Chapter 5
Model-Based Debugging of Embedded Software Systems

Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Juergen Wuebbelmann and Michael Uelschen

5.1 Introduction

Software engineering has gone through several paradigm shifts (assembly language → structural programming → object-oriented → model-driven development (MDD)), driven by the requirements for building more complex systems. The inherent necessity to achieve reliable systems, inline with these paradigm shifts, has ushered in copious verification and validation techniques.

However, debugging and runtime monitoring remains the widely used process for finding and resolving defects (bugs) that prevent correct operation of the underlying software system. Debugging tools, in general, help to identify errors at the various stages of the software development process. Some of the commonly used traditional debugging tools involve "printf" statements, data monitors, and operating system monitors [17]. On the other hand, some sophisticated techniques available in the embedded software development tools are profilers, memory testers, and execution tracers [5, 8, 32], to mention a few.

P. Iyenghar (✉) · E. Pulvermueller
Software Engineering Research Group, University of Osnabrueck, 4469, 49069 Osnabrueck, Germany
e-mail: piyengha@uos.de

E. Pulvermueller
e-mail: elke.pulvermueller@uos.de

C. Westerkamp · J. Wuebbelmann · M. Uelschen
University of Applied Sciences, 1940, 49009 Osnabrueck, Germany
e-mail: c.westerkamp@hs-osnabrueck.de

J. Wuebbelmann
e-mail: j.wuebbelmann@hs-osnabrueck.de

M. Uelschen
e-mail: m.uelschen@hs-osnabrueck.de

D. Lettnin and M. Winterholer (eds.), *Embedded Software Verification and Debugging*, Embedded Systems, DOI 10.1007/978-1-4614-2266-2_5

Observing and examining the behavior of software execution at runtime can be termed as runtime or online software monitoring and verification. Embedded systems present particular challenges for monitoring, which necessitate a nonintrusive or a minimally intrusive monitoring methodology. This is especially true for resource constrained, deeply embedded systems (e.g., 16-bit systems with 64 KiByte memory). Runtime monitoring techniques used for several application purposes [5, 8, 32] often employ a target output/computer to diagnose and interpret the results in formats such as plain text and graphs. However, with the advent of model-driven methodologies, the applicability and usage of models, such as Unified Modeling Language (UML) [4] diagrams, are under evaluation for model-based runtime monitoring and debugging of software systems.

In the field of real-time embedded software systems, model-based debugging and visualizing embedded software systems, using diagrams, such as sequence and timing diagrams, presents an exciting outlook. Model-based visualization of embedded software system behavior (at runtime), using sequence diagrams and state charts, is possible in proprietary MDD tools, such as Rhapsody [14]. However, such existing runtime monitoring and debugging methodologies are not well-suited for applicability in deeply embedded systems, primarily because of the monitoring overhead involved. The following section enumerates the drawbacks of debugging, runtime monitoring, and visualization of target behavior in real time, in the state-of-the-art tools and methodologies.

5.1.1 Problem Statement

The drawbacks of model-based debugging and runtime monitoring, which involve extensive source code instrumentation in the underlying software, are discussed below. The crux lies in the applicability of such techniques in deeply embedded software systems. An example of such a resource constrained embedded system is a 16-bit system with less than 64 KiByte memory.

- **Significant instrumented code size** The instrumented code (for debugging and monitoring) increases with an increase in the application size. The instrumented code varies based on the application size and complexity. Hence, there arises a question of scalability and applicability of such an approach in debugging small embedded software systems. For instance, an existing MDD tool [14], while supporting model-based monitoring of embedded systems, makes use of techniques such as dynamic source code instrumentation or downloading a significant instrumented code on the target.
- **Requirement for sophisticated interfaces**: Often, sophisticated interfaces and communication protocols are required to download the instrumented code and visualize the behavior of the target in real time at the host computer. For instance, let us consider the MDD tool Rhapsody [14], which provides a "live-animation" feature for model-based visualization of the target behavior in real time.

It introduces significant (and dynamic) source code instrumentation overhead. Further, it requires sophisticated debug communication interfaces (e.g., TCP/IP over Ethernet) to download the debug code on the target and visualize the behavior at the host, using UML diagrams. It is intuitive to perceive that such techniques would further result in protocol and performance overhead during debugging and/or runtime monitoring. Moreover, such interfaces are not necessarily available in memory-size constrained, deeply embedded targets.

- **What you verify is not what you deliver**: The instrumented code is not only significant, it is often removed after the debugging process/verification is completed. This implies that the behavior of the RTESS during debugging may not remain the same after the debugging process is complete. For instance, there could be a change in the program-instructions or clock cycles before which the specific system code is executed (because of the execution cycles of the instrumented code). This necessarily means that the system that is debugged/verified is not the same system that is delivered as the end-product.

Thus, the existing tools introduce unbounded overhead and highly intrusive monitoring mechanisms for model-based debugging and visualization of real-time behavior of targets, using UML diagrams. Clearly, such approaches are not suitable for applicability in deeply embedded systems.

5.1.2 Contribution

Based on the problems stated above, it is clear that there is a need for an integrated model-based debugging framework and monitoring approach, which provides scalable model-based debugging for deeply embedded systems. Such a monitoring methodology is referred to in this paper as the time and memory-size aware runtime monitoring.

