Graphical Model Debugger Framework for Embedded Systems

Zeng, Kebin; Guo, Yu; Angelov, Christo K.

Published in:
Proceedings of the Conference "Design, Automation & Test in Europe" DATE 2010, Dresden, Germany

Publication date:
2010

Document version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Graphical Model Debugger Framework for Embedded Systems

Kebin Zeng
Mads Clausen Institute
University of Southern Denmark
Alsion2, 6400 Soenderborg
Denmark
kebin@mci.sdu.dk

Yu Guo
Mads Clausen Institute
University of Southern Denmark
Alsion2, 6400 Soenderborg
Denmark
guo@mci.sdu.dk

Christo K. Angelov
Mads Clausen Institute
University of Southern Denmark
Alsion2, 6400 Soenderborg
Denmark
angelov@mci.sdu.dk

Abstract—Model Driven Software Development has offered a faster way to design and implement embedded real-time software by moving the design to a model level, and by transforming models to code. However, the testing of embedded systems has remained at the code level. This paper presents a Graphical Model Debugger Framework, providing an auxiliary avenue of analysis of system models at runtime by executing generated code and updating models synchronously, which allows embedded developers to focus on the model level. With the model debugger, embedded developers can graphically test their design model and check the running status of the system, which offers a debugging capability on a higher level of abstraction. The framework intends to contribute a tool to the Eclipse society, especially suitable for model-driven development of embedded systems.

Keywords - embedded systems; model-driven development; model debugger; eclipse

I. INTRODUCTION

Model-driven development (MDD) has been a hot topic of research, emphasizing software development without (or with less) manual coding [1][8]. It is also one of the technologies used for embedded software development. With this approach, the development process largely depends on system models, hence accurate system modeling becomes especially important. There are already some mature modeling techniques that can be used to build systems graphically, e.g. UML, as well as various domain-specific modeling languages, such as COMDES [3][14], which has been specifically intended for embedded control applications. Along with modeling techniques, model quality assurance is of particular importance, which is accomplished via a combination of techniques, such as verification, simulation, and testing. This paper presents a complementary technique, i.e. the Graphical Model Debugger Framework (GMDF), which can be used to ensure model quality by debugging system models at runtime.

In embedded system development, system complexity leads to increased error rate in the modeling process, which is why GMDF may have a significant impact for embedded software development. GMDF mainly aims at detecting design model errors at runtime. It provides an approach for setting up a graphical model debugger to support developers, discovering system errors at the model level. The name model debugger indicates that the debugger is an evolutionary tool, targeting a model as the object of testing, instead of code. However, it could also help in finding bugs in the model transformation processes.

Fig. 1 describes a general framework for developing embedded systems. Compared with traditional embedded system development, MDD employs a modeling tool instead of a code editor to develop the system model. Besides, in order to generate executable code, MDD requires model transformation rather than a traditional compiler. Consequently, it is reasonable to introduce a model debugger situated in a similar position as a code debugger, which will be used to check and debug models.

The rest of the paper is structured as follows: Section 2 presents GMDF in more detail, including functions and operating principles. In section 3, we briefly present project implementation methods. Thereafter, we provide a demonstration to illustrate the workflow of the GMDF prototype. Section 4 introduces a number of related projects, which have inspired this work. Finally, section 5 presents concluding remarks summarizing the main features of GMDF.

II. GRAPHICAL MODEL DEBUGGER FRAMEWORK

Obviously, the aim of a model debugger is to provide developers with model-level information at runtime in order to monitor and manage the execution of the system. Therefore, it has to present the most significant model-level information as output to system developers (e.g. graphical representation of
domain-specific design models). In addition, the debugger should also offer some user control features, and these have to be provided at the model level as well (such as a model-level breakpoint function). Based on the above considerations, GMDF has been conceived so as to provide the following functionalities:

- Model-level step-wise execution and breakpoint functionality
- Model behavior animation
- Customized graphical model templates
- Automatic model abstraction and generation
- Compatibility with multi-type and multi-input models

Taking the above-mentioned functionalities as project requirements, the structural blueprint of GMDF has been designed as shown in Fig. 2.

