Chapter 2

Why Systems-on-Chip needs *More* UML like a Hole in the Head

Stephen J. Mellor, John R. Wolfe, Campbell McCausland

Accelerated Technology, a Division of Mentor Graphics
Tucson, AZ, USA

Abstract Let's be clear from the outset: SoC most certainly can make use of UML; SoC just doesn't need more UML, or even all of it. Rather, we build executable models of system behavior and translate them into hardware and software using a small well-defined core of UML. No more UML!

2.1 Problem and Solution

2.1.1 A Caricature of the State of the Practice

In this section, we caricature today's development process so as to illuminate the problems that we address in our approach.

Partition: At the beginning of an SoC project, it is common for the hardware and software teams to build a specification, usually in natural language. This defines a proposed partitioning into hardware and software so the two teams, with different skills, can head off in parallel.

Verifying the hardware/software partition requires the ability to test the system, but it takes months of effort to produce a prototype that can be executed. Yet we need to execute the prototype before we will know whether the logic designers and the software engineers have the same understanding of the hardware/software interface. We also need to run the prototype system before we can measure its performance, but if the performance is unacceptable, we must spend weeks changing the hardware/software partition, making the entire process circular.

Interface: The only thing connecting the two separate teams, each with different skills, heading off in parallel, is a hardware/software interface specifica-

G. Martin and W. Müller (eds.), UML for SOC Design, 17–36.

tion, written in natural language. Two teams with disparate disciplines working against an ambiguous document to produce a coherent system. Sounds like a line from a cheap novel.

Invariably, the two components do not mesh properly. The reasons are myriad: the logic designers didn't really mean what they said about that register in the document; the software engineers didn't read all of the document, especially that part about waiting a microsecond between whacking those two particular bits; and of course, the most common failure mode of all, logic interface changes that came about during the construction of the behavioral models that didn't make it back into the interface specification.

Integration: So what's a few interface problems among friends? Nothing really. Just time. And money. And market share. We've been doing it this way for years. It's nothing a few days (well, weeks) in the lab won't solve. Besides, shooting these bugs is fun, and everyone is always so pleased when it finally works. It's a great bonding experience.

Eventually, the teams manage to get the prototype running, at least well enough that they can begin measuring the performance of the system. "Performance" has a number of meanings: Along with the obvious execution time and latency issues, memory usage, gate count, power consumption and its evil twin, heat dissipation, top the list of performance concerns in many of today's embedded devices.

There's nothing like a performance bottleneck to throw a bucket of cold water on the bonding rituals of the integration heroes. Unlike interface problems, you don't fix hardware/software partition problems with a few long nights in the lab. No, this is when the engineers head back to their desks to ponder why they didn't pursue that career as a long-haul truck driver. At least they'd spend more time at home.

2.1.2 Problems to Address

Given this is how we operate, what are the problems? They are, of course, deeply interrelated.

Partition: For the early part of the process, logic designers and coders are actually doing the same thing, just using different approaches. Distilled, we are all involved in:

- Gathering, analyzing, and articulating requirements for a product.
- Creating abstractions for solutions to these requirements.
- Formalizing these abstractions.
- Rendering these abstractions in a solution of some form.

- Testing the result (almost always through execution).

So, can't we all just get along? At least in the beginning.

Interface: It is typical to describe the interface between hardware and software with point-by-point connections, such as "write 0x8000 to address 0x3840 to activate the magnetron tube;" and "when the door is opened interrupt 5 will be activated." This approach requires that each connection between hardware and software be defined, one by one, even though there is commonality in the approach taken. Using a software example, each function call takes different parameters, but all function calls are implemented in the same way. At present, developers have to construct each interface by hand, both at specification and implementation time. We need to separate the kinds of interface (how we do function calls) from each interface (the parameters for each function call) and apply the kinds of interface automatically just like a compiler.

Integration: We partitioned the system specification into hardware and software because they are two very different products with different skills required to construct them. However, that partitioning introduces multiple problems as we outlined above so we need to keep the hardware and software teams working together for as long as possible early in the process by maintaining common elements between the two camps. That which they share in common is the functionality of the system—what the system does.

A significant issue in system integration, even assuming the hardware and software integrate without any problem, is that the performance may not be adequate. Moreover, even if it is adequate at the outset, version 2.0 of the product may benefit from shifting some functionality from software to hardware or vice versa. And here we are, back at the partitioning problem.

2.1.3 Towards a Solution

Happily, the problems described above do suggest some solutions.

Build a Single Application Model: The functionality of the system can be implemented in either hardware or software. It is therefore advantageous to express the solution in a manner that is independent of the implementation. The specification should be more formal than English language text, and it should raise the level of abstraction at which the specification is expressed, which, in turn, increases visibility and communication. The specification should be agreed upon by both hardware and software teams, and the desired functioning established, for each increment, as early as possible.

