Model Based Debugging and Testing of Embedded Systems Without Affecting the Runtime Behaviour

Michael Spieker[1], Arne Noyer[2], Padma Iyenghar[1],
Gert Bikker[2], Juergen Wuebbelmann[1], Clemens Westerkamp[1]
1- Institute of Computer Engineering, University of Applied Sciences Osnabrück, Germany
2- Institute for Distributed Systems, University of Applied Sciences Wolfenbüttel, Germany
{m.spieker, p.iyenghar, j.wuebbelmann, c.westerkamp}@hs-osnabrueck.de
{arne.noyer, g.bikker}@ostfalia.de

Abstract

Model-based approaches to develop software for embedded systems are becoming increasingly important in the last ten years. In this context, it is desirable to carry out debugging at the model level in conjunction with the model-based development of embedded software. This paper presents such a model-based debugging approach which can be used to visualize the behaviour of embedded software on a host PC. The various constraints in embedded software development and execution pose a big challenge to monitor the embedded system without affecting its runtime behaviour. The focus of this work is to describe the underlying monitoring mechanism and the methodology to visualize the embedded system behaviour in real time. This includes different approaches to get the trace data from the target without or with minimal influence on the runtime behaviour of the monitored embedded system.

1. Introduction

Nowadays embedded systems are used in several application fields. Whereas, such devices are used not only in day-to-day appliances such as mobile phones, but also for critical medical appliances such as nutrition and syringe pumps. Particularly in the critical region it must be ensured that the software developed for the embedded system will work correctly under all circumstances and exclude failures or disturbances. To ensure this, the behaviour of the program running on the embedded system has to be examined carefully. In more than 80% of embedded software engineering projects, “C” is the most commonly used high-level language for programming the microcontroller, because it is possible to program close to the hardware [1]. This works very well for smaller projects and also allows the subsequent debugging of programs. Due to the growing complexity of embedded software engineering projects, it is getting harder to maintain an overview of the dependencies between different modules written in high level programming languages such as “C”. So the implementation in C code means more effort [2] and the depending tests for a correct behaviour of the Software getting much more complex. To solve this problem, model driven development is becoming more important in the sector of embedded systems to describe the behaviour of the underlying software [3][4][5].

Embedded systems often have limited resources (e.g. RAM/ROM, processing power) and stringent real time requirements. This necessitates the usage of efficient code generators (in model-based development tools for embedded systems) which focus on minimizing resource usage (e.g. code-size, execution time). The modeling tool under consideration in this paper is Merapi-Modeling [8] which includes state, sequence and class diagrams. With these diagrams it is possible to describe the behaviour of an embedded software application. The aforementioned diagrams are specified in the standardized Unified Modeling Language (UML) [6]. To get an executable application from the model, C-code is generated by a code generator, that can be compiled exactly as before directly for a specific microcontroller. The code generator from Merapi-Modeling [8] is specially focussed on these requirements.

With the use of code generators (i.e., in model-based tools) the failures on C level can be minimised but still there can be failures on the model level. To successfully find bugs on the model level in an efficient way, there is the requirement to debug and execute tests on model level. This makes it essential to monitor the behaviour of an executed application on the model level to find the reason of an abnormal behaviour. This is possible with the animation mode from a modelling tool, Rhapsody[13]. But this solution needs a lot of instrumentation for the generated code which inflate the code size with a factor of 1,5 to 3,5. The use of this instrumentation within an embedded system can cause two problems: 1. the available memory from the device is too small to store the additional instrumentation in addition to the application code. 2. The execution time of the instrumentation (e.g for monitoring, debugging, etc) could violate the real time requirements for the application. To monitor the behaviour of the application a small instrumentation of the generated C-code from the model is necessary. This instrumentation gives information about actions happen on the device.

The approach introduced in this paper gives the possibility to monitor the application with a minimal instrumentation which can be ignored in most cases. At the runtime of the application the monitored behaviour can be displayed in the modelling tool Merapi-Modeling and the UML Target Debugger (UTD) [10]. The remaining of this paper is organized as follows.

Following this introduction section, chapter 2 discusses related work. Chapter 3 presents the modelling tool Merapi-Modeling in combination with the UTD. In chapter 4 different monitor approaches are presented, which are used to get trace data from the device (to the host). Chapter 5 concludes this work and gives an outlook for future work.