**Fig. 2** Graphical Model Debugger Framework: Structural View

GMDF consists of three parts. The left-hand side of the diagram describes the input information provided by developers in order to debug their models. Then, a Graphical Debugger Model (GDM) is established based on user inputs, which performs the role of a server interacting with the executable code. Once the application is running, GDM starts animating the system’s running behaviors at a model level via the Graphical Model Debugger Engine. These three parts of the GMDF are further described below.

**User Input:** In order to enable the model debugger, developers have to provide two items as input: input models and executable code, marked A) on the left-hand side of the figure representing the framework.

The input models are domain-specific models, which specify system functions and behaviors that are derived from system requirements. Meta-modeling is a key concept of model-driven development, which has been introduced to specify the abstract syntax of a modeling language. There are two very common cases under consideration: Firstly, input models may consist of more than one type of model. For instance, a heterogeneous model consists of both state transition and dataflow models, whereby a state instance invokes a particular instance of a dataflow model. Secondly, complex input models may contain more than one instance of specific input models, such as multiple state machine models interacting with each other. Hence, multiple-type and multiple-instance input models have to be also taken in consideration. In principle, GMDF could accept all types of system model that follow the MOF specification [2].

Another side of user input is the executable code. After system modeling, software developers need to transform the model in order to obtain executable code. Besides normal MDD flow, GMDF requires that developers implement a predefined command interface in order to enable GDM to receive commands from the tested program. Specifically, the executable code with a command interface could be implemented automatically by a code generator based on input models. This is an active command interface solution, where the application code itself sends out commands by means of extra functional codes.

In embedded system debugging, executable code and GDM are normally operating on different computers. Therefore, hardware communication techniques are inevitably involved in the implementation of the command interface. With leading hardware access/communication techniques, the overhead of using additional codes to send commands to GDM can be eliminated, which is of particular significance in time-critical embedded applications. Hence, we suggest the use of the Joint Test Action Group interface (JTAG, IEEE 1149.1) [5] [6], which provides a passive communication solution.

JTAG can be characterized as a passive solution because real-time information/data is in fact extracted passively. In this case, GDM can send monitoring instructions to JTAG through the USB/PCI protocol. Then, JTAG takes charge of fetching real-time data from embedded chipsets (e.g. flash and/or RAM) via its hardware interface, and sending real-time information back to GDM via the USB/PCI protocol. From the list of information items fetched by JTAG, the user needs to select one or more monitored variables that are considered to be critical (e.g. variable “s” is critical if it saves state information in a state machine model). Thereafter, GDM will always be notified and then execute appropriate reactions when the selected monitored variable changes its value at runtime. Consequently, when using JTAG, a command interface is established without any code modifications.

Executable code ensures interaction with, and debugging of systems operating on real hardware at runtime, which cannot be achieved by means of other techniques, such as system simulation or verification via model checking. Meanwhile, active and passive communication solutions are compatible with various embedded system applications. Hence, it is possible to check the correctness of the model in real time via communication with the executable code, such that GDM receives specific commands (events) at particular points of execution.

**GDM:** The GDM is the core of GMDF. Based on input models and executable code, GDM is constructed as an on-call server. The executable code, which is running in the embedded
controller, works as a client continuously sending commands to GDM, as depicted in Fig. 2 B).

![Fig. 3 Meta-model of Graphical Debugger Model](image1)

Fig. 3 shows the fundamental meta-model of GDM. The meta-model provides the basic elements needed to construct a debug model from the user input meta-model. It actually establishes an event-driven finite state machine, which can be animated by the GDM. It is normally in a waiting state, listening for commands and performing the corresponding reactions.