Build an Executable Application Model: Indeed, the specification should be executable. The UML is a vehicle for expressing executable specifications now that we have the action semantics of UML 1.5 and its latest version, the action

model of UML 2.0. This action model was expressly designed to be free of implementation decisions and to allow transformation into both hardware and software. Executable application models enable earlier feedback on desired functionality.

Don't Model Implementation Structure: This follows directly from the above. If the application model must be translatable into either hardware or software, the modeling language must not contain elements designed to capture implementation, such as tasking or pipelining. At the same time, the modeling language must be rich enough to allow efficient implementations. Chief among the topics here is concurrency, both at the macro level (several threads of control as tasks or processors and blocks of logic that execute concurrently) and at the micro level (several elements in a computation executing at once).

In other words, we need to capture the natural concurrency *of the application* without specifying an implementation.

Map the Application Model to Implementation: We translate the executable UML application model into an implementation by generating text in hardware and software description languages. This is accomplished by a set of mapping rules that 'reads' selected elements of the executable UML application model and produces text. The rules, just like a software compiler generating a function call in our example above, establish the mechanisms for communicating between hardware and software according to the same pattern.

Crucially, the elements to be translated into hardware or software can be selected by marking up the application model, which allows us to change the partition between hardware and software as a part of exploring the architectural solution space.

All this provides a way to eliminate completely the hardware/software interface problems that are discovered during the initial integration, and to allow us to change the partition between the hardware and software in a matter of hours. The remainder of this chapter describes how this all works.

2.2 Executable and Translatable UML Application Models

2.2.1 Separation between Application and Architecture

A UML model can be made executable by simply adding code to it. However, this approach views a model as a blueprint to be filled out with more and more elaborate implementation detail and so ties the model to a specific implementation.

Instead, we must allow developers to model the underlying semantics of a subject matter without having to worry about $e.g.$ number of the processors, data-structure organization, or the number of threads. This ability to spec-

ify functionality without specifying implementation is the difference between blueprint-type models and executable-and-translatable models.

How can we capture the functionality of the system without specifying implementation? The trick is to separate the *application* from the *architecture*, and that trick differentiates blueprint-type models from executable ones. Figure 2.1 illustrates this separation. Focus on the dotted line that separates the two parts.

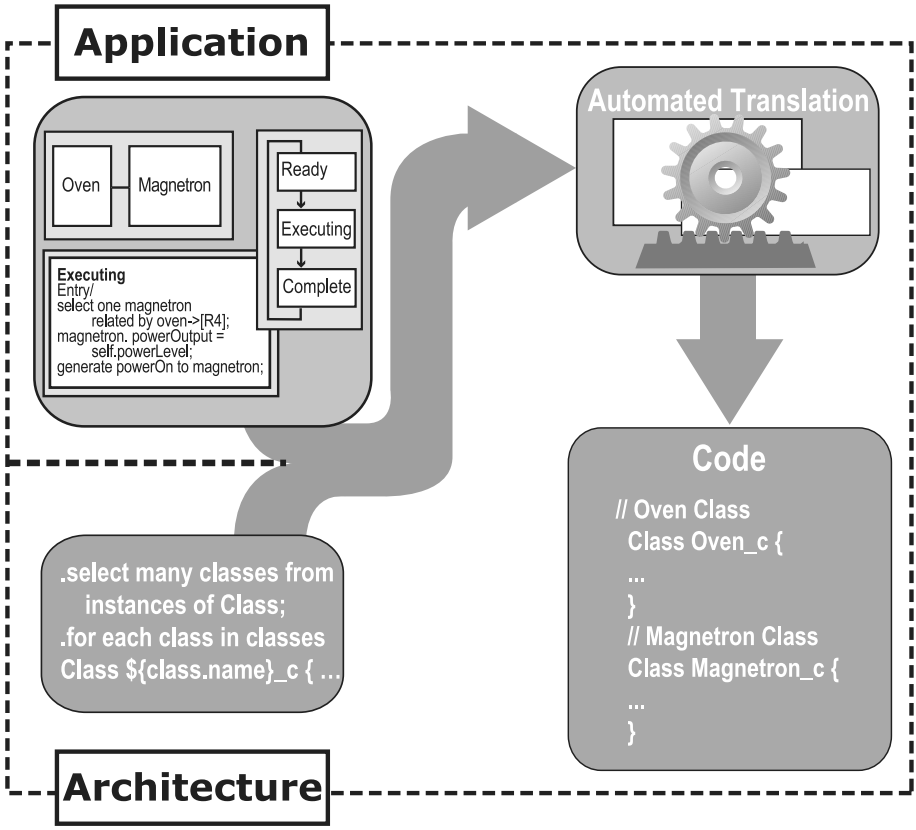

Figure 2.1. The Separation between Application and Architecture

The dotted line between "application" and "architecture" enables the capture of an executable application model by representing it in a simple form: sets of data that are to be manipulated; states the elements of the solution go through; and some functions that execute to access data, synchronize the behavior of the elements, and carry out computation. This dotted line, consisting of sets, states, and functions, captures the *modeling language*.

