Gaining Insight into Executable Models during Run-Time – Architecture and Mappings

Philipp Graf and Klaus D. Müller-Glaser
University of Karlsruhe
Laboratory for Information Processing Technology (ITIV)
Engesserstr. 5, 76131 Karlsruhe, Germany
{graf, mueller-glaser}@itiv.uni-karlsruhe.de

Abstract

Model based development based on different domain specific tools and graphical notations gains increasing importance in system design of embedded electronic systems allowing fast concept-oriented prototyping from model to code. This paper describes an extension to our seamless model based development approach: An architecture for debugging models that are executed on target systems or in dedicated rapid-prototyping environments. We discuss the advantages of such an approach as opposed to simulation, describe our universal architecture and present our implementation prototype. We focus on the definition of MOF-based runtime models and their synchronization with the runtime target state. An example of debugging state-charts shows the feasibility of the approach.

1 Introduction

When developing embedded systems cost, time-to-market and quality are getting in conflict with ever increasing complexity of the design. This complexity results from growing functionality, a design space getting larger, distribution of functionality to different distributed control units, optimization of power dissipation and performance requirements. As of today up to 80 electronic control units (ECU) are built into a upper-class car, communicating over various different communication networks. An Airbus A380 contains over 1000 field programmable gate arrays (FPGA).

With increasing complexity one can observe a shift from electronic and pure control based systems toward software-based systems. The object-oriented paradigm is becoming more and more accepted in the domain of embedded software development and is supported by a growing number of graphical tools for computer aided systems and software engineering (CASE). Relevant notations include the Unified Modeling Language (UML) [11], Statecharts [7] and Signalflow-based modeling [13]. Graphical approaches usually feature a higher level of abstraction, also by putting the platform dependent non-business logic into transformations like Object Management Groups’s (OMG) Model Driven Architecture suggests [10]. We see such approaches as logical next steps after the progress from low-level assembly language programs over high level languages to the object oriented paradigm for specifying embedded software.

In addition to pure simulation approaches only few tools exist that leverage the process of finding errors in embedded software from the level of source-code to the level of graphical modeling. It is very likely that while transforming a model to code its structure changes dramatically. A simple Stateflow diagram consisting of few states generates into several hundred lines of C-code. Also, the code was not written by the developer himself but generated, making a source-code level view seem inappropriate while debugging. Insight into processing of the model on a real target system or in the context of a processor-in-the-loop (PiL) or hardware-in-the-loop (HiL) simulation is not possible. If the system was heterogeneously developed using various specialized development tools, finding errors in the complete system is only possible on the source-code level.

Thus, the challenge we face lies in constructing a system that leverages the debugging of executable models to model-level and allows to debug executable models across notation boundaries. Section 2 briefly presents our view on integrated model-based development and introduces the already existing tool-chain. Section 3 focuses on the question, why debugging has advantages to pure simulation approaches. In Section 4 we introduce the concept of runtime information models needed for debugging and their synchronization, for which we present an example in Section 5. These models lead to a flexible debug architecture presented in Section 6. In Section 7 we present our imple-
mentation before closing with conclusions and an outlook on future work.

2 Model based development of software for embedded systems

The following section gives a brief overview on our development process for specifying software targeted on embedded systems based on different modeling domains and transforming it to an executable binary that is appropriate for rapid prototyping. A longer and more complete description of the approach and tool support is given in the original conference paper in [6].

In our eyes, defining a complete and potentially complex design process for our approach would be too rigid. Nevertheless, the presented approach puts constraints on the development process, which can be incorporated in different process models.

Figure 1. Top-down design flow from requirements to executable

Figure 1 presents a basic top-down design flow. The described development process is realized and supported by the software tool Aquintos.GS (formerly GeneralStore), a research spin-off of our department now available commercially [1]. It offers versioned model- and user-management, thus allowing concurrent engineering of large-scale heterogeneous models.

The design process starts from a textual specification document, which is the basis for the following requirements specification phase. Starting from there a preliminary and roughly outlined class model can be extracted. Subsequently the model of the embedded system is divided into different domain sub-models, where each sub-model is a software component, a subsystem in the control-domain, or a state chart in the time-discrete domain.

Now the developers can use their CASE-tool of choice in a notation familiar to their area of expertise. The already mentioned integration platform Aquintos.GS applies automated transformations on these notations to convert them to a top-level UML metamodel in a bidirectional way. With respect to the runtime mapping concept introduced in Section 4 it is important to emphasize that we provide a complete structural transformation from and to the UML. Thus, for example state diagrams modeled using MATLAB Stateflow [13] are stored in our data-base based on the UML state diagram meta-model.