One such model-based debugging approach, which addresses the aforementioned limitations, is discussed in [20]. With this model-based debugging approach, the behavior of memory-size constrained RTESS can be visualized in real time, using UML sequence and timing diagrams (at the design level using a minimally intrusive target monitor). Performance metrics and evaluation of the debugging approach proposed in [20] is discussed in [19]. This book chapter elaborates on the debugging approach presented in [19, 20] and extends it with the following novel contributions.

- A brief outline of the model-based debugging approach is provided.
- The requirements of a time and memory-aware runtime monitoring methodology, toward applicability for model-based visualization of target behavior, are elaborated.
- Two variants of the proposed monitoring methodology, i.e., (a) software and (b) on-chip monitoring are presented and their prototype implementation is discussed.

- Frame formats for sending the trace data between host and target in the proposed model-based debugging approach is elaborated.
- An experimental evaluation of the proposed monitoring mechanisms is provided. A discussion and evaluation of the proposed approach, in comparison with the existing approaches, is presented.

The remainder of this paper is organized as follows. Section 5.2 deals with related work. The model-based debugging methodology is outlined in Sect. 5.3. The runtime monitoring methodologies are discussed in Sect. 5.4. A prototype of the monitoring methodologies and an experimental evaluation are discussed in Sect. 5.5. The performance metrics of the proposed approach are discussed in Sect. 5.6. A discussion on the salient features of the proposed approach is presented in Sect. 5.7. Section 5.8 concludes this paper.

5.2 Related Work

The exponential growth of the embedded software systems necessitates the use of advanced and automated methods of development and testing. In this context, Model-Driven Architecture (MDA), proposed by the OMG [28], promises several advantages and superiority, superseding the traditional way of developing the embedded systems. This is supported by the studies conducted in [3, 9, 22, 23].

The MDA model is related to multiple standards including the UML. UML comprises of general purpose diagrams and profiles. UML profiles introduced by the OMG consist of a set of new stereotypes for a particular domain. The widespread applicability of UML (general purpose diagrams and profiles) as a modeling language for embedded systems is evident from the numerous studies in the literature [20, 22]. In our proposed model-based debugging methodology we use UML to specify the design model (e.g., class diagram, state charts, etc.). UML interaction diagrams, such as timing diagram and sequence diagram, are used for visualizing the target behavior in real time at the design level in our approach.

Irrespective of the evolution in embedded software engineering, the model-based tools for embedded systems continue to use monitoring approaches for applications such as debugging and testing. Software monitoring has been in use for over 35 years for a variety of domains and application purposes [5, 29]. Some of the domains in which runtime monitoring is applied include, distributed systems, fault-tolerant systems, real-time critical systems, and embedded systems [32].

Domains such as the embedded systems present particular challenges for monitoring, as the system internals may not be easily observable and have limited resources or real-time constraints. Hence, utmost care must be taken to avoid incurring extensive runtime overhead in the form of additional resources (e.g., memory, time). A technique for time-aware instrumentation of embedded software is discussed in [8]. It demonstrates how instrumentation can be used to maximize trace reliability and computing the minimal trace buffer size. However, most of the aforementioned

monitoring approaches, except [8], concentrate on applications running on desktop computers. They do not consider their impact on the time or memory requirement of the applications.

Monitoring systems are classified according to the probes/instrumentation used, as hardware, software, hybrid, and on-chip monitoring. This classification is used both in early and late surveys on monitoring [32]. A time-aware, software-based instrumentation methodology is presented in [8]. Similarly, a software-based monitoring approach discussed in [20] is used for visualizing the behavior of targets in real time, using UML diagrams. Software monitoring with controllable overhead is discussed in [13]. A hybrid monitoring approach using a proprietary tool and In-Circuit Emulator (ICE) for testing of embedded systems software against UML models is discussed in [11]. A survey on monitoring approaches is presented in [32]. However, the applicability of the monitoring approach for visualizing target behavior in real time (esp. in deeply embedded targets) is missing in [8, 11, 13, 20, 32].

In this paper, a model-based debugging mechanism, which makes use of a target monitor in the embedded system and a target debugger at the host computer (design level), is discussed. Two variants of the target monitor, namely (a) software and (b) on-chip monitoring mechanisms, are presented. The target monitor sends trace data pertinent to the behavior of the target to the host side by back annotation. Some related work pertaining to target monitor and back annotation of trace data from target to host are discussed here. A study conducted in [12] deals with back annotations and continuous feedback about target behavior to the host side. This study is conducted on Java-based microprocessors for worst-case execution time (WCET) analysis. However, Java-based microprocessors are not necessarily the preferred choice in a memory-size constrained RTESS. In [21], an implementation of an event-driven hardware/software collaborative monitor system, enabling system-level monitoring on target at different abstraction levels is presented. In this work, the monitor system is claimed to collaborate seamlessly with other components in a model-driven testing tool chain. Though [12, 21] deal with back annotation and monitor systems respectively, a collaborative approach toward model-based debugging using a model-based target debugger is unavailable. Similarly, the suitability of runtime verification and monitoring approaches for embedded systems is discussed in [32]. Nevertheless, monitoring approaches for supporting model-based runtime visualization of embedded system behavior is missing in [32].

Commercial MDD-based tools such as [2, 7, 14] are limited in terms of debugging in real time at the design level for RTESS. Moreover, these tools and their model-based debugging feature cannot be used for small RTESS, because of memory, performance and protocol overhead (TCP/IP over Ethernet, etc.) with the target system. All these result in potentially inefficient communication between the target and the host. The dynamic source code instrumentation introduced by these tools could also result in affecting the real-time behavior of the RTESS.