Only the semantics of the modeling language matter for translation purposes. If a class is represented graphically as a box, or even as text, this is of no consequence. Equally, no semantics content is added by composite structures. They can be convenient for the modeler, but to be executed they must be reduced to their constituent classes. Similar arguments can be made for the rich vocabulary of state machines that also can be reduced to simpler elements.

The UML is just an accessible graphical front-end for those simple elements. When you build a 'class' such as CookingStep in a microwave oven, that represents a set of possible cooking steps you might execute, each with a cooking time and power level. Similarly, when you describe the lifecycle of a cooking step using a state machine, it follows a sequence of states as synchronized by other state machines (when you open the microwave door, it should stop cooking), external signals (such as a stop button), and timers. And in each state, we execute some functions.

A selection of these simple elements makes up the executable UML modeling language. The number of elements is low to ease the construction of translators and more importantly to ease the burden of learning the language and eliminate the ambiguity that accompanies the use of multiple constructs to represent the same concept. The elements are sufficiently primitive to be translatable into multiple implementations, but be powerful enough to be useful. Determining exactly which elements make up an executable UML is therefore a matter of judgment, which implies there are many possible executable UMLs.

Naturally, it's a bit more complicated than that, but the point is that any model can be represented in terms of these primitive concepts. And once that's done, we can manipulate those primitive concepts *completely independently of the application details*.

2.2.2 Actions

The introduction of the Action Semantics enables execution of UML models, but at a higher level of abstraction than code. The difference between an ordinary, boring programming language and a UML action language is analogous to the difference between assembly code and a programming language. They both specify completely the work to be done, but they do so at different levels of language abstraction. Programming languages abstract away details of the hardware platform so you can write what needs to be done without having to worry about $e.g.$ the number of registers on the target machine, the structure of the stack or how parameters are passed to functions. The existence of standards also made programs portable across multiple machines.

To illustrate the issues, consider the following fragment of logic:

```
Sum = 0;
Until Calls = null do
    Sum += Calls.timeOfLastCall;
    Calls = Calls.next;
endUntil
```

The elements in this fragment include assignments, `until`, `null`, an expression (with equality), an increment of a variable, and a linked list of some sort.

The semantics of `While not (expr)`and `Until (expr)` are the same. One of them is syntactic sugar—a sweeter way of expressing the same thing.[1] Of course, which one is syntactic sugar, and which the one true way of expressing the statement is often a matter of heated debate. One can always add syntactic sugar to hide a poverty of primitives *so long as the new syntax can be reduced to the primitives*. The choice of primitives is a matter of judgment and there is no bright line.

Consider, too, a triplet of classes, X, Y, and Z with associations between them. We might wish to traverse from the x instance of X to Y to Z to get the p attribute (x->Y->Z.p). The primitives involved here are a traversal and a data access. However, we could choose to implement these classes by joining them together to remove the time-consuming traversal. In this case, the specification (x->Y->Z.p) can be implemented by a simple data access xyz.p, where xyz is an instance of the combined classes X, Y, and Z. In this example, the traversal is a single logical unit expressed as a set of primitives. Defining the specification language so that these primitives are grouped together in a specification allows the translation process to be more efficient. For example, if `GetTheTimesOfAllCalls` is a query defined as a single unit in the manner of (x->Y->Z.p), so that all knowledge of the data structures is localized in one place, we can implement it however we choose.

We can go further. The logic above simply sums the times of some calls that happen to be stored in a list. We can recast it at a higher level of abstraction as: `GetTheTimesOfAllCalls`, sum them. This is the formulation used by the UML action model. The application model can be translated into a linked list as in the code above, a table, distributed across processors, or a simple array.

This last point is the reason for the word 'translatable.' An executable translatable UML has to be defined so that the primitives can be translated into any implementation. This means isolating all data access logic from the computations that act on the data. Similarly, the computations must be separated from the control structures (in the fragment above, the loop) so that the specification is independent of today's implementation—including whether to implement in hardware or software.

[1] We once saw the following code.

```
Constant Hell_Freezes_Over = False;
Until Hell_Freezes_Over do. . . . Sweet!
```

2.2.3 An Executable and Translatable UML—Static Elements

Executable and Translatable UML (xtUML, or just Executable UML) [131] defines a carefully selected streamlined subset of UML to support the needs of execution- and translation-based development, which is enforced not by convention but by execution: Either an application model compiles and executes correctly, or it doesn't.

The notational subset has an underlying execution. All diagrams ($e.g.$ class diagrams, state machines, activity specifications) are "projections" or "views" of this underlying model. Other UML models that do not support execution, such as use case diagrams, may be used freely to help build the xtUML models, but as they do not have an executable semantics, they are not a part of the xtUML language. The xtUML model is the formal specification of the solution to be built on the chip.