With the aim of building the graphical debugger model (GDM), GMDF defines an “abstraction” procedure to specify the process of user model conversion, whereby GDM is obtained from the user model via a user-specified mapping. A number of rules are applied for setting up the mapping between an input model and GDM (i.e. the relations between the elements of the input language meta-model and the corresponding elements of the GDM meta-model). During abstraction, the GDM pattern provides the options of displaying objectives in different forms according to user requirements. For instance, a meta-model element “state” from input models could be displayed as a line or as a shape.

Fig. 4 is a screenshot from the current prototype, which shows the user interface for setting up the model mapping. The meta-model element list on the left-hand side is used to select an element and then choose the corresponding GDM pattern from the GDM pattern options shown on the right-hand side (e.g. Rectangle, Triangle, Circle or Arrow in Fig. 4). This is displayed in the existing pairing list (to the right of the meta-model element list) where the user can view and delete his previous pairings. Once user specified mapping is finished (triggered by the “ABSTRACTION FINISHED” button in Fig. 4), a GDM can be obtained automatically.

![Fig. 4 Graphical Model Debugger Screenshot: Abstraction Guide](image2)

Obviously, the model might be represented as a still graphical notation if animation is not considered. However, the latter is essential for model-level debugging and is an important feature of GMDF. Therefore, GDM has a command interface, as a counterpart of the command interface of conventional user input, which provides appropriate reactions when receiving commands (events) from the code being executed, i.e. specific actions to be performed on the model in response to events coming from the system under test (e.g. highlighting a GDM element). GMDF provides a user interface to setup commands associated with reaction types, which is similar to the one shown in Fig. 4.

**Runtime Engine:** A runtime engine first takes a debug model as input and displays it graphically. Next, the engine implemented as an event-driven state machine, waits for commands sent by the target embedded code. Once an event arrives, it performs corresponding actions (e.g. an animation) and other graphical model debugger functionalities.

If the actions taken are not consistent with system requirements, a bug is considered to be found. In that case, the system model has to be re-checked or re-designed, and code can be generated again. In principle, there are two kinds of bugs that can be checked with a runtime model debugger: design-errors that take place during system modeling, and implementation errors that happen during model transformation.

Design-errors arise from inconsistencies between system requirements specifications and the system model. Therefore, a model debugger aims at checking whether the application meets system requirements and fulfills its intended purpose in a “real” operational environment, rather than via simulation and verification. Design errors can be explored by a model debugger at runtime on the condition that the running code is correctly generated from the user model.

However, in embedded system development, various hardware settings force developers to adopt a hybrid-coding procedure, i.e. automatic generation of code using a code generator plus manual coding. Moreover, the software development cycle normally requires that system modeling, compilation and operation take place on different platforms. The above factors result in an increased error rate during system implementation. In general, implementation errors are less likely to arise if a high-quality automatic code generator is employed. Therefore, the primary job of a model debugger is to debug design-errors.

Obviously, there are also complex situations where both kinds of error may occur during the development process. The differentiation of different types of bugs in such a complex situation is a subject of future work, and this could possibly be another potential advantage of the model debugger technique, compared to other model quality assurance techniques.

### III. IMPLEMENTATION AND DEMONSTRATION

The framework is currently implemented as a prototype in the Eclipse platform, whose modeling project has been widely used as a foundation for model-driven software development. Sophisticated graphical modeling solutions such as the
Graphical Editing Framework (GEF), Eclipse Modeling Framework (EMF), and Graphical Modeling Framework (GMF) have been proven to be mature techniques for model-driven development. The meta-model of the framework is built in EMF, since it is intended to be a general framework being able to take any EMF-based user meta-model as input.

In real-time embedded applications, model-level animation (e.g. a state transition in a state machine model) might occur in milliseconds. Therefore, GDM animation will trace model-level behavior and always make a record of the execution trace. Consequently, the user can monitor his application by means of GDM, which animates the generated debug model by e.g. highlighting active states at runtime (see also Fig. 5).