To allow communication between the various model parts they have to be linked. One possible yet inflexible, error-prone and badly maintainable approach is to link the system parts on the code level. Instead, interfaces between the modeling domains can be specified on model level in the overall UML representation of the system and are transformed prior to code generation to a generatable class model.

The Aquintos.GS platform allows for partitioning of the whole system into subsystems. Thus, we enable the usage of different domain specific code generators. Each code generator has benefits in specialized application fields. For control-systems there are commercial code generators like Embedded Coder [13] or ECCO [4]. In the software domain, we provide a code generator as a plug-in to enable structural and behavioral code generation directly from a UML model.

The generated source code is transformed to an executable binary during a last step. The obtained executable file can then be deployed to the execution target allowing validation of the model.

3 Debugging and Simulation

Both debugging and simulation focus on the same task during development: searching and removing defects that exist in a model and that lead to errors or even failure of the whole system. They do so in four phases: detecting, finding, analyzing and finally eliminating defects. While detecting errors or failures can be accomplished using testing, finding and analyzing defects requires tool assistance that in principle can be given by debugging and simulation approaches alike.

While debugging requires code generation and execution on the target system or a rapid prototyping device, simulation approaches work by interpreting the model. However, this interpretation can include code generation, for example
to compile accelerated simulation kernels [2].

For simulation the target system is not required. Using a debugger implies that the model is executed on a target system or a rapid prototyping device that is or at least resembles the target unit.

Designing embedded software using models imposes additional constraints on a simulation approach. Generally embedded software exists in the context of peripherals and the surrounding environment. For simulation this means that actuators and sensors need to be implemented and also simulation of the environment needs to be provided. The equivalence of the simulated peripherals to the real system is however problematic. In fact the need to also model and implement these adds complexity and additional sources of defects to the design.

A second source of the unclear semantic equivalence of simulator and real system is added when following a model based approach and transforming this model to code automatically. In case of simulation the execution semantics are provided by the simulator that might not exactly be the same than the execution semantics that are implicitly assigned to the model during the transformation phase to source code. When specifying state charts, execution semantics can vary between tools, simulators and different code generators.

Simulation offers pure functional validation. When simulating the execution of a model we know of no ways to validate non-functional requirements, especially concerning performance characteristics. Testing on real-time constraints or detecting and resolving race-conditions requires execution on a platform as close to the target platform as possible.

For a heterogeneous development environment as we focus on in our work, simulators can be coupled. The visualization is usually accomplished using dedicated simulation tools or a visualization inside the design tools. For debugging we couple the model before code generation as described in Section 2. The execution is then visualized using either the development tools or a dedicated visualization layer. We decide for the latter approach, mainly because the back-annotation facilities of existing CASE-tools are, if existing at all, rather rudimentary, proprietary and undocumented. A debugging tool exists for models developed with i-Logix Rhapsody [8].

However, some of the mentioned advantages for debugging can also be regarded as advantages for a simulation approach. This particularly holds true for early design phases where hardware and peripherals are not existing yet and pure functional validation is sufficient. Also reproducibility can be better in simulation.

4 Runtime models and synchronisation

Debugging of a system that was developed model-based can be understood as reversion of the various transformation steps starting from the model down to executable code.

**Figure 2. Mappings between model layers**

Figure 2 illustrates the situation: The result of software design is a model, which is detailed enough to be generated to source code formulated in a high-level language (e.g. C/C++ in embedded software). The source code is transformed again using a compiler resulting in an executable binary ready to be deployed to the target platform. Following the Model Driven Architecture approach of the OMG, artifacts on all three layers can be regarded as models.

The binary file (e.g. the model on the lowest level) is executed on the target platform. Basically, the task of a debugger is to extract runtime information out of the executed binary from the target platform and transport this information back through the abstraction layers to model level.

Fortunately, the first step, the extraction from runtime level to source code level is already accomplished by modern source-level debuggers. Examples include the GNU Debugger (GDB) [5], a member of the GNU Compiler toolchain, the JAVA Debug Interface [12] or proprietary source-level interfaces to debuggers for embedded processors.

The extracted information refers to debug artifacts on source level such as lines of code, variables and other source-level symbols. As we want to obtain model level information (for example the currently active state of an executed state diagram), we need to provide a mapping to model level of the runtime information.

However, the described reversion must not be confused with reverse-engineering where the aim is to reconstruct the
modeling-artifacts themselves. From the viewpoint of the executed model artifacts can be static (elements originating from development) or dynamic (runtime information).