Thus, it is evident that even though model-based development and debugging is being used for RTESS, applicability of model-based debugging approaches for deeply embedded systems is still in fledgling stages (both in academia and commercial tools).

5.3 Model-Based Debugging Framework

The proposed framework for model-based, design level debugging of deeply embedded systems discussed in this paper, is shown in Fig. 5.1. An outline of the proposed framework is provided here to place in context the main focus of this paper. This paper concentrates on a time and memory-aware runtime monitoring methodology for debugging and visualizing the (deeply embedded) target behavior in real time (on the host).

5.3.1 Overview

The proposed framework comprises of a target debugger on the host side with a runtime monitoring solution on the target side as seen in Fig. 5.1.

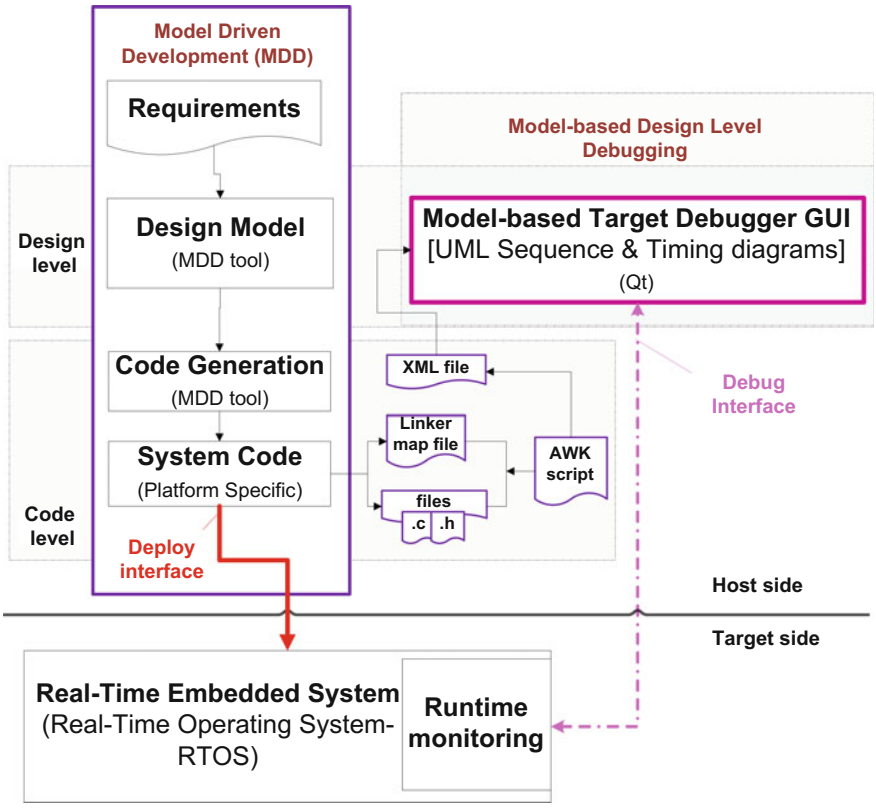


Fig. 5.1 Model-based design level debugging for embedded systems

5.3.1.1 MDD Phase and System Code

Based on the requirements, the design model for the embedded software application to be developed is specified (Fig. 5.1) using a modeling language (e.g., UML using a MDD tool Rhapsody [14]). The main functionality of the application can be specified, for example, in UML class diagrams. The detailed functionality and reactive behavior of each class can be represented as a state diagram, for example, using UML state charts. The next step is the automatic code generation process, which is most often supported in MDD tools (e.g., [14]). In the prototype, the system code is generated in C. The system code thus obtained can be executed on the target after cross-compilation. During the compilation of the system code, an XML file is generated at the host side (Fig. 5.1) using an AWK script (which parses the linker map file, source, and header files for a given project). The generated code is deployed using a deploy interface (such as the Keil Ulink [27] adapter) on the target, which runs a real-time operating system (RTOS). An RTOS framework suitable for resource constrained embedded systems, namely OORTX-RXF [33], is used in the prototype. However, this approach can also be applied to other UML-based modeling tools (e.g., [7]) and modeling alternatives, such as Matlab/Simulink [26] and LabView [24]. In the prototype, the system code is generated in the programming language C. Note that this approach is independent of the modeling language, the language in which the system code is generated and the underlying RTOS in the embedded system.

5.3.1.2 Target Debugger Graphical User Interface (GUI)

The model-based, design level target debugger graphical user interface (GUI) on the host side receives back-annotated trace data from the target monitor using a debug communication interface. The trace data provides details about the target behavior in real time. The target debugger reconstructs the behavior of the target in real time using UML interaction diagrams, such as the sequence diagram and the timing diagram in the GUI. The target debugger is implemented in the User Interface (UI) framework Qt [30].

The target debugger GUI consists of three blocks as shown in Fig. 5.2. Block (a) shows the classes, objects, states, and attributes available in the embedded software running on the target system. Block (b) displays the sequence of events and the temporal behavior of the target using UML sequence diagrams with time stamps (sequence diagram tab) and UML timing diagrams (timing diagram tab). Block (c) displays the reconstructed messages on the host side (based on back-annotated data from target and the XML file).