2. Related Work

In model-based software engineering with the UML, different types of diagrams are used to describe the architecture and the behaviour of the software. To visualize and debug the behaviour of an embedded software at the model-level, various UML diagram types such as state charts, sequence diagrams and timing diagrams can be employed.

Embedded Systems are often also reactive systems that react to events in their environment. Their behaviour differs depending on their current state and their previous behaviour. Therefore state charts are predestined to model the behaviour of components of reactive systems [11]. Animating state charts allows analysing how an application really reacts on different events and how the current states of components change, whose behaviour was defined by using state charts. For realizing the animation there a different possibilities, that are discussed in the following.

One solution is that a simulation is executed on the model, even before code gets generated and compiled. By doing this waiting time for code generation and compiling can be avoided. However, there is the challenge that generated code should behave exactly as it is predicted by the simulation. Furthermore, the model can contain expressions, which are specific to the programming language, for which code should be generated. This makes a simulation a very complex task. An example of a software that offers simulation using state charts is Yakindu [12]. Here there are special state chart expressions, which have to be independent of a programming language. The model must not contain any expressions which are specific to a certain programming language. The simulation engine, which is used for the animation, is based on the functionality of the (Java-) code generator of Yakindu to close the gap between the simulation and the generated code as much as possible. This solution is for embedded systems for our purposes not practical. It is unacceptable for debugging when there may be any difference between the simulation and the actual behaviour. The application could behave differently on the target as it can be determined in a model-level.

Therefore it is conclusive to perform the animation directly during the execution of the application. The application's behaviour must then be observed by the animation. An obvious solution is that a communication between the animation and the application is established. One such commercially available tool which provides the aforementioned functionality is IBM Rational Rhapsody [13]. It allows setting the instrumentation mode for a model to "Live Animation". By activating this option the generated code of each class is instrumented to provide information about the application's behaviour at runtime. This has the consequence that the total size of the application increases by a factor of 1.5 – 3.5 depending on the number of classes. The increase of the application's size and the associated increase of needed memory and additional computing time are not acceptable for many embedded systems. The additional instrumentation as part of the application is not suitable because of runtime or memory reasons. Another problem is the communication between the target system and the host PC, which is usually realized by using Ethernet. For most embedded systems, it is not possible/planned to implement an Ethernet connection – be it for hardware reasons, that the resources do not have sufficient power or that needed pins are already used by the application otherwise. On the other hand, there may not be enough memory for an Ethernet stack or there is no suitable operating system.

In summary, it can be stated that model-based debugging of embedded software systems is highly desirable. However, the existing tools which support model-based debugging of embedded software may result in influencing the runtime behaviour of the underlying application. Therefore a major challenge is to enable model-based debugging of embedded software without or with minimal influence on runtime behaviour of the embedded system. To achieve this goal, this paper introduces a minimally intrusive monitoring mechanism for model-based debugging and visualizing the target behaviour in real time.

3. Merapi-Modeling

At the Ostfalia University, during the execution of several industry-sponsored projects, UML is used as the modelling language for design and development of embedded software. Based on the experience that was gained, the question was asked, which elements of the UML are really needed for software development for embedded systems and whether it makes sense even to violate the UML standard in some cases [14].

From the results of the research project *Embedded Modelling Environment* [17] the model-based development environment Merapi-Modeling is developed. This IDE has a special focus on the usage of UML-based elements for the development of embedded systems. Thus, a "lean" IDE can be made available for developers, which offers a high degree of usability.

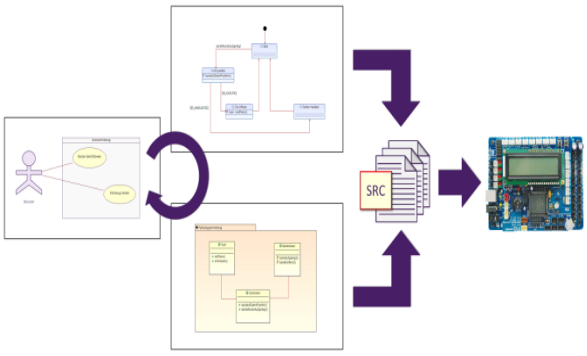

Figure 1: Modeling instead of programming of embedded systems

Particularly important for the modelling of embedded software are sequence diagrams, class diagrams, object diagrams and state charts. Classes and objects are used to describe the structure of the system, while state charts are used to define the behaviour of the objects. With sequence diagrams scenario test cases can be described. Often these diagrams are sufficient in order to iteratively design software for embedded systems (Figure 1) and can subsequently be used to generate code.

