Charles Simonyi, the recently returned space tourist, is an intriguing character. So much so that a lengthy, and suitably fascinating, profile was published in *Technology Review*. (See http://www.technologyreview.com/Infotech/18047/). A sidebar, "Intentional Programming Explained" (http://www.technologyreview.com/Infotech/18047/page13/) contained a throwaway line that suggested that UML models could not be executed. This is simply not true, as I will outline in this white paper. The white paper focuses on embedded systems, because that is the market in which I work, but the ideas apply (and have been applied) in many domains.

For more background on the discussion triggered by *Technology Review*, and I personally got involved, please see the appendix.

Embedded Systems in UML

Stephen J Mellor
StephenMellor@StephenMellor.com

Freeter

Executable UML models allow the designer to verify the design before any code is written. A translation phase, using a set of rules, builds a system in a strict and repeatable manner. The engineer has full control over these rules to enable generation of the most appropriate code for a particular architecture.

Rather than bit-fiddling with the generated code, a more structured and repeatable process is now used. If there is a problem with the behavior of the application, then the model is changed accordingly, and if there is a problem with the performance of the code, then the rules are adjusted. This separation of the application (using the design tool and verify tool) from the architecture (processing the design rules) results in a more maintainable and efficient system.

Using UML in Embedded Systems

There are many ways in which the UML can be said to be used. Most people use UML informally. That is, they *sketch* out a few diagrams on the proverbial beer mat and discuss their abstractions with their peers (*sic*). From there, coding proceeds directly. Martin Fowler's book, *UML Distilled* [1], takes this point of view.

Others use UML as a *blueprint* that specifies software structure. There is an intended near one-to-one correspondence between the UML diagrams and the code. A tool can generate code frames from these models, and the developer fills in the rest using the models as a guide. Note that if the developer finds a better solution while coding, the model will no longer reflect the code. Users of this approach must have a detailed understanding of their code structure and the corresponding elements in UML. They'll find *UML in a Nutshell* [2] useful in this (though the authors do not explicitly promote this approach.)

Others go further. The developer can add code directly to the model making the model *executable*. Note that if the generated output is changed by hand, the model no longer reflects the code. This may then lead to a desire to "round trip" and generate a model from the code, which only makes sense so long as the model bears a one-to-one relationship to the code. This point of view is promoted by Bruce Powel Douglass. [3]

The fourth use is *translatable* models that have action language as a part of the model. The action language is executable, but it can also be translated into *any* implementation. A given model may be translated into an implementation that has many tasks or one, many processors or one, or even into different hardware/software partitions. In contrast to the second and third uses, there is no necessary correspondence between the structure of the model and the structure of the implementation except that the behavior defined by the model must be preserved. Mellor and Balcer hold this viewpoint [1], as described in *Executable UML: A Foundation for Model-Driven Architecture* [4]

Rationale

In a code-based approach to development exemplified by using sketches or blueprints, we often construct documents or notes that are reviewed with the clients and experts. Their efforts clarify just what is to be built, expand our (the developers') understanding of the problem and associated technology, and remove errors. However, even with all our best efforts and theirs, it is often the case that most errors are discovered during the first system build.

The inevitable result is that we must scramble to fix the errors in the short window before delivery. The sixteen-hour days so necessitated are painful for us, but they are also painful for the business as it is forced to explain why a product, already marketing at fever pitch is not available for customers.

Moreover, the verification gap—the difference between when an assertion is made and when it can be shown to be true or false—between those documents and something that runs can be several months, during which time we expend effort on building the wrong thing. It is this fact that partially explains Barry Boehm's famous curve, which shows an exponential cost-to-fix as the project proceeds from conception to deployment. [5]

These problems can be addressed by building executable models. The key word here is executable. An executable model, because it executes, can be shown to be correct or not, right from the moment it is built. Moreover, because the model can be supplied with test cases, we can close the verification gap, a key reason for the deprecation of modeling by many of signatories to the Agile Alliance [6].

Translation of models also offers platform independence—the notion that we can build models without committing to a software architecture. This is the essence of translatability: by separating out the application logic from the manner in which it is implemented, we can specify the behavior of a system so that it can be targeted to several implementations, some of which have not yet been conceived.