The essential components of xtUML are illustrated in Figure 2.2, which shows a set of classes and objects that use state[2] machines to communicate. Each state machine has a set of actions triggered by the state changes in the state machine. The actions then cause synchronization, data access, and functional computations to be executed.

Each class may be endowed with a state model that describes the behavior of each instance of the class. The state model uses only states (including initial pseudostates and final states), transitions, events, and actions. Each class may also be endowed with a state model that describes the behavior of the collection of instances. Each class in a subtyping hierarchy may have a state model defined for it as a graphical convenience.

A complete set of actions, as provided by UML 1.5 and later, makes UML a computationally-complete specification language with a defined "abstract syntax" for creating objects, sending signals to them, accessing data about instances, and executing general computations. An action language "concrete syntax" [3] provides a notation for expressing these computations.

2.2.4 xtUML Dynamics

Figure 2.2 showed the static structure of xtUML, but a language is meaningful only with a definition of the dynamics. To execute and translate, the language has to have well-defined execution semantics that define how it exe-

[2] UML 2 uses "state machine" to mean both the diagram (previously known as a state chart diagram) and the executing instance that has state. To reduce ambiguity, we have chosen to use "state model" for the diagram describing the behaviors, and "state machine" for the executing instance. Where either meaning could apply, we use "state machine" to be consistent with the UML.

[3] BridgePoint ® provides OAL (Object Action Language) that is compliant with the abstract syntax standard, but there is presently no action language (notation) standard.

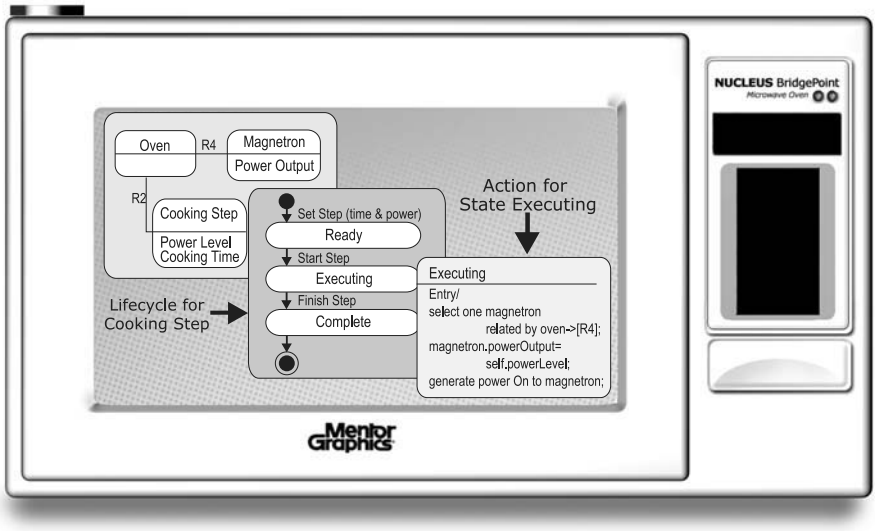

Figure 2.2. The Structure of an xtUML Model

cutes at run time. xtUML has specific unambiguous rules regarding dynamic behavior, stated in terms of a set of communicating state machines, the only active elements in an xtUML program.

Each object and class can have a state model that captures its behavior over time. Every state machine is in exactly one state at a time, and all state machines execute concurrently with respect to one another. Each state machine synchronizes its behavior with another by sending a signal that is interpreted by the receiver's state machine as an event. On receipt of a signal, a state machine fires a transition and executes an activity, a set of actions that must run to completion before that state machine processes the next event.

Each activity comprises a set of actions that execute concurrently unless otherwise constrained by data or control flow, and these actions may access data of other objects. It is the proper task of the modeler to specify the correct sequencing and to ensure object data consistency.

The essential elements, then, are a set of classes and objects with concurrently executing state machines. State machines communicate only by signals. On receipt of a signal, a state machine executes a set of actions that runs to completion before the next signal is processed. (Other state machines can continue merrily along their way. Only *this* state machine's activities must run-to-completion.)

Signal order is preserved between sender and receiver object pairs, so that the actions in the destination state of the receiver execute *after* the action that sent the signal. This captures desired cause and effect.

It is a wholly separate problem to guarantee that signals do not get out of order, links fail, *etc.*, just as it is separate problem to ensure sequential execution of instructions in a parallel machine.

Those are the rules of the language, but what is really going on is that xtUML is a concurrent specification language. Rules about synchronization and object data consistency are simply rules for that language, just as in C we execute one statement after another and data is accessed one statement at a time. We specify in such a concurrent language so that we may *translate it* onto concurrent, distributed platforms; hardware definition languages; as well as fully synchronous, single tasking environments.

