Model-based Development of Embedded Systems: Executable Models vs. Code Generation

Tim Schattkowsky[1]

[1] University of Paderborn, C-LAB,
D 33102 Paderborn, Germany
tim@c-lab.de

Abstract. The use of models during the development of embedded systems is nowadays fairly limited. During the evolution of the system, the implementation and the design models often tend to get out of sync. The use of Model Driven Architecture in the development of embedded systems makes this problem more imminent, as it is much more demanding by introducing two separate modeling levels. Thus, there is a need to introduce techniques that overcome this problem. Complete code generation and the use of executable models potentially eliminate the need for a manual implementation that may get out of sync with the design models. We discuss important properties of such approaches and outline the advantages and possibilities of using a UML virtual machine in embedded systems.

1 Introduction

Nowadays, the role of the UML in the development of embedded systems, if it is used at all, is usually limited to the specification of these systems. The employment of UML in the design of embedded systems has caused severe problems in practice. Objections from developers stem from their experience that the implementation of the UML design models has to take place manually. This often results in software that significantly differs from their design models, rendering these almost useless. There are multiple reasons for this. Some are caused by the methodology (or lack thereof) used for modeling these systems, others are caused by the role of the models in the development process.

The *design* of the embedded system starts at a high level of abstraction. This abstraction allows an easier and sometimes more formal assessment of the problem. However, the introduction of platform-specific properties significantly increases the complexity of the models and often introduces additional elements into these models that break capabilities to easily verify the correctness of these models.

The *implementation* of embedded systems is nowadays done using traditional imperative programming languages like C or C++. Usually, the design models are translated into these languages manually. This translation takes place by the developer attempting to resemble the modeled behavior using the implementation language. However, since the models usually mainly consist of state charts, this translation

tends to be error prone. Furthermore, especially for embedded systems, the implementation step often already breaks consistency between the design models and the actual application, as the design models often do not fit the implementation platform because the designer was simply unaware of platform specifics or left these out the simplify the design or enable certain verification methods.

Even more, during the *evolution* of the application this tends to get worse. Thus, the actual implementation and the models often tend to get out of sync after a short period of development. The role of the design models later in the application life cycle is often limited. Evolution often only takes place on the implementation level without incorporating evolution of the design models. This renders the design models quite useless after some time. In better cases, the design models are updated manually to reflect changes in the implementation. However, this helps only for documentation purposes.

Model Driven Architecture (MDA) [7] appears to be a direct assessment of the problem, as it separates the Platform Independent Model (PIM) that intuitively solves the problem from the Platform Specific Model (PSM) that describes the actual implementation. However, the inclusion of platform characteristics at the modeling level does not solve the problem. Additional efforts are moved from the implementation to the model mapping. This may actually cause more work to be done at first hand.

The outlined *problems are caused to a large amount be the de facto independence between model and implementation causing these to get out of sync*. There are two well-known approaches that to overcome this problem – code generation and executable models. In this paper, we will discuss their properties with respect to application in embedded systems and their use in MDA-based approaches.

2 Code Generation from Models

Code generation is a well-known way of deriving an implementation from models. A machine computes executable program code directly from the models. This may yield different qualities of generated code. The code generation often only produces *code skeletons*. These skeletons are incomplete fragments of code intended to be by a developer completed in the language of the generated code.

However, models providing complete information on a system can be used to generate the fully operational system from the models. In this case, the generated code can be compiled to the executable system without the touch of a developer. In this case, the code may not even be intended to be modified be the developer, as this would raise the problem of reintegrating the changes into the model.

Only systems derived by complete code generation are inherently in sync with their models. As the models are expected to be equivalent to the code, it is not desirable to allow changes to the generated code. The same changes could be made to the models as well. Thus, only changes on the model should be allowed.

There is considerable support for code generation from UML models available. Tools like Rational XDE [10], MagicDraw [4] and Together [1] are capable of generating code skeletons from UML models to some extend. However, no fully model-based development is currently supported, especially not at MDA level.

