Model-Based Specification and Execution of Embedded Real-Time Systems

Tim Schattkowsky, Wolfgang Mueller
Paderborn University, Paderborn, Germany

1. Introduction

Embedded systems design comes in different variations and is most often due to the specific application or project [1]. Sometimes, graphical means like StateCharts or Matlab/Simulink are applied for graphical specification. Code generators target for different micro controllers such as C166 and different Real-Time Operating Systems (RTOSs) like OSEK. There have been efforts to investigate retargetable compilers to easily adopt them to different hardware platforms.

Most recently, the MDA approach (Model-Driven Architecture) became also recognized in the domain embedded systems design. MDA is based on the idea that specification and development is based on a platform-independent model (PIM). The PIM has to be mapped to a platform-specific model (PSM) in some way. This PSM is representing the actual implementation.

Executable UML seems to become a major role here, as it enables the PSM itself to be executable. However, current approaches target towards platform-specific code generation. Examples are ${}^{X}{}_{T}$UML [2] and xUML [4]. The latter is based on the Action Specification Language (ASL), which defines the computational model of active objects for code generation.

A different approach to portable code is the idea of a *Virtual Machine (VM)*. A VM is a virtual computer defining the runtime environment for the executed software. It functions as a low-level abstraction layer defined by its behavior and the format of the executed software program. This program consists often of *bytecode* similar to the machine code executed by a microprocessor. One such example is the Java VM. Compiled Java bytecode programs can run on any Java VM. The use of a VM eliminates the need to translate the models to different platforms by code generators. Instead, only the VM itself has to be ported to each platform. Available software should instantly run on any new VM implementation. Improvements to the runtime environment are immediately beneficial to existing already delivered software. This reduces costs for application development and testing.

We propose a methodology for an executable UML 2.0 subset based on State Transition Diagrams (STDs) and Sequence Diagrams (SD) that covers interrupts, exceptions, and timeouts. We have defined a UML Virtual Machine (UVM) as the run-time environment for complete executable specifications based on that executable UML subset. Such specifications are compiled to binary programs consisting of data structures (STDs) and bytecode (SDs). These binary programes are executed directly by the UVM.

2. Methodology

We employ UML Class Diagrams, STDs and SDs to define the complete executable application. The remainder of this paper focuses on SDs and STDs, as the use of class diagrams for type definition is quite straightforward.

UML has introduced STDs as a variation of Harel's StateCharts. However, several behavioral aspects like concurrent objects and simple data-oriented algorithms (e.g., sorting) cannot be easily expressed. The UML 2.0 sequence diagrams (SDs) have several extensions for loops etc, which make them an ideal complement for the use of STDs. Thus, we can overcome these drawbacks of just using STDs through the integration of SDs with STDs by using SDs as the Action[1] language to describe the Activities associated with States or Transitions.

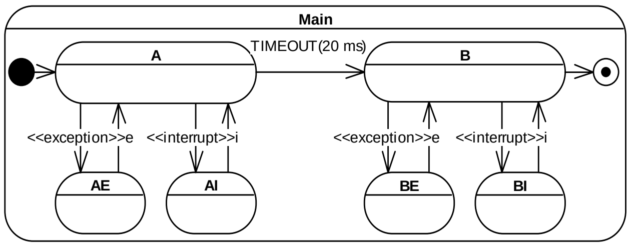

Figure 1: Top Level STD with two Primary States

Our approach starts from a state-oriented specification of the operation behavior by a STD. After clearly identifying the primary States, exceptions, and interrupts, which are due to individual application and available sensors, the primary top level States are defined and connected to the primary initial and final State. For

[1] Note that references to classes from the UML metamodel are capitalized.

timeouts, we use a relative TimeTrigger with an Integer value and a physical time unit as parameter. Interrupts and exceptions are introduced as stereotypes of Trigger. An example is given in Figure 1, which shows a top level STD with two primary States A and B and corresponding Transition for exceptions and interrupts. When the given timeout is reached, an exception is thrown and the Transition performed. The primary States are composite States, which may recursively embed STDs or SDs. The combination of diagrams used is mainly due to the individual application.