To hold and link the artifacts, the UML-meta-model which is based on the Meta Object Facility (MOF) [9], is extended with a runtime metamodel. Figure 3 shows the relation of this extension to OMG’s meta-layer concept. Each layer is an abstraction of the underlying layer with the top layer (M3) at the highest abstraction level. The bottom layer (M0) comprises the information that we wish to describe. On the model layer (M1) there is the meta-data of the M0 layer, the so-called model. Object-oriented software is typically described on the M1 layer as a UML model. The runtime information is also stored on this layer and linked with the existing modeling artifacts. The metamodel on the M2 layer consists of descriptions that define the structure and semantics of meta-data (e.g., the structure of a valid UML model). These are the metamodels, e.g., UML 1.4, or UML 1.5, and define the language and notation for describing different kinds of data respectively (M1). The metamodel on this level is extended to allow storage of runtime information. The extension allows to dynamically gather debug data needed for debugging and link them with elements of the static model-part on layer M1. Finally, on the M3 layer there is the self describing meta-metamodel MOF. It is used to describe metamodels and define their structure, syntax, and semantic. It is an object-oriented language for defining meta-data.

The obtained runtime information on model level can be used to visualize the internal state of the executable. The visualization can also use elements of the static system part. As an example, for visualizing the runtime state of a state chart the static information is used to render the diagram itself. The dynamic model part is only used to annotate or assign a different color to the active states in the diagram.

5 Mapping example: Executable Stateflow diagrams

To clarify the work a typical mapper accomplishes, the time-discrete execution of Stateflow diagrams and the mapping involved is described in this section. MATLAB Stateflow is a CASE-tool that allows to describe event-discrete models in syntax and semantics close to UML State diagrams and is frequently used by control engineers. Using the extension Embedded Coder/Realtime Workshop optimized C-code can be generated. The source-code is integrated into the framework code generated from a UML-based super-model using the mentioned coupling approach.

When considering an executable Stateflow diagram at runtime and the artifacts that are of interest from the users side during debugging, the following questions can be identified:

- Active State: Which basic states in the diagram are currently active?
- Triggered Events: Which events where triggered and will be relevant for the coming time-step?
- Data values: Which are the values of the data context in Stateflow?
- Transitions: Which transition(s) were triggered during the past time step?

Using this data an application developer can gain insight into state and execution properties of the model. Additionally run-time control has to be transformed from stepping through lines of code to stepping ahead one or several time-steps in the execution of Stateflow diagrams.

Our model-level debugger implements the acquisition of the mentioned model artifacts through mappers and model-level run-control of the target using a controller. The controller is based on setting source-level breakpoints on strategical positions. Resuming the execution on source-level makes the target stop whenever the next model-level time-step is completed.

To allow mapping of run-time data for later display to the user, the UML meta-model holding static model artifacts is extended with meta-classes that allow storage of the data that is acquired by the mapper. Figure 4 shows a detail of this extension for Stateflow diagrams. While modeling and code-generation is accomplished through Stateflow, the debugger works on the representation of the model that was transformed to the UML metamodel. Figure 4 shows the UML metamodel for State diagrams and a simple extension for data values (called Guards in UML). Basis for the runtime model is the DebugContext to which any number of GuardValues can be attached. Each GuardValue is linked to the static Guard it stores the value for.
The described data structures are filled with values by mappers. As an example we describe the mapping of the currently active basic state from the run-time information available from the source-level debugger.

Figure 5 shows a simple Stateflow diagram and the corresponding data structures in C-code when rendering it using Embedded Coder/Realtime Workshop. Every composite XOR-state is rendered into one variable (Figure 5 middle) holding information on the activity of its direct substates in the hierarchy. These variables can be reconstructed purely from model information, namely the state name and a numbering depending on the position in the hierarchy that follows a deterministic rule. The obtained numeric values have to be mapped to model information again. Figure 5 (bottom) shows the mapping. Every level has its own alphabetically sorted enumeration.

Using these connections the debugger performs the mapping using the following algorithm:

1. Begin with top-level composite state, e.g. the diagram.
2. Reconstruct variable name that holds activity information for this level as described before.
3. Acquire value from target via the driver layer.
4. Reconstruct the active substate from the value as described above.
5. Check type of is sub-state. If the substate is a basic state add it to runtime model. If the substate is a exclusive-or-state, recurse into it continuing with step 2. If the substate is a concurrent-state, iterate over all its children recursing into every child with step 2.

By applying this algorithm we obtain a list of all concurrently active basic states in the Stateflow diagram. This information is used for visualization later on.

6 Architecture for debugging of heterogeneous systems

Figure 6 shows the architecture that is the result of the approach introduced in Section 4. It is built up from layers that continuously abstract further from target architecture, source-level debugger and programming language toward visualization on model level.