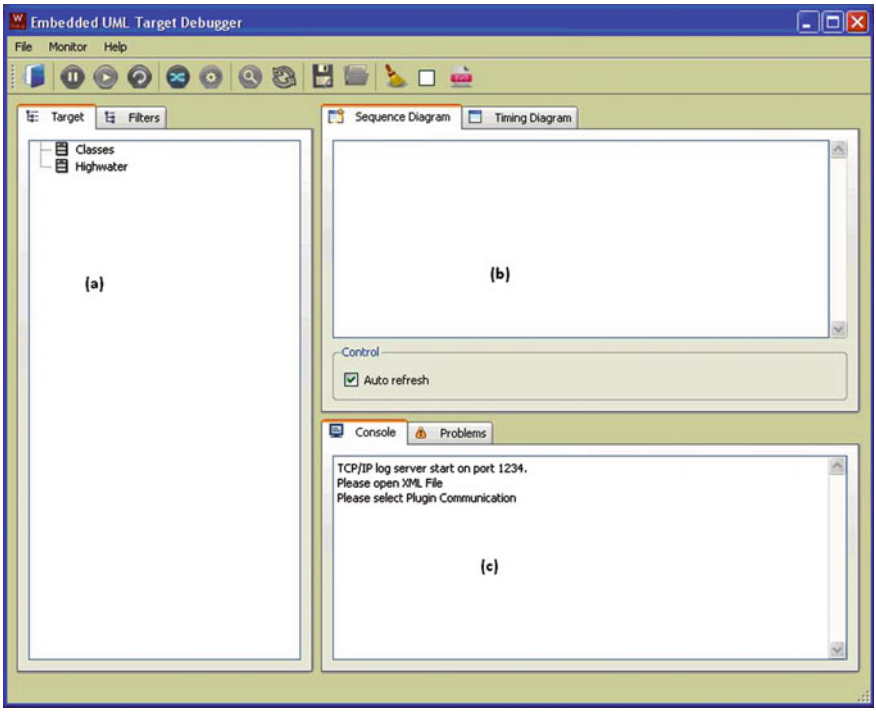


Fig. 5.2 Model-based target debugger GUI

5.3.1.3 Target Monitor

The target monitor is primarily used for runtime monitoring, i.e., to send pertinent trace data to the host about target behavior in real time. Two variants of the runtime monitoring methodology, (a) software and (b) on-chip monitoring are discussed in detail in Sect. 5.4.

5.3.1.4 Visualizing Target Behavior in Real Time

On the host computer, the animation program in the target debugger GUI is started. The XML file generated for the corresponding project is loaded in the target debugger. A debug communication interface (for the communication between target and host) is chosen. The target debugger is now ready for receiving the trace data from the target.

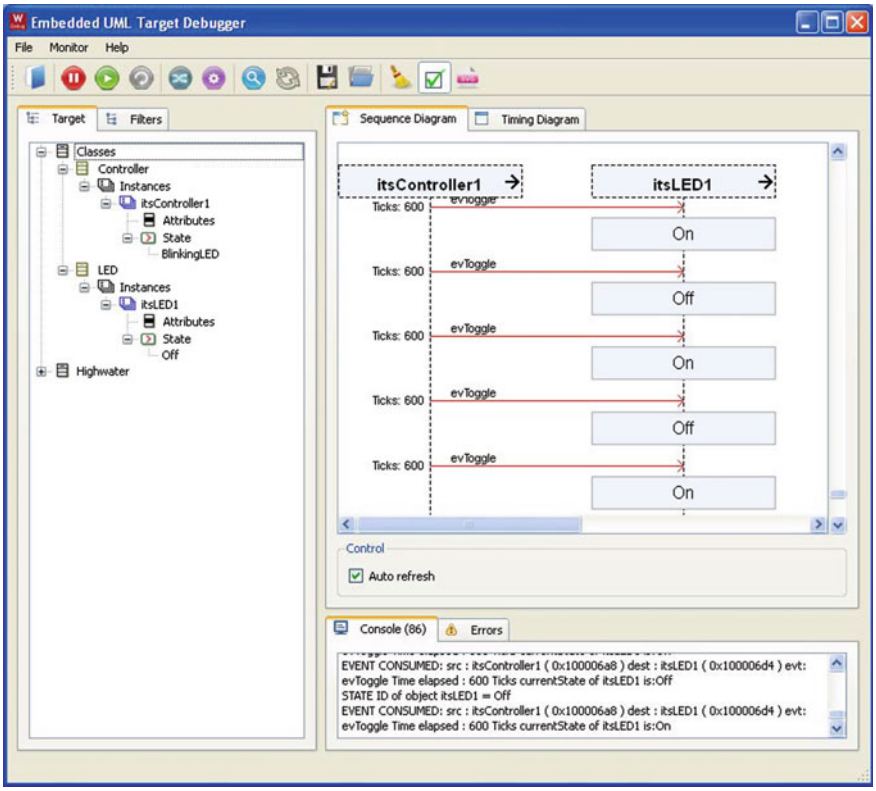


Fig. 5.3 Visualizing target behavior in real time using UML sequence diagrams in the target debugger GUI

Once the system code is deployed in the target, the target monitor (bundled with the RTOS framework, e.g., RXF) starts sending state, event and temporal information pertinent to the behavior of the target. This trace data is sent via the chosen debug communication interface to the host computer.

The animation program in the target debugger receives and decodes this trace information with the aid of the XML file. It reconstructs the target behavior on the host computer, at the design level, using UML sequence diagrams with time stamps and timing diagrams (Figs. 5.3 and 5.4). Thus, with the aid of this approach the target behavior can be visualized and debugged in real time at the host computer.