Building an Executable Model

Application models are executable; it is this fact that allows us to extend verification up the design flow so that verification may occur before coding. An executable model can be described in as few as three primary models as illustrated in Figure 1. Other diagrams are used to support construction of the primary models, to present overviews, and for translation, [For a complete description, see 4; for an extended

[1] Martin Fowler has independently arrived at this characterization, though he does not call out "translatable" as different from "executable".

example, see 7] The first part is the declaration of the conceptual entities in a particular subject matter; these are represented using a Unified Modeling Language (UML) class diagram. It is critical to remember that this use of a "class diagram" says nothing about the software structure. In an executable model, a "class diagram" is only a convenient representation for a conceptual grouping of information and behavior.

Behavior over time is represented using a subset of a UML state chart diagram (the second part of Figure 1). There is a state-chart diagram for each object of a class[2]. All the dynamic behavior in an application model takes place as a part of a state machine.

An executable model is meaningless without rules that define execution. In executable UML, each state machine executes concurrently with all other state machines. They communicate by sending signals that define precedence relationships between sequences of actions. Just as with class diagrams, the state chart diagram does not imply an implementation; signals may be implemented in any manner that preserves the desired precedence of actions.

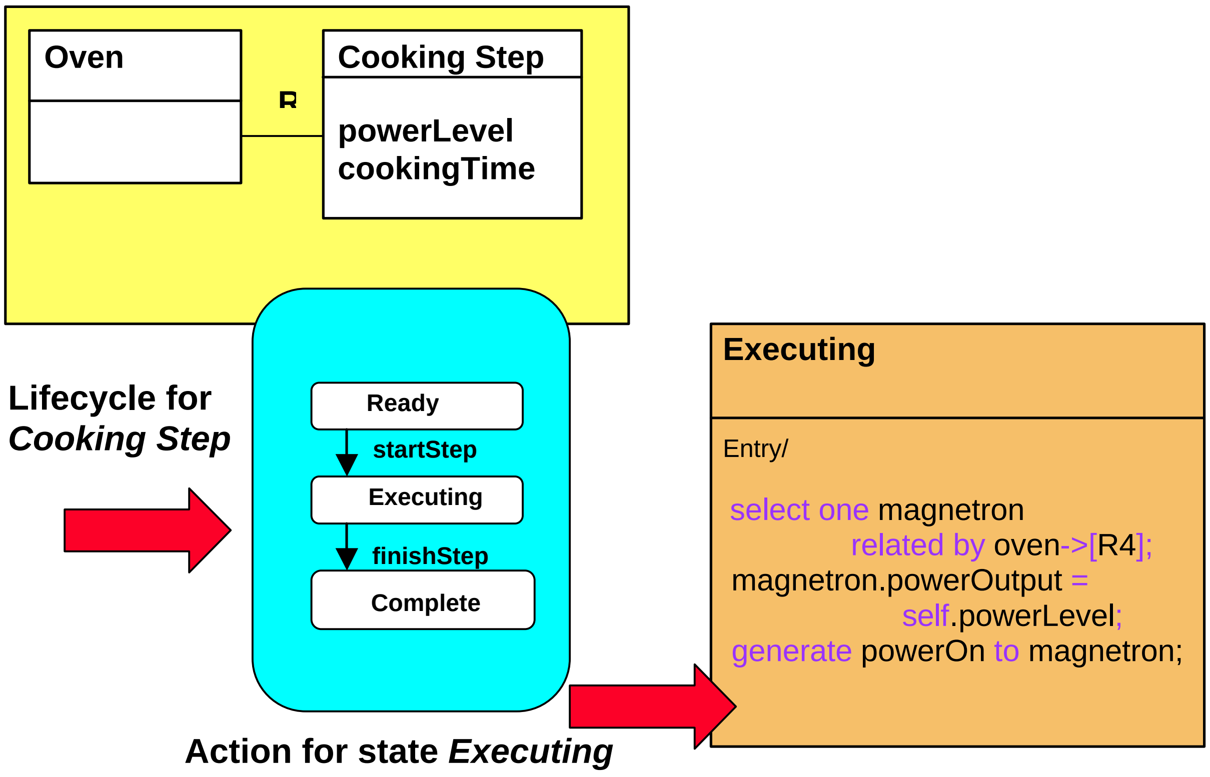


Figure 1: An application model example

Actions are core artifacts of translatable UML because they define behavior, and are implementation independent, in that they do not assume a software structure. Consider, for example, a function that sums the total time used in the last *n* phone calls. Looping through a linked list of recent calls is one way to implement this, but that definition relies of a specific software structure: a linked list. UML actions are