At system construction time, the conceptual objects are mapped to threads and processors. The model compiler's job is to maintain the desired sequencing specified in the application models, but it may choose to distribute objects, sequentialize them, even duplicate them redundantly, or split them apart, so long as the defined behavior is preserved.

2.2.5 Application Model Verification

An application model can be executed independent of implementation. No design details or code need be added, so formal test cases can be executed against the application model to verify that requirements have been properly met. Critically, xtUML is structured to allow developers to model the underlying semantics of a problem without having to worry about whether it is to be implemented in hardware or software.

Verification of application models is exactly that: It verifies the behavior of the application models and nothing else. It does not check that 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 carried out by model execution.

When we construct an application model, such as that shown in Figure 2.2, we declare the *types* of entities, but models execute on instances. To test the behavior of the example in Figure 2.2, therefore, we need first to create instances. We can do this with action language or with an instance editor.

We then define operations to create 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 application model can be changed immediately and the test rerun. Care should be exercised to ensure the correct initial conditions still apply. 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.

When each test run is complete, we need to establish whether it passed. This can be achieved either by interactively inspecting attribute values and the state of the application models or by 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.

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

2.3 Manipulating the Application Models

2.3.1 Capturing Application Models

When we build an application model such as that in Figure 2.2, its semantic content must be captured somehow. This is accomplished by building a model of the modeling language itself.

The classes Oven and CookingStep, for example, are both instances of the class Class in the model of the modeling language[4]. Similarly, the states Ready and Executing for the class CookingStep are captured as instances of the class State, and attributes powerOutput and pulseTimer of the class Magnetron are captured as instances of the class Attribute. This is also true for the actions. The action language statement `generate powerOn to magnetron;` is an instance of the metamodel class GenerateSignalAction.

Naturally, a tool will also capture information about the graphical layout of the boxes on the diagrams entered by the developer, but this is not necessary for the translation process. Only the semantics of the application model is necessary for that, and that's exactly what is captured in a metamodel.

2.3.2 Rules and Rule Languages

We have captured the semantics of an application model completely in a neutral formalism allowing us to write rules. One rule might take a 'class' represented as a set CookingStep(cookingTime, powerLevel), and produce a class declaration. Crucially, the rule could just as easily produce a struct for a C program, or a VHDL entity. Similarly, we may define rules that turn states into arrays, lists, switch statements, or follow the State pattern from the Design Patterns community. (This is why we put 'class' in quotation marks. A 'class' in an executable model represents the set of data that can be transformed

[4]The formal name for a model whose instances are types in another model is a *metamodel*.

into anything that captures that data; in a blueprint-type model, a class is an instruction to construct a class in the software.)

These rules let us separate the application from the architecture. The xtUML model captures the problem domain graphically, and represents it in the metamodel. The rules read the application as stored in the metamodel, and turn that into code.

There are many possible rule languages. All that's required is the ability to traverse a model and emit text. As an example, the rule below generates code for private data members of a class by selecting all related attributes and iterating over them. All lines beginning with a period ('.') are commands to the rule language processor, which traverses the metamodel whose instances represent the executable model and performs text substitutions.

```
.Function PrivateDataMember( class Class )
.select many PDMs from instances of Attribute related to Class
.for each PDM in PDMs
${PDM.Type} ${PDM.Name};
.endfor
```

`${PDM.Type}` recovers the type of the attribute, and substitutes it on the output stream. Similarly, the fragment `${PDM.Name}` substitutes the name of the attribute. The space that separates them and the lone ';' is just text, copied without change onto the output stream.

Table 2.1. C++ Code Generation

Rule	**Generated Code**

`.select many stateS related to instances of`
`  class->[R13]StateChart->[R14]State`
`  where (isFinal == False);`
`public:`
`  enum states_e`
`  { NO_STATE = 0,`
`.for each state in stateS`
`  .if ( not last stateS )`
`  ${state.Name},`
`  .else`
`  NUM_STATES = ${state.Name}`
`  .endif;`
`.endfor;`
`  };`
	`public:`
	`  enum states_e`
	`  { NO_STATE = 0,`
	`  Ready,`
	`  Executing,`
	`  NUM_STATES =`
	`  Complete`
	`  };`

In the more complete example in Table 2.1, the rule uses italics for references to instances of the metamodel; underlining to refer to names of classes and at-

tributes in the metamodel; and noticeably different capitalization to distinguish between collections of instances vs. individual ones.

You may wonder what the produced code is for. It is an enumeration of states with a variable NUM_STATES automatically set to be the count for the number of elements in the enumeration. (There is a similar rule that produces an enumeration of signals.) The enumerations are used to declare a two-dimensional array containing the pointers to the activity to be executed. You may not like this code, or you may have a better way to do it. Cool: all you have to do is modify the rule and regenerate. Every single place where this code appears will then be changed. Propagating changes this way enables rapid performance optimization.