In the following discussion, the functionality of GMDF is illustrated with the current prototype, which accepts COMDES design models as input data. COMDES is a component-based framework for distributed control systems, featuring open architecture and predictable operation under hard real-time constraints [3]. It provides a domain-specific modeling language, which is defined in terms of formal design models specifying relevant aspects of system structure and behavior [14].

System structure is described by a hierarchical data flow model, whereby an application is modeled as a network of distributed embedded actors that communicate by exchanging labeled messages (signals) using non-blocking state-message communication. Actors are modeled as component networks that are configured from prefabricated executable components such as basic (signal processing), composite, modal and state-machine function blocks.

Component behavior is generally specified in terms of functions relating input to outputs signals. However, the behavior of stateful components is usually described with state machine models (state transition graphs), which can be ultimately represented by state transition functions. Hence, actor behavior is specified by composite functions representing signal transformations - from input to output signals. Accordingly, system behavior is specified by actor-level composite functions representing the overall sequence of computation – from system input to system output signals.

COMDES treats separately functional and timing behavior, whereby a clocked synchronous model of execution is applied at actor and system levels, i.e. Distributed Timed Multitasking. With this model, input and output signals are latched at task (transaction) start and deadline instants, respectively, resulting in the elimination of I/O jitter at both actor task and transaction levels. The timing aspect of a COMDES system is managed by the underlying runtime environment, which implements the distributed timed multitasking model of computation in the context of COMDES.

The COMDES meta-model is supported by the COMDES Development Toolset [4] [13], and an application model can be obtained using the toolset. Executable code with a command interface is generated automatically based on the application model. Fig. 6, consisting of screenshots from the prototype, illustrates the workflow of the current GMDF tool.

The GMDF prototype operates on the Eclipse platform as a plug-in that can be started by the user once input prerequisites, i.e. input meta-model, input model and executable code, are available (see Fig. 6, No. 1). The workflow of the prototype program starts with the generation of an interface used to select the input files where meta-model and model data is stored (Fig. 6, No. 2). Thereafter, a model abstraction guide interface is generated via clicking on a model file in order to set up the model mapping (Fig. 6, No. 3). When the abstraction phase is finished, an initial GDM file is automatically generated, and command reaction information is subsequently added (i.e. which command triggers which type of reaction) using the command setting interface (Fig. 6, No. 4). Based on these settings, the GDM is created (Fig. 6, No. 5) and a communication channel to the embedded controller is established in the meantime. At this stage, all settings of the GDM are completed and the model debugger goes immediately to its initial state, waiting for commands coming from the code executed in the target system. Thus, the GDM continuously interacts with code execution at runtime via the command interface, depicted with a bidirectional arrow in Fig. 6. Consequently, the user can monitor his application by means of GDM, which animates the generated debug model by e.g. highlighting active states at runtime (see also Fig. 5).

In real-time embedded applications, model-level animation (e.g. a state transition in a state machine model) might occur in milliseconds. Therefore, GDM animation will trace model-level behavior and always make a record of the execution trace. The
user can then monitor the application’s behavior via a replay function associated with a timing diagram.

IV. RELATED WORK

During the process of GMDF development, we have not found a similar graphical model debugger framework on the Eclipse platform that is generic for various kinds of model, and at the same time - targeted at embedded software development. However, the idea and implementation presented in the paper have been inspired by a number of existing techniques and tools, e.g. LabVIEW, Data Display Debugger (DDD), Eclipse Coordination Tools (ECT) and UML Debugger.

LabVIEW is a well-known platform and development environment from National Instruments [9]. It is one of the top commercial software developments in graphical design and test, whose main feature is the use of a graphical programming language based on the dataflow concept, as well advanced code generation and visualization facilities based on extensive libraries of predefined components (LabVIEW blocks).