[2] More accurately, there may be one state chart for each instance and there may be one state chart for the behavior of the collection of instances as a whole. Moreover, instances may participate in subtyping hierarchies. See [4] for more details.

cast so we first get the times for each of the calls, then sum them up. This subtle difference in expression allows us to access stored data in any way we choose, so as to get the best performance.

Verifying an Executable Model

Verification of application models is exactly that: It verifies the behavior of the application models and nothing else. It does not check that the system will be fast enough or small enough; it checks only that the application does what you (your client and your experts) want.

Verification is performed by model execution. When we construct a model, such as that shown in Figure 1, we declare the types of entities whose behavior exists in the system, but models execute on instances. To test the behavior of the example in Figure 1, therefore, we need first to create instances. Action language can do that; it can also set up initial values for those instances.

A special TestClass can be set up with a series of states for each test. The first state creates the object instances on which the test will execute. Then, we send a signal to one of the created objects to cause it to begin going though its lifecycle. Of course, actions in that object may be signal-sends that cause behavior in other objects. Eventually the cascade of signals will cease and the system will once again be in a steady state, ready for another test.

Each test can be run interpretively. Should the test fail, the model can be changed immediately and the test rerun. Care should be exercised to ensure the correct initial conditions still apply. When each test run is complete, we need to establish whether it passed. This can also be achieved using action language that simply checks expected values against actuals. A report can be produced in regression-test mode that states whether each test passed or failed.

It is useful to be able to 'run' the whole test at once; to 'jog' through each state change, and to 'step' through each action. Animated state machines make for a cute demo, but we have found it more useful to see a trace of actions, which can be examined automatically later.

Selecting test cases is a whole other topic. There are many theories on how best to test a system ranging from "just bang on it till it breaks" through to attempts at complete coverage.