Starting point is the modeled software that is executed on the embedded target platform. Different target architectures include or support different source level debuggers, that allow extraction of run-time information over proprietary interfaces. Examples are the Java Debug Interface (JDI) that maps into the classpath of the debuggee, the text based interface of GDB or proprietary access on debug hardware of embedded processors.

The driver layer unifies these source level debuggers as far as possible. As the debuggers do not offer the same set of features and information, interfaces were defined that group sets of abilities. Drivers generally implement only a
Figure 5. Stateflow diagram (top) and data structures generated by Realtime Workshop: state information (middle) and mapping (bottom)

Figure 6. Layered debugging architecture

set of the interfaces, influencing the abilities the debug system is able to offer to the higher levels. Common interfaces include run-control (starting, stopping, resetting, stepping), access to symbol data (memory addresses of methods) and expression evaluation (reading variable values). Hardware-based debuggers often additionally offer an interface for acquiring and reading real-time trace data from the target.

Viewed from top, the architecture offers different viewpoints to the user of the debugger. A viewpoint defines a specific view on the system and mediates between target system and the visualization layer that is atop of it. A typical debug session for heterogeneously modeled software uses several viewpoints. The object-configuration of the object-oriented software-intensive part is visualized using an UML object-diagram and every embedded Statechart shows the behavior of components, classes or the state of port protocols. The complete set of viewpoints allows insight in all aspects of the running model.

The visualization layer is based on the open-source development platform Eclipse and the associated diagramming framework GEF [3]. The diagram model is based on the Diagram Interchange (DI) metamodel which is part of the UML specification since the UML 2.0.

Between viewpoint and driver mapper and controller are found that bridge the model layers as described in Section 4. The viewpoint uses the controller to control the system from the viewpoint of the model. An execution step in a state chart corresponds to a complex sequence of commands on the source code level. The mapper updates the runtime information model, that builds the basis for the visualization with information from the source-level debug drivers. An example for mapping Stateflow charts was outlined in Section 5.

7 Prototype

Figure 7 shows a screen capture of our current implementation of the Model Debugger. The tool is extending the bare skeleton of the Eclipse platform by various plugins that allow handling of models and visual debugging.

The prototype allows to open models from XMI, the format for exchanging UML models, or MATLAB Simulink files using the standard Eclipse Navigator view. These files can be visualized using a view showing a model tree of all model artifacts and a tabular detail view of properties and relations (lower right part in Figure 7). DI-based diagrams contained in these models can be opened in a graphical viewer.

Debugging is centered around the DebugSession view in the upper left corner of Figure 7. After connecting to a debug target through a driver the user can add viewpoints with each viewpoint opening a view or a graphical diagram editor as its visualization component. In the figure there exists one viewpoint showing a Stateflow chart. This diagram is already augmented with information on the currently active states in the running target.

The DebugSession view’s tool bar allows the user to run, reset and step the target based on the viewpoint that is currently selected in the list. In the lower left part the DebugContext view allows a insight into the dynamic part of the model as it was described in Section 4 based on a tree view.

8 Conclusions and outlook

Automated transformation from model to executable code is getting increasingly common to enable rapid system development and requires new appropriate model-based ways for debugging and monitoring such models directly
on an embedded target platform. In this paper we described an architecture that allows the definition of various debugging-perspectives and -views independent of the actual execution-platform, thus maximizing reusability inside the architecture.

One key achievement of our approach is that various axes of heterogeneity can be covered while still maximizing reusability:

- Development using different domain specific modeling tools: Through adapted viewpoints the user/developer can obtain adequate views on the various subsystems.
- Different development tool-chains and target systems: By implementing a driver other target-platforms running with different processors, operating system and debug interfaces can be integrated. Higher level layers may remain untouched. One example would be to utilize real-time trace abilities of embedded processor debugging hardware to reduce impact on timing.
- Distributed systems: Using various driver instances it is possible to debug models that are executed concurrently on different hardware nodes using one instance of the debugger platform.
- Heterogeneous architectures: Model based Hardware-/Software-Co-Design can be supported by Hardware-/Software-Co-Debugging. With increasing importance of reconfigurable hardware and parallel hardware- and software-development, a model based approach allows late decisions on the execution platform of functionality. Here we plan to further investigate in transparent debugging of such systems.

The architecture can be utilized to explore new approaches for assisting the developer in finding faults in the system he is developing. At the current state of implementation run-time information of object configurations in UML models as well as various run-time attributes of Stateflow diagrams can be acquired from various debuggers, stored in the run-time model and be visualized.

References