The code generator of Merapi-Modeling tool is specialized to generate C code that operates most efficiently on embedded systems. The code generation is also designed to support the Willert Realtime Execution Framework (RXF) [7]. The RXF is a very efficient interface between UML models and the target platform, which consists of the CPU, the compiler and a runtime system or a RTOS. A product version of the RXF can easily be selected in the user interface of Merapi-Modeling, so that it will be integrated in the generated code.

Due to the use of Eclipse [15] and its plug-in concept as a basis for Merapi-Modeling, the code generation of it can still be exchanged flexibly and / or be customized. Furthermore, an advantage of using the eclipse plug-in concept is that Merapi-Modeling can be integrated into an existing Eclipse Platform, which is already used by developers to write C Code.

The UTD is a tool by Willert Software Tools [7], which can monitor an application on the target that uses the RXF. It can be used to debug the generated code model based. For debugging it offers its own sequence and timing diagrams. To achieve that the UTD (Figure 2, in the centre of the figure) can be used to debug code that was generated by Merapi-Modeling and that also in Merapi-Modeling information about the executed application is available to animate state charts, some preparations have to be made.

For the compatibility between Merapi-Modeling and the UTD an XML file is generated in addition to the code (Figure 2), which is needed by the UTD to function properly. The XML file has to contain information about address ranges of classes among other things [16]. But during the code generation the information about addresses is not yet available. The code generation operates on the model and can only extract data like the name of elements and write it into the XML file. The address ranges of the elements have to be gathered after compiling the code by accessing a map file, which gets created by the compiler. Then the address information can be integrated into the xml file at the correct positions.

Merapi-Modeling allows the user to use state charts for model based debugging. Therefore the UTD can be started from Merapi-Modeling directly. Then a communication between Merapi-Modeling and the interfaces of the UTD can be established. The UTD has two TCP/IP-Interfaces for this purpose. By using the Log-Interface data from the target can be received, while the Test-Interface can be used to send messages to the target – for example to inject events into the RXF.

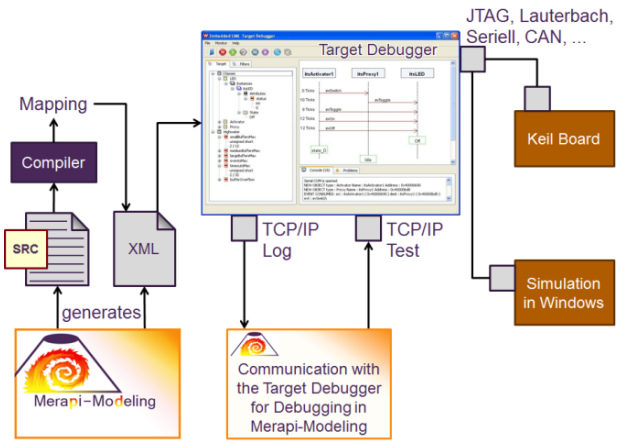

Figure 2: Communication between Merapi-Modeling and the Target Debugger

Thereby information about active objects and their current states are available in Merapi-Modeling. For each object an animated state chart can be opened, which can be used for debugging. By doing this it can be closely monitored, in what state an object is at a time (Figure 3).

Also information about the event, which occurred when the current state of an object changed, is available. Thus, the transition can be visually highlighted, that was followed to the new state. Because the state chart animation is performed directly in Merapi-Modeling, all information about the model is there and the animation knows, on which events an object is reacting. This makes it possible that in the debugging user interface events can be selected for objects, which are then injected into the target. For this purpose a message is sent to the Test-Interface of the UTD that actually does execute the event in the target. It can be examined immediately, how the object behaves and changes its current state by analysing the object's animated state chart. Beside the functionality to inject events, the user can stop the execution of the application on the target and can run the execution step by step. When the user controls these functionalities from the user interface, also messages are sent to the UTD.