3 Directly Executable Models

The most obvious way of keeping the implementation and the design synchronous is quite straightforward using the design models as the implementation. For UML, this implies the existence of a Virtual Machine (VM) capable of executing the UML subset used in the modeling approach.

The use of a virtual machine eliminates the need to translate the models into a different language use a code generator. This has several advantages. The model becomes directly executable on any platform providing the necessary VM. This eliminates the effort necessary to generate new executable code each time a new target platform appears or some improved mechanisms for running the software on a platform have to be incorporated. Such updates will be made to the VM without touching the executed models. Thus, unlike for code generation approaches there is no dependency on the creator of the model to run it on a new or enhanced platform. This is a substantial gain, as all changes to the execution environment (the VM) are now immediately beneficial to all executed software. Furthermore, the models are always immediately executable. This decreases the turnaround time during development. However, [12] seems to overestimate the value of this benefit, as it can be automated to a large extend. This may practically nullify the benefit.

4 Comparing Resource Requirements

There are also some problems with the use of executable models. A VM is a more generic approach than generated code and uses potentially more resources. It is likely to be more time and memory consuming the specialized generated code. In the field of embedded systems this is an important consideration. There is a strong belief, that these systems are resource limited in a way that we need to consider different techniques for development on these systems. While this was true when the systems where limited in a sense, that both computing power and memory where close to nonexistent, it is worth reconsidering this nowadays. We are approaching an age where especially the memory required holding the executable part of a software is of an ever decreasing size compared to the memory available. Even if in some systems memory is still a consideration, it will fade over time.

What is most interesting in this context is the size and the memory overhead of the VM, because it is very likely that the VM is more complex and has a larger memory imprint then the software executed on such a VM. Otherwise the difference in memory usage between using a VM and executing generated Code comes down to the relation in size between the models and the generated code. The size of the VM is however strongly related to the size of their counterpart for generated code, which is the runtime library the code relies on. These are similar in function and very likely require comparable resources. Thus, the efficient realization of a VM does not need to have drawbacks on this side.

What remains is only the difference in size for the executed code. It seems to be reasonable to consider this difference as fairly constant. The usual way of code gen-

eration implies that the size of generated code strongly correlates with the model size. This makes the size difference undesirable for most applications, especially in the near future.

5 Approaches on Executable UML

There has been already a lot of work done in defining the semantics of UML models for simulation and code generation. Most this work is currently based on the UML 1.x specification. Thus, these approaches had to introduce additional means of describing the behavior that extended beyond UML itself.

The xUML approach [11] targets at creating executable application models and includes a complete development methodology targeting at embedded system development. xUML is currently based on the UML 1.x specification. It creates a well-defined platform for executable models based on class diagrams and state diagrams. The core of the approach is the Action Specification Language (ASL) used to define the behavior of active objects during their lifecycle. With this language, clear action semantics are introduced into the models. This work has been integrated into an OMG-adopted standard for precise action semantics for the UML [6]. However, to create the executable Model, the xUML approach relies on a platform-specific (i.e., Ada) code generation mechanism. Different specific compilers can be used to create the complete software system from the UML models and ASL.

The Executable and Translatable UML ($^{X}_{T}$UML) [8] approach defines a development process that incorporates complete code generation from an application specific UML model. It relies on translation modules that have to generate specific and 100% accurate code for the application target platform. The underlying models are executable and can undergo certain verifications. The application of $^{X}_{T}$UML has been outlined [9][5].

For virtual machines, the Sun Java Platform [14] is the most well-known example of a virtual machine nowadays. It is designed to support, among others, some features that are of interest here. It executes “bytecode” representing object-oriented programs. The Java VM [3] has been implemented in hardware, und thus it demonstrates the applicability of the idea of executing fully object-oriented in a VM that can be implemented in both hard- and software.