Translating an Executable Model

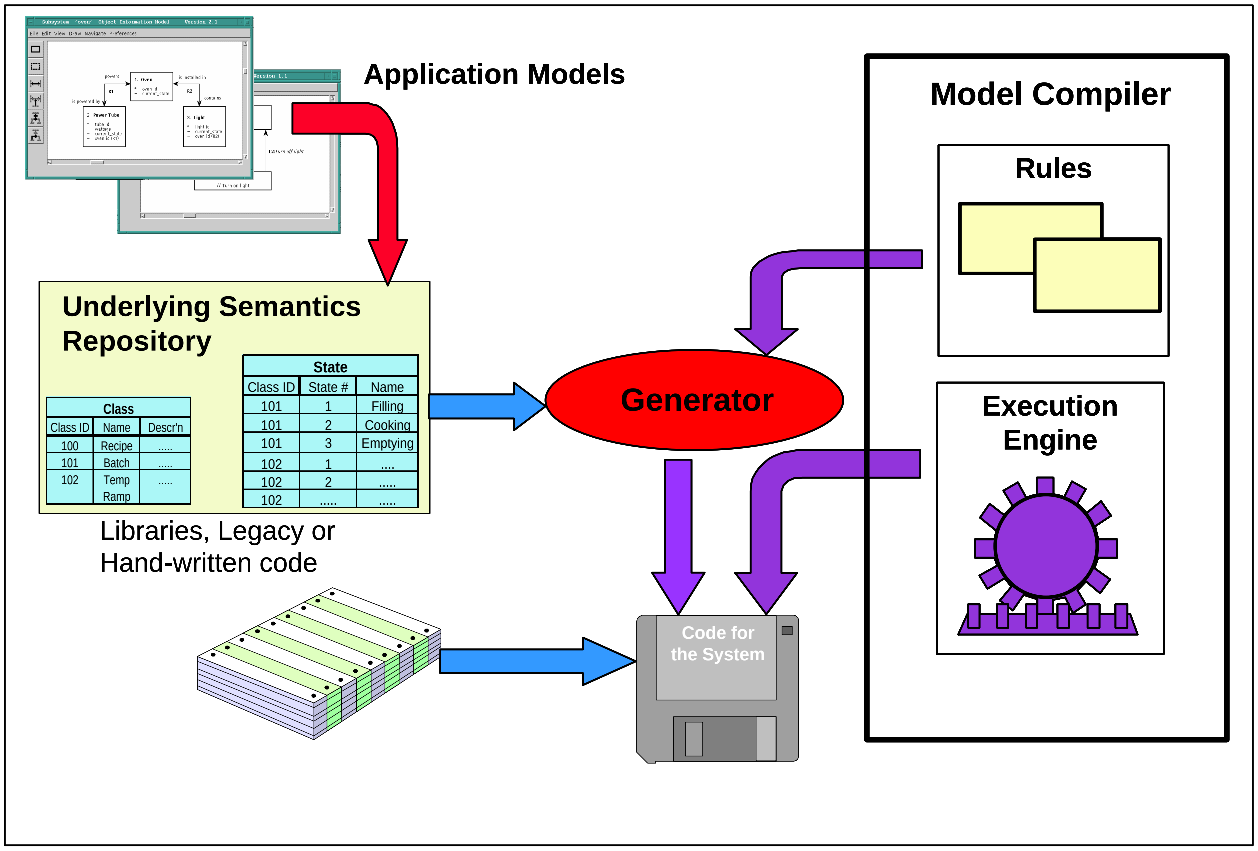
A *model compiler* comprises a collection of reusable components together with a set of formalized rules on how to embed the application into it. The reusable components should include ways to access program data such as lists and trees, to store persistent data, to sequence actions, to activate and deactivate tasks and so on. The formalized rules show the usage of the reusable components in context. For example, if a task synchronizes with others by sending messages of a particular form, then the formalized rule shows the construction of the message, and the call to the message handling service. The content of the message is left as a placeholder to be filled in from the application.

We base the concept of a model compiler on the observation that to understand the fundamental architecture of a system, we need only to understand the structure of each of its prototypical elements and how they interrelate. For example, to understand the *design* of a Unix-based application, we need only understand pipes and filters. We don't need to understand the details of the semantics of an application–in fact they are a hindrance.

The details of model compilation are beyond the scope of this paper, but Figure 2 summarizes them. The meaning of the application is modeled and stored in a repository, as shown on the left-hand side of the diagram. This meaning includes all the concepts described in the sections above: class, state, action, and so on. (The graphical layout is also stored, but for model translation this is not relevant.) For example, the action language `Create object instance myOven of Oven;` is stored as a CreateObjectAction, a reference (myOven), and a reference to the class Oven. This may be translated into `myOven = new Oven;` some list management, or a `malloc` of the appropriate number of words, or even a VHDL entity.

The model compiler is on the right-hand side of the diagram. It consists of a library of reusable elements, such as list or memory management functions, and a set of rules. Each rule generates text. For example, the rule that creates objects would read CreateObjectActions and create an object in C++ for each one. The rule language has been standardized by the Object Management Group, and can be found in [8].

The result of applying the rule (the Generator oval in the center of the diagram) is text. This text can be compiled with the libraries in the model compiler, other libraries, legacy code, and so on. The result is a program, or set of programs, that can be compiled, linked, and run.



Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....

State		
Class ID	State #	Name
101	1	Filling
101	2	Cooking
101	3	Emptying
102	1	....
102	2	.....
102	.....	.....

Figure 2: The structure of Model Compilation

There is little value, therefore, in thinking of a design as the repetitive use of the design decisions as they relate to the application components. This is time-consuming, tedious, and an invitation to unnecessary and unwanted creativity. Instead, we should think of design as the definition of the reusable components and how to use them in the context of an application. Note that this approach scales well, so that a project can make best use of experienced designers, because the model compiler contains only information about the system design. The approach is also platform independent, meaning that a model expressed in this manner can be translated onto any platform. All you need is its model compiler.