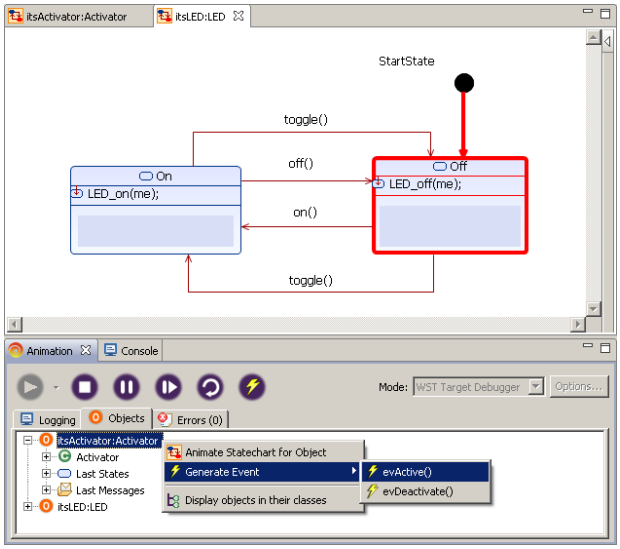

Figure 3: An animated state chart in Merapi-Modeling

4. Target Monitoring

In this chapter different approaches are described to capture the trace data via a monitor from the target. This trace data is needed to display the behaviour of the application in animated diagrams in the UTD or Merapi-Modeling tool. There are different kinds of monitors to capture the trace data from the target which are described in the following sections 4.1 – 4.3. The three alternatives are hardware, software and on-chip monitoring mechanisms. This paper focuses on software and on-chip monitoring mechanisms (with minimal influence on the runtime behaviour of embedded software). Whereas, a hardware monitoring solution (e.g. using In Circuit Emulators-ICE) may not be considered as a practical choice for model-based debugging, a theoretical description of the pros and cons of such an approach is also provided below.

4.1 Software Monitor

A software-based runtime monitoring solution, introduced in [9], can be encapsulated with the underlying embedded software application (i.e., source code). This monitor has a static size and is independent from the model and is used as interface between the RXF / user code and the UTD displayed in Figure 4.

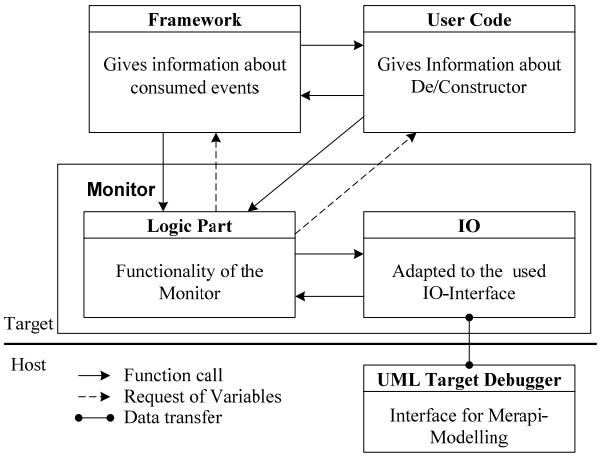

Figure 4: Position of the Monitors in the application

This software monitor collects information about the behaviour of the application on model level during runtime. This information (i.e., the trace data) is sent over an IO interface of the monitor to the host. The UTD on the host receives and processes this trace data. This solution is termed as a software monitor approach, because the process of collecting the trace data as well as sending the trace data is carried out by the monitoring software in the embedded system. For the connection between the monitor and the UTD, various debug interfaces such as EIA-232 or JTAG-based interfaces can be used. The IO block displayed in Figure 4 is actually implemented for two different kinds of connections: UART (EIA-232 on host side) and a data transfer using the JTAG interface. The trace data about the behaviour of the application descends from the RXF Framework and a small part from the generated code from the model. The required resources for the monitor such as the memory and execution time depending on the debug communication interface to the UTD are displayed in Table 1.