VMs for other programming languages have been created, including the once groundbreaking UCSD Pascal and Smalltalk [2]. Software emulators for computer hardware emulating complete systems in software or the runtime for the classic Lucasarts Adventures [13] are examples for VMs as well.

Finally, there exists a proposal for a UML VM [12]. However, this does not cover the use of a true VM for UML models. Instead, it uses code generation to create Java code from the models. It appears that the runtime semantics including garbage-collected memory management are fully supplied by Java. This is currently not desirable for embedded systems.

6 Towards a UML Virtual Machine for use in Embedded Systems

As already described, we several advantages for the use of a VM over code generation. Some of these advantages have to be backed up by the design of the VM. The special properties of Embedded Systems have to be considered when considering the application of a VM in embedded systems. As said, the VM itself has to fit inside the system to meet the more limited memory requirements. Furthermore, a mechanism for providing timing-accurate execution has to be considered. The necessary timing information has to be contained in the executed model (i.e., as a description of maximum transition time between states),

By providing a VM for UML with a small memory footprint and no need for a stack to execute the VM itself (as opposed to using a stack at runtime for the executed model) it is possible to synthesize such a VM to a FPGA to directly provide a flexible execution environment. The JAVA VM has been hindered by both their need for a stack and the garbage collection memory management to achieve this nowadays.

A UML VM targeting at embedded systems could be designed in a way that overcomes these problems. Such a VM could be based on executing a mixture of state charts and sequential code in a way that enables the complete development of the whole system using MDA by using VM-compatible models as PSMs. The PSM for embedded systems is in this special case very likely to be much related to the PIM. As the development would be focused on the construction of these models, the approach inherently enables fully model-based development.

7 Conclusion & Future Work

We have discussed different aspects of using code generation and executable models to support MDA. We found the use of these methods very desirable, as both can fully eliminate the need for manual implementation in the development of embedded systems. However, we found that there are some important advantages in the use of a VM, which make it more desirable than code generation approaches. The use of a VM keeps models and implementation inherently in sync. Furthermore, improvements to the runtime environment are instantly available to all application compatible with the VM. Even more, these applications are instantly available for new platforms hosting the VM.

We are working on the implementation of such a virtual machine for use in embedded systems. The VM under development complies with the requirements outlined here and is based on the current UML 2.0 specification. We plan an evaluation of the VM in a european industry context.

References

[1] Borland Software Corp.: Together. http://www.borland.com/together, 2003.

[2] Goldberg, A., Robson, D: Smalltalk: The Language and Its Implementation. Addison-Wesley, 1983.

[3] Lindholm, T., Yellin, F.: The JavaTM Virtual Machine Specification. Second Edition, Addison-Wesley, 1999.

[4] Magic Draw. http://www.magicdraw.com, 2003.

[5] Mellor, S., Balcer, M., Balcer, M.J., Mellor, S.J.: Executable UML: A Foundation for Model Driven Architecture. Addison-Wesley, 2002.

[6] Object Management Group, The: Action Semantics for the UML. OMG ad/2000-08-04, 2000.

[7] Object Management Group, The: Model Driven Architecture (MDA). OMG ormsc/2001-07-01, 2001.

[8] Project Technology, Inc.: Executable and Translatable UML - Summary. http://www.projtech.com/pdfs/xtuml/xtuml_summary.pdf, 2002.

[9] Project Technology, Inc.: Model Driven Architecture (MDA). http://www.projtech.com/info/mda.html, 2003.

[10] Rational Software: Rational XDE. http://www.rational.com/products/xde/, 2003.

[11] Raistrick, C., Wilkie, I., Carter, C.: Executable UML (xUML). In Proc. 3rd International Conference on the Unified Modeling Language UML, 2000.

[12] Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The Architecture of a UML Virtual Machine. In Proceedings OOPSLA 2001, ACM Press, 2001.

[13] ScummVM. http://www.scummvm.org/, 2003.

[14] Sun Microsystems, Inc.: The Java Language Specification. Second Edition. 2000.