While the generated code is less than half the size of the rule, the rule can generate any number of these enumerations, all in the same way, all equally right—or wrong.

We have also used the rule language to generate VHDL in Table 2.2.

Table 2.2. VHDL Code Generation

Rule	**Generated Code**

`.select many stateS related to instances of`
`  class->[R13]StateChart->[R14]State`
`  where (isFinal == False);`
`TYPE t_${class.Name}State IS {`
`.for each state in stateS`
`  .if (not last stateS)`
`  ${state.Name},`
`  .else`
`  ${state.Name}`
`  .end if`
`.end for`
`};`
	`TYPE t_CookingStepState`
	`IS {`
	`  Ready,`
	`  Executing,`
	`  Complete`
	`};`

The rule language can be used in conjunction with the generator to generate code in any language: C, C++, Java, Ada, and, if you know the syntax, Klingon.

2.3.3 Model Compilers and System Construction

So, we can build a platform-independent model of our application, we can execute it to verify that it functions properly, and as we've just seen we can translate it into text of any form. The instrument for organizing the collection of translation rules is a model compiler, and as a result, the overall architecture of the generated system is then encapsulated within the model compiler.

Each model compiler is coupled to the target, but the model compiler is independent of the application models that it translates. This is important because maintaining this separation of application from design allows us to reuse both application model and model compiler as needed. We can translate the same application model for many different targets by using different model compilers, but the models of the application do not change. Similarly we can use the same model compiler to translate any number of application models for a given target without changing the model compiler.

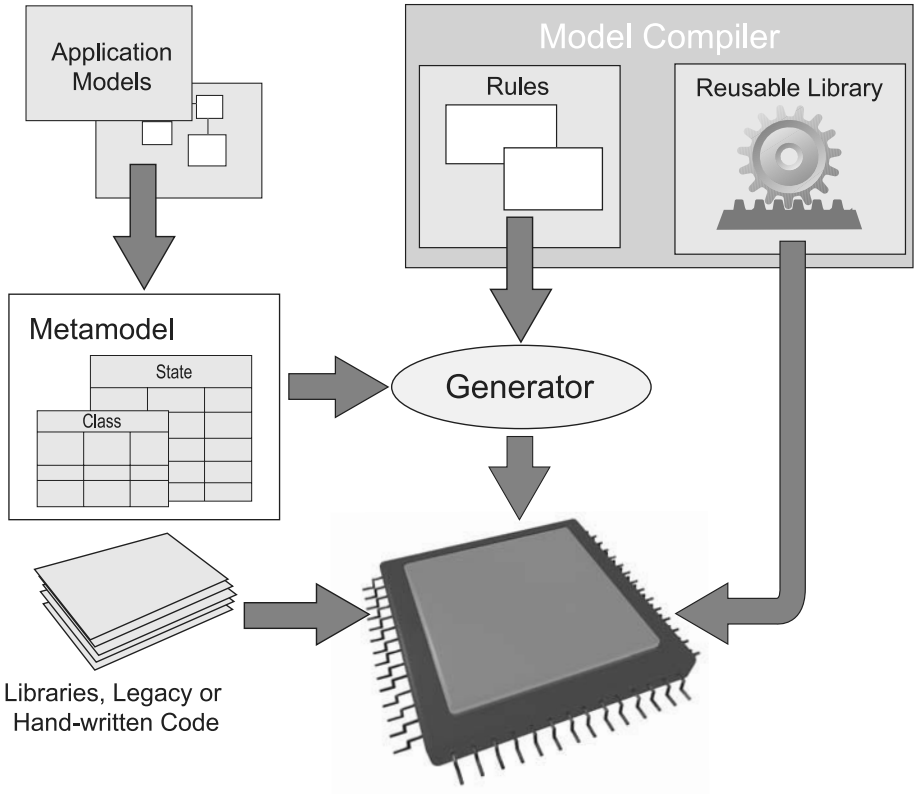

Figure 2.3. Model Compiler and System Construction

Each model compiler encapsulates the architecture of the generated system. If we're generating only software, then a software architecture—the design and implementation approaches used to render each element of the model in some implementation language like C++ or C—has been captured.

For example, a model compiler for an object-oriented architecture will likely translate each UML class to a C++ class or a C struct with each associated UML attribute being translated to a data member of the class or a member of the struct.

The state machines in the application model would be translated into two dimensional arrays where the row index represents the current state of an object, and the received event provides the column index. Each cell contains the value of the next state for the transition identified by the current state (row) and the received event (column). This next-state value is then used as an index into an array of function pointers, each corresponding to a state. This particular approach leads to a constant-time dispatching mechanism for the actions of each state machine.

Of course we can use alternative implementations depending on our needs. For example in some cases, we might choose to use a switch statement or nested if-else statements to implement the state machine, each of which would have slightly different speed and space characteristics (if-else is linear in the product of states and events).