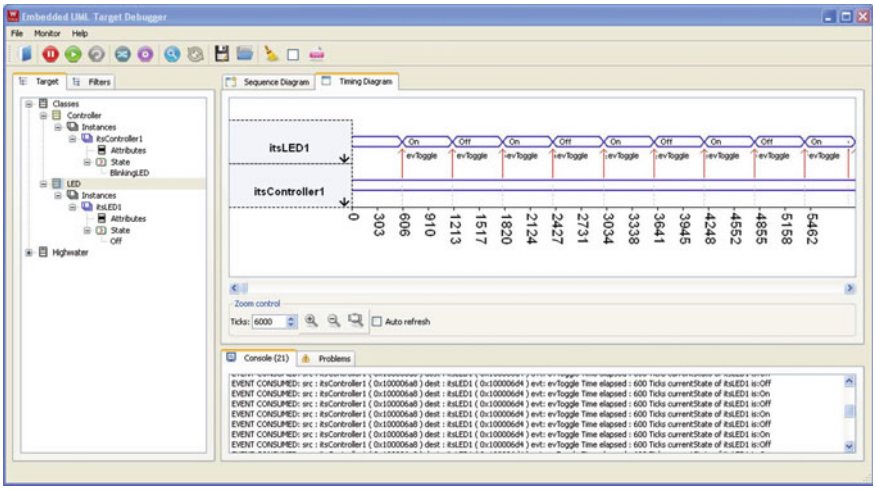


Fig. 5.4 Visualizing target behavior in real time using UML timing diagrams in the target debugger GUI

5.4 Runtime Monitoring

To perceive a time- and memory size-aware runtime monitoring approach, this section begins with a brief outline on the classification of runtime monitoring approaches and a discussion on their pros and cons. The requirements for a time-and memory-aware monitoring approach, which is capable of sufficiently observing the target behavior in real time, are outlined. Based on the trace data generated by the runtime monitoring approach, the target behavior is visualized/debugged using UML interaction diagrams on the host side. The requirements are discussed in the context of two variants of the runtime monitoring mechanism proposed in this paper, namely, (a) generic software-based and (b) on-chip monitoring.

5.4.1 Classification of Runtime Monitoring

Monitoring can be coarsely defined as the use of probes for producing data traces to help the developer/tester gain insight into the origins of misbehavior in the system under test (SUT). Based on the type of a probe used, runtime monitoring is classified into software, hardware, hybrid, and on-chip monitoring. In software monitoring, additional code is added to the target software to obtain the trace data. This is also termed as software instrumentation. In hardware monitoring a dedicated monitoring hardware is attached to the target system for obtaining the trace data. The use of a combination of additional software and hardware to monitor a target is termed

as hybrid monitoring. On-chip monitoring refers to the use of additional built-in, on-chip debugging hardware incorporated in the target to obtain the trace data.

5.4.1.1 Pros and Cons

A drawback of software monitoring is the overhead incurred by executing the additional code, at the target, to obtain the trace data. This incurs memory and time overhead in the target. Interference with the target system's normal operation may arise if the execution of the target software is delayed because of the time spent in the monitoring code. For example, model-based tools, such as [15, 16], make use of software monitoring by instrumentation of the source code to debug, visualize target behavior (using UML diagrams) or execute test cases. Since the debug code is downloaded on the target, the instrumentation overhead is significant. To gain a deeper understanding, consider a simple experiment comprising of application scenarios with 2, 4, 6, and 8 classes in the design model. To visualize the behavior of the target, the instrumented code generated for these examples in [14] shows an increase of 150–250%[1] of source code in comparison with the respective application code size. Clearly, such approaches are not suitable for resource constrained embedded systems. Hence, when using a software monitoring mechanism, it is imperative to minimize the monitoring overhead (e.g., [8]).

The significant advantage of hardware monitoring arises from the nonintrusion benefits obtained by using additional hardware [31]. However, scalability of hardware monitors is affected with respect to monitoring more complex systems [32]. On the other hand, hybrid monitoring makes use of advantages of each approach (i.e., software and hardware monitoring) while at the same time attempts to mitigate their disadvantages.

The key advantage for on-chip monitoring is the presence of an on-chip trace unit, which provides watch points, data tracing, and system profiling for the processor [4]. This can be treated as the major enabling technology, in the future, for target monitoring and testing. A prerequisite is that the underlying processor in the embedded system should support this feature. Whereas, the trace data obtained from the on-chip trace units is in a standardized format (e.g., Manchester encoding) a disadvantage, at this juncture, is the lack of open source/standard tools for communicating the real-time trace data. For example, the real-time trace data from the microcontroller (e.g., MCB1700 evaluation board with Cortex-M3) can be sent to the host only by using proprietary tools (e.g., ULINKpro [6] for Cortex-M3 [4] architecture).

[1] Obtained by measurement.

5.4.2 *Time-and Memory-Aware Runtime Monitoring Approaches*

In the case of embedded systems, the main factor influencing the usage of a monitoring mechanism is the monitoring overhead [32]. Toward this direction, this section focuses on proposing a time-and memory-aware monitoring approach for debugging and visualizing the target behavior in real time. In this context, the requirements of the two variants of runtime monitoring, relating to the main scope of this paper, are discussed. The two variants are the (a) generic software-based and (b) on-chip (software) monitoring methodology for debugging/visualizing the target behavior on the host side.