The overall process of system development in LabVIEW follows a typical MDA concept. LabVIEW users start with a “modeling process” by selecting different patterns/models and setting up the relationship among them. Once the modeling process is done, LabVIEW provides additional functions, such as code generation, simulation, visualization, etc. In particular, it is possible to validate a particular design through simulation, involving an animated graphical model of the system under investigation. This technique has been also adopted in GMDF; however, in our case it is used for visualization of the debugging process involving the target hardware/software platform under test (and not just software simulation). Another major difference is the use of abstraction in GMDF, which makes it possible to investigate systems specified in terms of both data flow and state machine models, whereas LabVIEW is limited to data flow models only.

DDD is abbreviation of Data Display Debugger, which is a graphical front-end for command-line debuggers [10]. DDD has become famous through its interactive graphical data display, where data structures are displayed in the form of graphs. Compared with a “normal” debugger, DDD has moved one-step forward with visually structured data and some information between data transfers. It adds an abstraction layer – the DDD layer, between GDB (the GNU Debugger, usually used for open source-based embedded development) and screen output. DDD and GDB run as separate processes, interacting through the GDB command line interface. Using DDD-GDB communication, DDD provides graphical data to the display instead of the original text generated by GDB.

In spite of advanced visualization techniques, DDD debugging is actually done at the coding level. It requires only source code, and needs neither system model nor MDD concepts. However, the model debugger is an opposite concept, which focuses on debugging a model, rather than code. However, the DDD project has provided valuable information concerning the command line interface, which is used to organize the graphical data, based on debugger commands. Obtaining information from the executable code is very important in order to debug a system at runtime, and the command-line technique offers a possible solution to this problem.

The Eclipse Coordination Tools (ECT) is a set of plug-ins for the Eclipse platform, supporting component-based design of applications using the Reo framework [11]. The tools currently support the following functions: graphical editing of Reo connectors and constraint automata; animating Reo connectors using on-the-fly generated Flash byte code; code generation from Reo connectors or constraint automata.

In general, ECT employs a typical model-driven approach, providing different types of representation meta-models that can
be used by developers to set up their system models, and generate executable code automatically in order to investigate system behavior via animation.

Reo is a channel-based coordination language, used to develop the ECT graphical modeling and animation tools. The user can design a Reo model on the Eclipse platform by means of a graphical editor plug-in. The model is then used to generate animation code, i.e. Flash byte code. This has prompted us to consider a possible implementation, using Flash scripts to manage animation on the Eclipse platform.

However, the ECT’s animation technique is targeted at a Reo-specific model, rather than any EMF-based model, as required in this work. On the other hand, the adopted modeling concept is not intended for embedded system development and model-level debugging is not supported.

The UML Model Debugger [12] is developed by the Model Driven Engineering Technologies group, IBM Haifa R&D Labs. UML is a general-purpose modeling language that has been widely used in system modeling. It has been chosen as an input language for the above project aimed at developing a model debugger on the Eclipse platform. Nowadays, the input model for the tool is limited to UML only, but there are plans to extend it to any MOF-supported model (much in the same way as GMDF).

IBM’s UML Debugger project has given us a lot of inspiration and makes our own project even more promising. However, there are still differences between GMDF and the UML debugger. As a general-purpose modeling language, UML is not particularly appropriate for certain application domains. Conversely, GMDF is aimed at domain-specific models, which are usually intended for embedded systems development. Therefore, GMDF provides an innovative way, in which the users can observe model behavior in real-time, while the application code is executing on the embedded target platform.

V. CONCLUSION

The paper has presented the Graphical Model Debugger Framework, which has been specifically designed to support model-driven embedded system development. The framework facilitates the testing of embedded application, and helps developers focus on the behavior of the designed system (e.g. state machine, data flow) at the model level, rather than observing a variable value or a line of code at code level. With the support of a meta-model, a debug model can be derived from the user’s input meta-model and application model. Then a runtime engine is responsible for displaying the debug model visually, and listening to commands sent by the running embedded code generated from user models. Bugs can be thus found if exceptional actions take place. An initial prototype of the framework has been developed on the Eclipse platform.

REFERENCES