For hardware implementations we might choose to translate each UML class into a collection of registers, one for each attribute in the class. Each state machine of the application model could be mapped to a VHDL case statement. There are, of course, many other possible implementations. UML classes can be mapped into blocks of RAM, and state machines can be translated into a data-driven and gate-conservative dispatcher.

But what about the interfaces between the hardware and software components of the system? These interfaces are just a part of the architecture encapsulated within the model compiler. Let's look at a simple example.

Suppose we have two UML classes, CookingStep and Oven where CookingStep is translated to software and Oven is translated to hardware. In this case the hardware architecture for Oven is a collection of memory-mapped registers. The generated interface for an action in CookingStep that accesses an attribute of Oven is then a simple memory access of the address allocated for the register in question.

Consider a slightly different hardware architecture in which the UML class Oven is mapped to only two addresses, one representing a control register and one for a data register. Accesses to attributes of the class would then be accomplished by writing the attribute identifier and the type of access (read or write) to the control register and then manipulating the data register accordingly (reading it after writing to the control register or writing it before writing to the control register).

The point here is that since the model compiler is generating the implementation for both the hardware and the software components of the system, it has sufficient information to generate the interface between the two, and it will do so without succumbing to the ambiguities of a natural language specification. It will do it correctly, every time.

It's possible to build and deploy model compilers that provide completely open translation systems. Exposing the translation rules in this way allows you

to make changes to the rules to meet your particular needs so that the model compiler generates code exactly the way you want. This puts the translation of the models completely under your control.[5]

OK, we have collections of rules that translate UML models into hardware and software implementations. In many cases we have multiple rules for translating each UML construct, one rule for each variation in our architecture. Now we need a way of selecting which rule to apply to which part of the model.

2.4 Marks

We have described a model compiler that has two parts, but we have not yet described how we tell the model compiler whether to generate hardware or software for a given model element. To do so, we need additional inputs to decide which mapping to perform. These additional inputs are provided as *marks*, which are lightweight, non-intrusive extensions to models that capture information required for model transformation without polluting those models.

Each kind of mark has a name and a type. In addition, a kind of mark can have a default value. In programmer-esque language, we might write

Mark HardSoft [isHardware, isSoftware] = isSoftware

which declares a kind of mark named HardSoft that can have one of two values, where the default is isSoftware.

Most marks apply to instances of metaclasses, so that, for example, if we have a metamodel with class Class, and two instances of that class, Oven and CookingStep, the mark HardSoft can have a separate value for each of those instances, isHardware for the Oven, say, and isSoftware for the CookingStep.

Were the kind of mark to apply instead to generated signals, then the marks would be associated with instances of the class GenerateSignalAction in the metamodel. A given application model element can have several marks associated with it. Each of these marks is an extended attribute of the appropriate metamodel class.

We do not intend to leave the impression that the metamodel should be extended directly. Marks are not part of either the application model or the metamodel, though they can refer to them both. Rather, we view the extended attributes of the metaclasses as being housed on a plastic sheet that can be peeled off at will for a different model compiler. This separation supports both model portability and longevity.

The separation also provides the ability to evaluate a number of different architectural possibilities without requiring modification of the application model. Just change the values of the marks. Not to put too fine a point on it, this solves

[5] We know of at least one vendor providing commercial model compilers in source-code form. More are sure to follow suit in the coming years.

the problem of changing the hardware-software partition after we have verified the behavior of the application model by executing test cases against it.

The plastic sheet analogy suggests that some marks might be related and could all be placed on the same sheet. A single sheet could contain multiple related marks, such as those indicating which types of hardware implementations should be applied to which elements of the application model. Removal of the plastic sheet, then, implies the removal of the entire hardware architecture represented by that sheet from the system.

Marks may also be quantities used to optimize the target implementation. Consider a model that must be transformed into an implementation that occupies as small an amount of memory as possible. We can save memory if we observe that a particular class has only one instance ($e.g.$, Oven). Such a class can be marked as extentLess, and no container need be generated for instances of that class, making references to the instances more efficient. Similarly, we can make trade-offs within the hardware architecture. Suppose the original target had plenty of address space available and consequently mapped each class attribute implemented in hardware to a specific address, making the software access to the attributes simple and efficient. In a subsequent release we move to a lower-cost processor with a constrained address space. Through marks we then instruct the model compiler to use a single pair of addresses for each class to provide access to all the attributes in the class. Since the model compiler knows how to generate the software required for this more interesting approach for accessing hardware-resident attributes, the application models do not change even though the nature of the hardware/software interface has been drastically altered.

There have to be ways for the modeler to assign values to marks. Some implementations provide for graphical drag-and-drop allocation of model elements into folders that correspond with marks; others define an editor for the defined mark sets that can display all marks defined by the model for a selected model element, with pull-down menus for each of the marks. Another option is to define text files, and then use the large set of available editing, and scripting tools.

