, dual-paradigm) programming language in which one paradigm may be used in syntactically bounded ways inside a dominant paradigm. In this case the imperative paradigm can be used boundedly inside the declarative (in particular, pure functional) paradigm. The syntactic bounding happens via "do" syntax and restrictions imposed by the type system. The semantics of the imperative

paradigm are defined by desugaring to the enclosing (dominant) pure functional paradigm.

I cite [1] as an example of of a multi-paradigm modeling language whose semantics are defined using an underlying reference language. Unfortunately the semantics of the reference language are not discussed in this article.

References

[1] G. Frick and K. D. Müller-Glaser. Semantic integration of modelling languages based on a reference language. In *Proc. of the 4th IMACS Symposium on Math. Modelling (MATHMOD)*, 2003.

[2] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, USA, 2004.

[3] P. Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.