For example, let us consider an implementation for the state machine diagram based on the construction of a state-event matrix which, :for the CookingStep, might look like this:

```
static const StateNumber_t Cooking_Step_StateEventMatrix[3][3] = {
    /* Column Headings */
    /* ‘startStep’, ‘interrupt’, ‘finishStep’ */
    /* Row 1: Cooking_State_1 'Ready' */
    { Cooking_State_2,  IS_IGNORED,  Cooking_State_3 },
    /* Row 2: Cooking_State_2 'Executing' */
    { IS_IGNORED,  Cooking_State_1,  Cooking_State_3 },
    /* Row 3: Cooking_State_3 'Complete' */
    { IS_IGNORED,  IS_IGNORED,  IS_IGNORED }
};
```

Note that the matrix has entries for every combination of the arrival of an event in a state. In some cases, such an event can be known to be an error; in others (such as repeated pressing of the Start button), it can safely be ignored. This information can be filled in using an appropriate interface, though it is not absolutely necessary.

The dispatching code for the CookingStep is as follows:

```
   this_state = instance->m_object_state;
   next_state = CookingStep_StateEventMatrix[ this_state ][ event_number ];
   if ( next_state <= 3 )
      {
        /* Execute the state action and update 'object state' */
        ( *CookingStep_Actions[ next_state ] )( instance, event );
          instance->m_object_state = next_state;
      }
```

Note that we have hard-coded the number of events as 3. We could have followed standard coding practice and defined a constant, but this only adds work for the compiler in code that is generally never read.

The rule that created this dispatching code looks like this:

```
        this_state = instance->${object_state.name};
       .invoke sem_name = GetStateEventMatrixName( object )
       next_state = ${sem_name.result}[ this_state ][ event_number ];
       if ( next_state <= ${max_state.number} )  {
         /* Execute the state action and update 'object state' */
         (*${action_array_name.result}[next_state] )( instance, event );
        .select any final_state related by state_model->SM_STATE[R501]
                where ( selected.isFinal != 0 )
        .if ( empty final_state )
           instance->${object_state.name} = next_state;
```

The first line is simply text—the output is exactly the same as the input.

The second line begins with a period (.), and is an instruction to the rule-language processor. It invokes a function to find the name of the state-event matrix associated with the class being manipulated—in this case, the CookingStep. (The parameter to this function is called `object` for historical reasons.)

The third line contains a string substitution function `${ …}`. This returns the string `CookingStep_StateEventMatrix`.

The fourth line also contains a string-substitution function to find the maximum value of the index into an array of states. (You can see more clearly now why the use of a constant in the generated code is not a coding problem.)

After a comment, we produce the code that jumps into an array of function pointers containing the work to be carried out in the action.

Then we find the final state of the state machine, if any.

And then—if none such exists—update the state of the instance. Different processing is required if there is a final state, but you get the picture.

For your interest, here is the code for one of the actions, with the original action shown as a comment. The strange names (`MW_MT`, for example) are generated from abbreviations for each of the class names. MW, for example, identifies the MicroWave subsystem, and MT identifies the Magnetron.

The last two lines create an event (the ninetieth, apparently) and sends it on.

```
/* SELECT ONE magnetron RELATED BY oven->MW_MT[R4] */
    magnetron = oven->mc_MW_MT_R4;

  /* ASSIGN magnetron.powerOutput = SELF.power_level */
   magnetron->m_powerOutput = self->m_power_level;

  /* GENERATE MW_MT1:'power_on'() TO magnetron */
  {
    Event_t *event90=NewEvent((void*) magnetron, &MO_MW_MT_Event1_sc );
    SendEvent( event90 );
  }
```

There is now a standard language for writing these rules. It can be found in [9]

Modifying the Architecture

One reason why you might modify the architectural rules is performance. If you do modify the rules, you have to verify they are correct. One approach is to compile a test model using the modified rules and see that its output is the same as the unmodified version. Alternatively, we can also borrow an old (programming-language-) compiler writer's trick: Build a test model that exercises every element of the language, in this case the modeling language. Finally, we can inspect the output, either manually, or by automatic comparison to an example that is known to be correct. For example, if we know we intend to generate a line `DigitalOutputFunction( bitValue, register);` then we can write a short script to compare outputs.