5.4.2.1 Software Monitoring

For a software-based monitoring approach to be applicable for deeply embedded systems, with minimal or ideally no overhead, it is imperative that it satisfies the following constraints:

- Generic monitoring routine i.e., independent of the application (size, complexity)
- Minimally intrusive runtime monitoring routine (e.g., a few bytes of memory)
- Modular software monitoring approach, independent of the debug communication interface used
- Minimizing communication overhead between the monitoring routine and the application (e.g., target debugger), which is decoding and interpreting the trace data at the host

With a minimal, generic monitoring routine and bounded, measurable/predetermined overhead (memory, time), the software-based target monitor can be delivered along with the final production code. Now, the additional monitoring overhead (memory, time), which is known beforehand, can be accommodated during the system design phase. This can be achieved by allocating additional resources (memory) or adjusting the scheduling properties (time).

5.4.2.2 On-Chip (Software) Monitoring

Another alternative of runtime monitoring is the on-chip monitoring methodology. However, open source standards for communicating the real-time trace data from the microcontroller to the host, i.e., accessing real-time trace data without the use of proprietary debug adaptors (e.g., ULINKpro [6] for Cortex-M3 [4]) is currently unavailable. On the other hand, on-chip monitoring can be treated as a major enabling technology for the future in the context of minimally intrusive debugging/testing of embedded systems. Hence, the on-chip monitoring approach is chosen as an

alternative for visualizing the target behavior in real time, using the model-based debugging approach proposed in this paper.

However, at this juncture, in order to insert the test stimuli/input data to the embedded system and receive the test results using the real-time trace functionality, without proprietary tools, additional hardware, and/or software components are necessitated. In this paper, such an approach (of adding additional software components) is followed in the experimental evaluation for debugging using on-chip monitoring (Sect. 5.5). Hence, this approach is denominated as *on-chip (software)* monitoring in the following section. On the other hand, to apply a memory and time-aware runtime monitoring methodology, such an additional software component, if included, should be a generic monitoring routine with minimal, bounded, and measurable overhead parameters.

An example of on-chip monitoring is available in the recently introduced Cortex-M3 processor/architecture (e.g., used in an evaluation board [27]) that supports real-time tracing using a built-in debug unit called Data Watchpoint and Trace (DWT) and Debug Access Port (DAP). However, to inject the test stimuli from the host computer (e.g., using the test framework approach) additional hardware and/or software components need to be developed.

5.5 Experimental Evaluation

An experimental evaluation based on the two monitoring approaches described above is presented in this section.

5.5.1 Software Monitoring

A prototype implementation of the proposed software-based runtime monitoring approach is described in this section. The aforementioned generic, software-based target monitor can be implemented in programming languages such as C and bundled with the RTOS framework in the embedded system. The target monitor routine is then either invoked by the host or the RTOS (and the generated code from the model) to send and receive debug data respectively. For example, the target monitor routine is invoked by the host (e.g., by the target debugger in the proposed model-based debugging approach) to inject the debug stimuli, in the form of events to the target, i.e., `host` $\xrightarrow{input}$ `target monitor`.

Similarly, the target monitor is invoked whenever an event is consumed at the target, to send an event-consumed notification to the host, i.e., `target monitor` $\xrightarrow{result}$ `host`. This trace data is then reconstructed at the host by the target debugger, as UML sequence/timing diagrams to visualize the target behavior in real time.

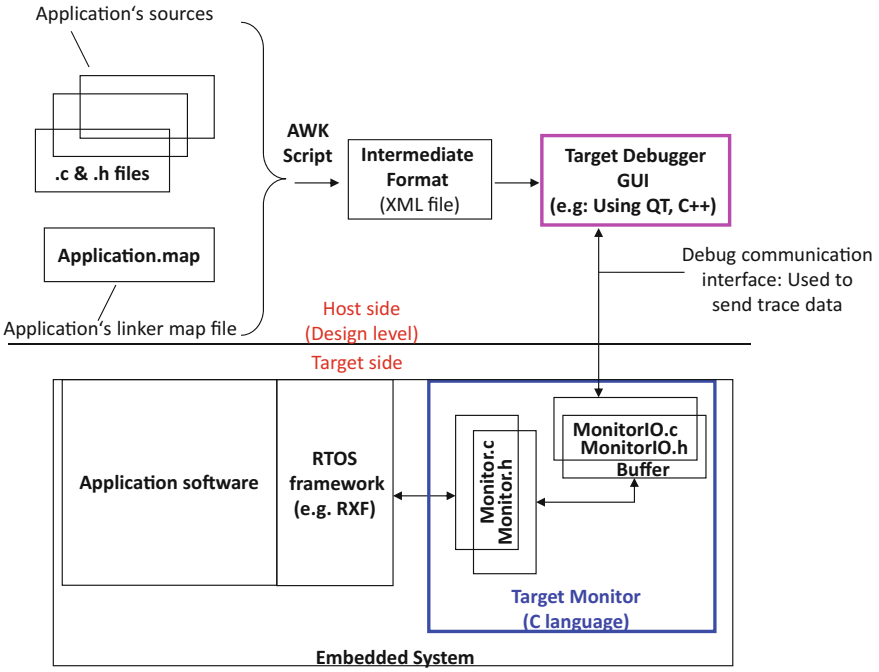


Fig. 5.5 Software-based runtime monitoring at the target side and XML file creation at the host side

Thus, the generic target monitor routine in the prototype (Fig. 5.5.) comprises of two main functionalities, namely, (a) communicating with the RTOS framework and (b) communicating with the host.