Table 1: Resources of the software monitor with different IO interfaces

Interface	Memory [Byte]		Additional time for the RXF to handle an event [factor]
	RAM	ROM	
EIA-232	250	1290	8,92
JTAG	84	1194	23,94

The additional time of the RXF (i.e., the RTOS framework used in the prototype) to handle an event is caused by the additional execution time of the monitor. This can result in performance degradation, which is also evident from Table 1. However, this additional software monitor may result in performance degradation affecting the run time behaviour of the embedded system.

4.2 Hardware Monitor

A possibility to reduce the limitations by the software monitor is the use of an In Circuit Emulator (ICE). This ICE is an additional trace hardware which can be used to monitor all program steps and memory accesses to reconstruct the whole program flow on the Host. This ICE is connected to the data and address bus of a bond out chip which offers outlined connectors to these busses. This allows to monitor the behaviour of the application without any instrumentation so that the runtime behaviour is not affected by the monitoring process. The disadvantage of this approach is that it is very costly and requires intelligent filter technologies for the trace data stream. This trace data filter must be matched to the model to reduce the trace data, because the unfiltered trace data stream has a very high data rate which cannot be processed by the host. Another disadvantage is that not every target platform is available as a bond-out chip with the needed connectors for the ICE. The main disadvantage of this solution is that the target system cannot be stimulated from external application running on the host. This makes it impossible to execute test cases or stimulate the system from the host side like injecting events, using an ICE.

4.3 On-chip trace Monitor

An alternative is the on-chip trace hardware supported from several embedded architectures like the Cortex-M3 architecture. With this on-chip trace technologies it is possible to get the same kind of trace data as described in chapter 4.2. All trace operations are done by the on-chip trace hardware without influencing the execution of the application. But the trace data is not as complete as the approach shown in chapter 4.2. For the instruction trace only the branch operations are transmitted to the host which can be used to reconstruct the complete program flow. For the data trace the Cortex-M3 architecture includes a Data Watchpoint and Trace (DWT) Unit which can be used to monitor read and write accesses of a defined memory space from the RAM. The DWT Unit generates a DWT Package if a write access to a monitored memory space occurs. This DWT Package includes the content of the monitored memory and is send via the Serial Wire Output (SWO) to the host shown in figure 4. Because the data stream of the SWO cannot be handled direct by the host, an additional trace hardware is needed to convert the data stream from the SWO into an ethernet or serial stream which can be processed by a computer. The DWT unit includes a small internal hardware buffer to store the DWT packages. With this buffer it is possible to trace four memory accesses of the application directly after each other. If there are more than four memory accesses after each other the internal buffer of the DWT unit overflows and trace data gets lost.

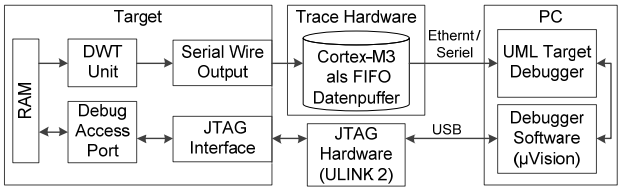

Figure 5: On-chip Trace Hardware

Because it is not possible to monitor every memory access on the target a small instrumentation is necessary which reduces the trace data to a minimum. Therefore four debug variables are used for obtaining the trace data. These are monitored by the DWT unit only for write accesses. These variables are global with a fixed address and have the following names:

1. debugEvent
2. debugConstructor
3. debugDestructor
4. rxfTicks

The first three variables are used to monitor the actions events, constructor calls and destructor calls. After one of these actions is performed the information (described in [9]) about this action is written into the depending variable. The variable rxfTicks is part of the RXF and is incremented by the system tick of the used RTOS. By monitoring the variable rxfTicks the UTD gets timing information about the actions happening on the target (which are monitored by the first three debug variables). So the timing behaviour of the application can be displayed in the UTD. The required memory size and execution time of the instrumentation is displayed in Table 2.

Table 2: Resources for on- chip Instrumentation

Memory [Byte]		Extra time for the RXF to handle an event [factor]
RAM	ROM	
24	74	1,03

In comparison to the software monitor approach this approach needs much less memory. Additionally the timing behaviour is influenced in a minimal way which can be ignored. It combines the advantages of the software monitor approach and hardware monitor approach.