An example of modification to the model compiler for performance purposes is memory management. Rather than using a standard, supplied, linked list to hold instances, you may choose to allocate fixed-size blocks of memory and maintain an index. This will maintain constant-speed memory allocation, and reduce fragmentation, at the possible expense of more memory. Your choice.

We need also to extend that test model to account for elements that are marked to use new rules. For example, using an array instead of a list will require some correlation between the array index and the objectIDs used to identify other instances. Getting these rule interactions right can be a chore, but it’s better than integrating developers’ disparate creations at the end of the project!

In more cases that you might suspect, there is no reason to modify the rules. Over the ten years since we have been producing fast, small-footprint model compilers for embedded systems, we have extended and enhanced the rules significantly. Because our “tricks” are expressed as rules, they can be propagated across the system quickly without increasing the size of code caused by duplication.

For example, although the CookingStep generates a sparse state-event matrix, a more realistic example might have a sequence of a dozen states, each triggered by a different event. This would lead to a 12x12 matrix, with 132 empty cells. To save memory, we might choose instead to use a linked list of triples (`CurrentState, Event, NextState`) and search it until we find a match for `CurrentState` and `Event`. We then invoke the action for `NextState` and change state to it. This involves a rewrite of the rule for the creation of the state-event matrix (not unreasonably), but once done it applies to all state machines.

We can also write a rule that determines if a matrix will be sparse or dense, and pick which generation rule to apply.

Conclusion

This translation technology relies on the separation of application from system architecture. It is analogous to separating a programming language from the program. We describe the application in a subset of UML that is sufficiently precise to be executed. One product, the Verifier, interprets the models and a set of initial conditions to allow for immediate online debugging of requirements. Once the models are correct, they are translated into implementation using a model compiler. If you wanted to deploy a fast, distributed, transaction-safe with rollback environment, you can use a model compiler that implements this. If you wanted to deploy to a small, embedded real-time environment, you’d use a different one. Model compilers exist for web apps, using Java; generating hardware; and so on.

The model compiler cannot know everything. For example, UML has attributes, but it doesn’t know whether they’re transient or you need to store them. The model compiler you select will provide some number of storage mechanisms (possibly zero), and it will export a set of “mark types” that you can use to select which mechanism to use. These are analogous to compiler flags, or the ‘inline’ keyword in C-based languages (though we would make ‘inline’ a mark type and then list externally all the functions to be compiled inline.)

These three things—models, model compilers, and marks—are sufficient to build applications, define system architectures, and show how one relates to the other, respectively. The technologies are in place, working, in use, and yielding results. Today. You can find out more by reading the appendix, which also refers to two of my books.

The value of this approach lies in the translation rules. Although execution is important and a major step forward from using the UML as a sketch or blueprint, it does not enable platform independence. While this term is often bandied about, as usual in this business, it has several meanings. We mean it in exactly the way real-time and embedded developers mean it: that we can write a “program” without specifying how the software is to be organized. This allows us to write it once and maintain a single model, even as our product’s technology changes.

Moreover, we can use the model compiler repeatedly in several unrelated systems, thus reducing costs significantly.

Acknowledgments

Cortland Starrett of Mentor Graphics Corporation for the examples and acting as reviewer.

References

[1] *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third Edition by Martin Fowler (Addison-Wesley Professional)

[2] *UML in a Nutshell* by Dan Pilone, Neil Pitman (O’Reilly Media)

[3] *Real Time UML: Advances in the UML for Real-Time Systems*, Third Edition, by Bruce Powel Douglass (The Addison-Wesley Object Technology Series)

[4] *Executable UML: A Foundation for Model-Driven Architecture*, Mellor and Balcer, (Addison-Wesley Professional)

[5] Boehm and Basili, Software Defect Reduction Top Ten List. *Computer Vol. 34*, January 2001

[6] www.aanpo.org The Agile Alliance website

[7] *Executable UML: A Case Study*, by Leon Starr

[8] http://www.omg.org/cgi-bin/doc?ad/2006-09-03

Appendix: Some History

I was pointed to the offending article by the Chairman of the OMG, Dr Richard Soley. Various promoters and vendors of executable and translatable models wrote a letter to the publication, which you can read at http://www.omg.org/trletter.html. We received a gracious reply from the author, Scott Rosenberg, who apologized for the error, explaining that a last-minute edit "was trying, imperfectly, to say that there is no *automated* or "just press this button"-style way to transform a UML diagram into working software."