The target monitor implementation is modularized based on these two functionalities in *Monitor.h/.c* and *MonitorIO.h/.c* routines respectively. The RTOS framework used in the prototype (OORTX-RXF [33]) comprises of a scheduler that handles the events. The target monitor functionality is used at this point by invoking (a *send* function in) *Monitor.c* in the RTOS framework for consumed events. The module *Monitor.c* in turn uses the functions in *MonitorIO.c* to send and receive data between the host computer and the embedded system via a debug communication interface (Fig. 5.5). *MonitorIO.h/.c* is configurable and implemented based on the APIs and functionalities available in a given debug communication interface (e.g., EIA-232 [1] or JTAG-based). The monitor implementation uses a configurable buffer to handle the trace data.

For example, when an event is processed and dispatched to its respective receiver in the embedded system, *Monitor_sendEvent(unsigned int* pEventData)* function in *Monitor.c* is used to notify the host about the event consumption at the embedded system. This in turn invokes the respective function in *MonitorIO.c* to send the trace data to the host computer, which is decoded by the target debugger in the host

computer. The target monitor prototype using a RS-232 debug interface requires a total memory size of approximately 1 Kbyte (1061 bytes ROM plus 135 bytes RAM). A comparison of the target monitor implementation for various debug interfaces and their experimental evaluation is described in Sect. 5.5.

5.5.1.1 Target Debugger

A prototype of the target debugger comprises of a decoding and animation program. The target debugger is implemented in the programming language C++ using the user interface framework Qt [30]. The target debugger decodes and interprets the trace data which is sent via a debug communication interface as seen in Fig. 5.5.

5.5.1.2 XML File Creation

During the compilation of the (application) system code, an AWK script parses the linker map file, source files, header files (of the application), and creates a symbol table data. This is stored in an intermediary format such as an XML file as seen in Fig. 5.5. The decoding program at the host makes use of this XML file (Fig. 5.5) to decode and interpret the incoming trace data which is sent via a debug communication interface as seen in Fig. 5.5. Significant overhead involved in sending trace data back and forth between the host and the target system is avoided by the use of the XML file at the host and predefined frame format for notifications [18].

5.5.1.3 Predefined Frame Format for Notifications

The predefined frame format for notifications, i.e., debug-input data (e.g., inject event) from the host computer and the debug results (e.g., event consumed notification) to the host computer are shown in Fig. 5.6.

The predefined frame format is based on the following design considerations, namely, (a) compactness, (b) minimum number of operations on the target and (c) extensibility. The frame format in the prototype implementation is shown in Fig. 5.6. The "length" field is mandatory, one byte in length and indicates the length of the parameters. The mandatory "command_id" field is also one byte in length. It denotes the command corresponding to the frame sent. The "parameters" field is optional and can be between 0 and 255 bytes in length. It denotes the data about the current command. The minimum length of the monitor frame is two bytes (1 byte each for length and command_id). The frame format for injecting events (from host to target) and the trace data format for sending the event-consumed notification (from target to host) is shown in Fig. 5.6. To inject an event (i.e., debug-input data) the required parameters are the destination of the event, the event to be injected, source of the event, and event parameters (if any). In this frame format, the event, source, and destination values occupy 4 bytes each.

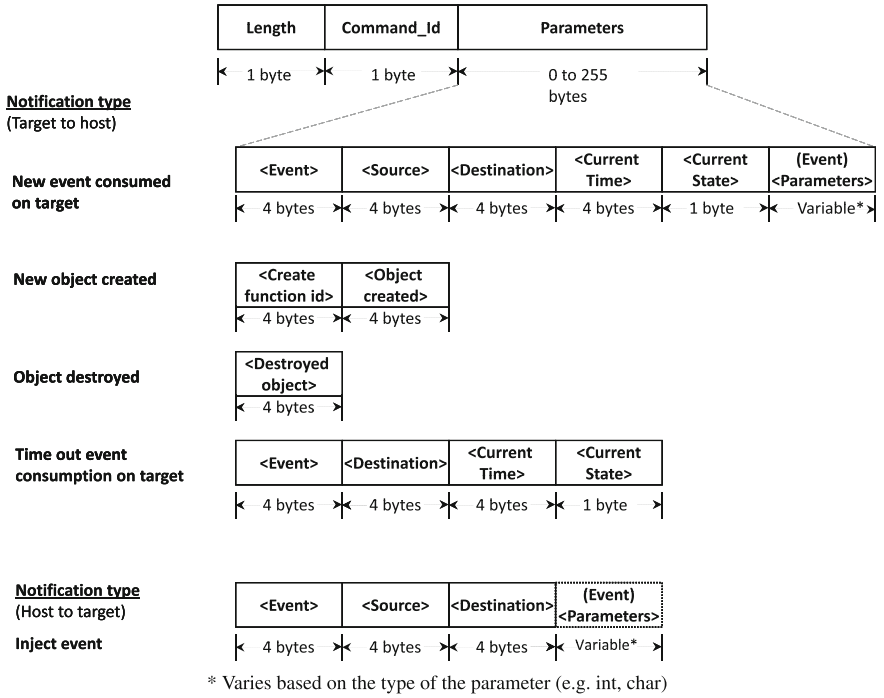


Fig. 5.6 Trace data frame format

For example, to inject an event, *evToggle(int LedNr)* from class *Controller* to class *LED*, to turn on an LED indicated by *LedNr*, i.e., $\texttt{Controller} \xrightarrow{evToggle(LedNr)} LED$, the parameters for the debug-input data in predefined format is *<LED evToggle Controller LedNr>*. However, in the proposed approach, since the object addresses of the above parameters are available at the host computer in the XML file, the debug-input data is *<0X18097098 0X00004374 0X18074897 0X00000001>*. This is received by the *MonitorIO.c/h* routine in the target. The *Monitor.c/h* routine, in turn, decodes, and inserts the corresponding debug-input data to the target. Similarly, the parameters for the debug result for this example in the predefined frame format (Fig. 5.6) are *<evToggle Controller LED CurrentTime ON LedNr>*. In other words, the 21 bytes for the parameters, for the event-consumed notification indicating the debug result, are *<0X00004374 0X18074897 0X18097098 0X00009870 02 0X00000001>*. Note that all events described at the design level, i.e., available in the system code, can be monitored.