3. Executing UML

The binary programs compiled from the UML models are executed by the UML VM (UVM), which can be implemented in hard- or software. This section outlines the structure and execution of such programs.

2.1 UML Program Code

In order to enable simple and efficient execution of these models, we use a transformation of the specification to an equivalent executable model based on binary encoding for the STDs and bytecode for the SDs.

The encoding of a STD results in a data structure that can be interpreted by the UVM to execute transitions. It is not necessary to include direct support for all STD features. Instead, the STDs are transformed into semantically equal simpler STDs where entry- exit- and transition Activities are mapped to additional States and Transitions. This is also used to resolve deferred Triggers.

Each State may have a timeout value. If this timeout value is less then the pending timeout (from a containing State), it is pushed on a timeout stack in the UVM, and becomes the pending timeout. A timer in the UVM triggers the corresponding transition if the State is not exited before the timeout. The timeout will be removed from the timeout stack when exiting the State.

Concurrent States are not supported in STDs. Thus, there is no need for an event queue at runtime. Conflicting transitions are not possible. This significantly reduces the complexity of the UVM implementation.

The encoding of the SD results in bytecode, which is equivalent to statements of the SD. However, since the sequence diagram may contain severally nested expressions and invocations. Those have to be resolved and mapped to the simpler instruction set used by the UVM. The code of the nested diagrams is transformed into a flattened version consisting of instructions executable by the UVM. Parameters are passed to and from Interactions using the stack like in other compiled high-level languages. The data structures used to hold class and instance information are similar to those in C++.

The instruction set supported by the virtual machine contains instructions similar to those of common microprocessors (cf. Table 1). However, the UVM uses no registers, as this is an unnecessary limitation. In contrast, the UVM is stack-oriented and uses no registers. Hardware implementations may make transparent use of registers. This allows flexible use of the available hardware resources and enables simple scalable implementation in both hard- and software.

Table 1: Excerpt of the UVM Instruction Set

Mnemonic	**Dst**	**Src**	**Semantics**
Traditional Instructions			
`mov.[b,l,d]`	`mem`	`mem/im`	`Dst ← Src`
`push.[b,l,d]`	`mem`		`[-sp] ← Dst`
`add.[b,l,d]`	`mem`	`mem/im`	`Dst ← Dst + Src`
`cmp.[b,l,d]`	`mem`	`mem/im`	`[Flags according]`
OO Control Instructions			
`new`	`mem`	`mem/im`	`Dst ← (instanceof Src)`
`destroy`	`mem`		`(destroy instance Dst)`
`invoke`	`mem`		`(invoke operation Dst)`
`invokea`	`mem`		`(invoke Dst asynch.)`
StateMachine Control Instructions			
`trans`	`mem`		`(transition to Dst)`
`event`	`mem`		`(handle event Dst imm.)`
`complete`			`(completion of State)`

In addition to the usual instructions for moving data, integer and floating point math, bit manipulations and flow control, instructions for managing and invoking objects have to be introduced. Furthermore, instructions for triggering Transitions have been added.

While the invocation of a class operation can be implemented directly using the concept of a C++ vtable, for instance, the support for object creation and destruction in the instruction set has wide implications, as the UVM must provide special memory management functions when those instructions are executed.

2.2 Runtime Environment

UML programs are executed starting from the primary initial State until either the sub-StateMachine reaches a final State or an unexpected exception or external event (i.e., interrupt) occurs. In such a case, the UVM stops the execution of the UML program, processes the events or exception and handles initiates scheduling if necessary.

The UVM has to provide certain high-level runtime functionality including memory management and scheduling. This functionality is provided as a predefined set of Operations of a static Runtime Class. This class may also contain initialization code.

References

[1] Marwedel, P.: Embedded Systems Design. Kluwer, 2003.

[2] Project Technology, www.projtech.com, 2003.

[3] The Object Management Group: Unified Modeling Language: Superstructure. OMG ad/2003-04-01, 2003.

[4] Raistrick, C., Wilkie, I., Carter, C.: Executable UML (xUML). In Proc. UML, 2000