This is not true either, so I dashed off a short note that said, in essence, "the hell we can't." Again, the author responded graciously asking for more information. By this time, we were holding a conversation by email, and the tone is informal and personal. I have cleaned it up slightly, as we all read more than enough formal papers.

Here is my letter:

This work has been going on for at least fifteen years. For some history, you might find http://www.devx.com/enterprise/Article/10717/1954?pf=true interesting.

During this time, several companies have sprung up that employ this technology. Kabira Technologies (http://www.kabira.com/) sell a suite of model compilers that allow you to build UML models and translate them directly into code. They employ an abstract typed action language that "knows about" UML, while complex comprehensive "details" are left to the model compilers. Kabira's targets market is fast, distributed systems that require transaction safety with rollback (vertical markets include telephone billing, stock trades, that sort of thing.) Ensuring transaction safety is not at all easy, and it is extremely error prone. But Kabira lets you identify the boundaries of the transaction (if different from default assumed transactions), and it does all that work for you. They get about 7-10 lines of difficult C++ for each line of action language. I am a member of their Technical Advisory Board.

My own company, Project Technology, founded in 1983, on the other hand, focused on real-time and embedded *horizontal* markets. Our flagship model compiler produces small-footprint C for embedded devices. The first version of this was released ten years ago. Since then, it has undergone repeated optimizations that make the code faster and smaller *than can be produced by hand*. Remember: The first SPEEDCODING/FORTRAN compilers were initially suspected of being slow, even though produced better code than 99% of developers. Details on the model compilers below, where I describe the technology a bit.

Project Technology was sold to Mentor Graphics, Corp. in 2004. You can find out about their products at http://www.mentor.com/products/embedded_software/nucleus_modeling/index.cfm. The link on the model compilers is most appropriate for automatic generation. Mentor has recently changed its strategy from horizontal ("we have a UML tool") to vertical markets such as telecom and automotive. The goal is to provide specific execution and translation capabilities in those markets, using UML, of course, but not trying to support all the uses of UML in those markets, let alone spreading outside the real-time and embedded space. My role has changed from employee to consultant on the horizontal issues, which I explain as I outline the technology.

Even after fifteen years, there remain flies (or "horizontal issues") in this ointment. Specifically, the tools do not fit together. Even though they all use UML, each can use different subsets of UML, and as the UML was defined without execution in mind, there is no guarantee that two interchanged models will execute the same way. There is no common action language (!). Accordingly, in 1998, I set out to do the following with the Object Management Group.

First, in 1998, I joined the fast-growing camp of people and organizations that understood the value of a shared general notation and admitted defeat in the notation wars. Then I set about making the UML useful by proposing a standard action language. This proved too big to be swallowed whole, and instead we decided to produce a standard "action model." (Previously, the UML had six actions, the seventh being "Uninterpretted String." Nowhere near enough to execute.) The action model defined a complete set of primitives, but no syntax for them. Great care was taken to ensure that these actions could also be translated to hardware.

Then I promoted a standard for translating models to text. The examples in the appendix use our own language; there is now a standard (and I need to get to work updating the examples!) Then, in 2004, I started up a group to agree on a common subset of UML and to define its semantics. This group is well on its way, and we hope to finish by the end of 2007. Finally, finally, we need a standard action language, finally. I expect to present the Request for Proposal in June 2007.

These elements—a defined executable subset of UML, a way to translate a model into text (or into another model), and a defined action language are enough to allow a team to produce both models and model compilers for general use across multiple tools and multiple computational environments.

We're not there yet, but I cannot overemphasize that we can do this *today*. We just don't have a way to interchange a common executable subset of UML models. As you know from experience with UML, the perception of a standard is enough to increase acceptance hugely. And greater acceptance means more tooling, and that means faster and more productive software and hardware generation with increasing reuse. So, Yes, there *is* an *automated* or "just press this button"-style way to transform a UML diagram into working software.