2.5 Work in Context

The work described here, UML models, metamodels, and transformations to text all fit into a larger context within the Object Management Group. The OMG, the organization that standardized the UML, has a standard approach for storing metamodels, not just the UML metamodel, and it is working now [152] to define a standard approach for transforming populated models to text.

These standards, and others, fit into a larger architecture called Model-Driven Architecture. You may have heard of MDA in an IT context, but the principles

behind it apply to system development in general, and they're not specific to a certain kind of system or even to software [132].

MDA is three things:[6]

- An OMG initiative to develop standards based on the idea that modeling is a better foundation for developing and maintaining systems
- A brand for standards and products that adhere to those standards
- A set of technologies and techniques associated with those standards

At present, MDA is still in development, and some of the technologies need further definition while others need to be standardized.

One key standard that is missing at the time of writing is a standard definition of an executable, translatable UML. While there are mechanisms that allow for the interchange of models between tools, there must be agreement on the UML elements they can each understand. That is, the tools must share a common subset of UML for the tools to communicate effectively. It is possible for one tool to be "more powerful" than another, but effectively that restricts the power of a two-tool tool chain to the weaker of the two. When multiple tools claim to be in the same tool chain, it is only as strong as its weakest link.

Model-driven *architecture*, of course, is the name of the game. Not only must there be standards for the UML and the rest, but it's also important that tools built to these standards also fit together within that architecture and so create a complete model-driven development environment. This set of tools, loosely sequenced, constitutes a tool chain.

There are many possible tools that need to be integrated to make a complete development environment. With the right standards, one can envision tools that do the following:

- Transform one representation of an underlying model to another representation friendlier to a reader
- Generate test vectors against the application model, and then run them
- Check for state-space completeness, decidability, reachability, and the like
- Manage deployment into processors, hardware, software, and firmware
- Mark models

[6]Dr. Richard Mark Soley, the Chairman of the OMG, defined MDA thusly. We also used his definition in [132], for which Dr. Soley was a reviewer.

- Partition or combine behavior application models for visualization or deployment
- Analyze performance against a given deployment
- Examine the generated code in execution (in other words, model debuggers)

We can imagine a developer receiving a application model from a vendor or colleague; turning that application model into a comfortable notation or format; making a change with a model builder; verifying that the behavior is correct by analysis and by running test cases; marking the models and deploying them; analyzing performance; debugging the resulting system, and so forth.

When developers have the ability to provide specification tidbits at varying levels of abstraction and then link them all together, MDA will face additional tool challenges regarding smooth integration between different specification levels, such as consistency checking, constraint propagation, and incremental mapping-function execution.

As the SoC community continues to push toward a higher level of abstraction for the specification of systems it will be important to establish and maintain relationships with organizations focused on model-driven development, even if the traditional focus of such groups has been software development.

2.6 How Does All This Stack Up?

Partition? Because the xtUML models accurately and precisely represent the application, and the implementation is generated, with absolute fidelity from these models, the partition can easily be changed, and a new implementation can then be generated. This replaces weeks of tedious manual changes to an implementation with a few hours of automatic generation.

With the ability to change the partition and regenerate the implementation, the developers can explore much more of the design space, measuring the performance of various allocation arrangements that would otherwise be prohibitively expensive to produce.

Interface? The interface between the hardware and software is defined by the model compiler. Because the implementation is generated, there can be no interface mismatches. Because we no longer have two separate teams of people working from a natural-language interface specification, the generated implementation is guaranteed to have exactly zero interface problems. (This does have the unfortunate side effect of reducing the number of opportunities for logic designers and software engineers to spend long nights together in the integration lab fixing interface problems.)

Marks are a non-intrusive way to specify allocation of system functionality without affecting system behavior. The existence of automated tools to cause

the generation of the system with interfaces known to be correct completely removes any interface problems that can lead to a failure to integrate the system.

Integration? Integration is now the integration of two independently tested pieces: the application model and the model compiler. Each can be understood separately, tested separately, reviewed separately, and built separately. The integration is realized by the generalized part of the model compiler (the generator) that embeds the application into the target platform.

The real integration issue now whether that model compiler meets the performance needs of the system. Should performance be less than adequate, we can change the allocation using the existing model compiler, or change the rules to create a new target architecture that does meet the performance requirements

2.7 A Hole in the Head?

SoC needs UML, but not a lot of it, and even less does SoC need more UML, especially more UML used specifically to capture hardware implementation. That way lies the primrose path to the ever-burning fires.

All SoC needs is a small, but powerful, subset of UML enabling abstract specification of behavior. Automated mappings enable interface definition in one place, so that consistency is guaranteed. Marks enable late decision making on the partition.

That's all we need; we need more UML like a hole in the head.

http://www.springer.com/978-0-387-25744-0

UML for SOC Design
Martin, G.; Müller, W. (Eds.)
2005, XII, 272 p., Hardcover
ISBN: 978-0-387-25744-0