It is possible to interact with the target over the Debug Access Port (DAP) and to inject test stimuli in the form of events using the proposed approach. So it is possible to run test cases with test applications on the host side while the test case is executed on the target. These test applications use the UTD to interact with the target which makes it possible to analyse the executed test cases at runtime.

5. Conclusion

The existing model-based debugging solutions use source code instrumentation (static/dynamic) to monitor an application's behaviour. This results in the increase of the application's size, needed memory and additional computing time, which is not suitable for many embedded systems. Also it could alter the timing behaviour of the system.

The software monitoring approach discussed in section 4.1 provides an opportunity for debugging embedded software systems with static instrumentation on target. When using the on-chip based approach for monitoring that was introduced in section 4.3, only small instrumentation is needed, which can be ignored in size and execution time. This means that there is no substantially increase of needed memory and computing time. Then model-based debugging has almost no influence on how the embedded system behaves. The UTD observes the application and allows using sequence and timing diagrams to visualize its behaviour.

Merapi-Modeling is a modeling tool, which provides state charts to define the behaviour of objects. For debugging purpose it establishes a connection to the UTD to receive information about an executed application on a target. By using this information the state charts can be animated and it is possible for using the state charts for debugging.

For further work it is planned that test cases can be specified in Merapi-Modeling by using sequence diagrams. Then events are injected into the target automatically according to the defined test cases and it is verified, if the application behaves like it was defined within the test cases.

6. References

[1] Christof Ebert, Capers Jones, "Embedded Software: Facts, Figures, and Future," Computer, IEEE pp. 42-54, April, 2009

[2] Selic, B.; "The pragmatics of model-driven development" Software, IEEE, vol.20, no.5, pp. 19- 25, Sept.-Oct. 2003

[3] G. P. Gu, D. C. Petriu: Early Evaluation of Software Performance based on the UML Performance Profile (In CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, Seiten 66 - 79, 2003).

[4] M. Mura, L. Murillo, M. Prevostini: Model-based design space exploration for RTES with SysML and MARTE (In Forum on Specification, Verification and (FDL), 2008).

[5] S. Demathieu, F. Thomas, C. André, S. Gérard, F. Terrier: First experiments using the UML profile for MARTE (In 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pages 50 - 57, 2008).

[6] OMG: Unified Modeling Language *http://www.omg.org/spec/UML* (April, 2012)

[7] Willert Software Tools GmbH *http://www.willert.de* (April, 2012)

[8] Merapi-Modeling: Model based development environment. Research project from the University of Applied Sciences Wolfenbüttel *http://www.ostfalia.de/cms/de/ivs/ease/Merapi/index.html* (April, 2012)

[9] P. Iyenghar, M.Spieker, P.Tecker, C.Westerkamp, J.Wuebbelmann, "UML Target Animation: A Comparison of Debug Interfaces for Design Level Debugging", in Proceedings of System, Software, SoC and Silicon Debug conference, S4D 2010, Sept 2010.

[10] UML Target Debugger *http://www.willert.de/assets/Datenblaetter/DatBl-Embedded-UML-Target-Debugger-V5.0en-2011-.pdf* (April, 2012)

[11] David Haril, Michal Politi: Modeling Reactive Systems with Statecharts - The Statemate Approach. Mcgraw-Hill Verlag, 1998.

[12] Yakindu – itemis AG *http://www.yakindu.de/* (April, 2012)

[13] IBM Rational Rhapsody, Version 7.6.2 *http://www.ibm.com/software/awdtools/rhapsody/* (March 2012)

[14] Gert Bikker, Kevin Barwich, Arne Noyer: Domänenspezifisch entwickeln mit UML. Embedded Software Engineering Kongress, Dezember 2010

[15] The Eclipse Foundation: Eclipse *http://www.eclipse.org/* (April, 2012)

[16] P.Iyenghar, E.Pulvermueller, C.Westerkamp, M.Uelschen, J.Wuebbelmann, "Model-Based Debugging of Embedded Software Systems", Gesellschaft Informatik (GI) - Softwaretechnik (SWT), August 2011

[17] Gert Bikker, Open tool framework for developing embedded systems (Embedded Modeling Environment), 2009, *http://www.ostfalia.de/export/sites/default/de/ivs/ease/upload/kurzinformation__ingenieurnachwuchs-projekt-embeddedme-bikker02.pdf*