Hence by this generic, software-based runtime monitoring methodology, only the debug-input data is injected to the target and the corresponding debug results are obtained as trace data from the target. This implies that the only instrumentation overhead required for debugging in the target is the software-based runtime monitoring overhead. However, note that by optimizing the software monitoring routine

and/or further minimizing the memory requirement, the generic software-based runtime monitor can also be part of the final production code.

5.5.2 On-Chip (Software) Monitoring

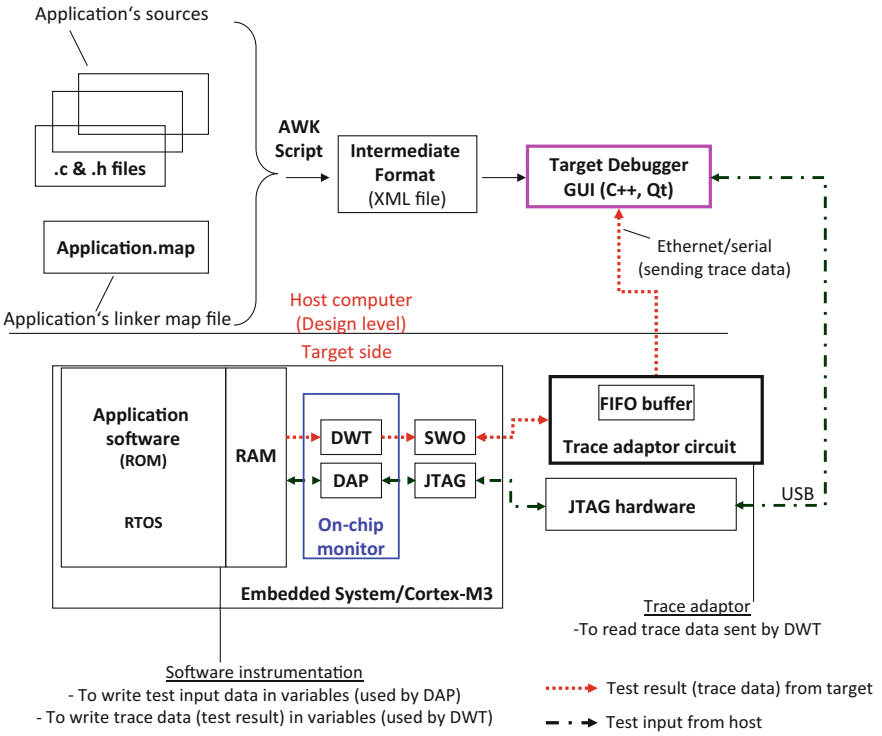
An example of on-chip monitoring is available in the recently introduced Cortex-M3 processor/architecture (e.g., used in an evaluation board [27]) that supports real-time tracing using built-in debug units such as DWT and DAP. The real-time trace functionality in this microcontroller can be used readily with proprietary tools such as μVision [6]. Whereas such proprietary tools cannot be used for inserting the debug stimuli or monitoring the target behavior.

To realize this goal and in order to provide a generic approach toward on-chip monitoring in this paper, a minimal (generic) software monitoring routine is introduced in the target. This can be termed as on-chip monitoring with additional software instrumentation. On the other hand, a trace adaptor (hardware circuit) is necessitated to forward the trace data from the on-chip unit to the host (without the use of any proprietary tools). The aforementioned trace adaptor unit and the generic software instrumentation used in the prototype evaluation are discussed below.

5.5.2.1 Trace Adaptor

Figure 5.7 provides an overall view of the on-chip monitoring arrangement in the prototype. It comprises of a DWT unit that provides support for monitoring data as it is being changed at the target. An additional trace adaptor circuit, with a FIFO buffer, is developed in the prototype. The trace adaptor, as the name implies, is necessary to adapt the trace data from the microcontroller (i.e., serial data stream to UDP data stream) and forward the trace data to the host. This arrangement provides sufficient time to process the data stream (byte-wise) at the FIFO buffer and eventually decode the trace data by the end application at the host computer (i.e., the target debugger in this paper). The DAP debug unit is used to provide support for inserting the input data (debug stimuli) to the embedded system.

There exist two paths for data transfer (Fig. 5.7) between the target and the host (indicated by two different line formattings). Each path is responsible for one functionality, namely injecting the debug input/stimuli to the target and sending the trace data to the host respectively. For example, the debug-input data in the form of events is injected to the embedded system with the aid of the DAP unit, using the JTAG interface. The debug-input data (e.g., regarding an event) comprises of an event, its source, destination and event parameters. The debug data result (i.e., the trace data) from the DWT unit is sent via a Serial Wire Output (SWO) interface, which is part of the on-chip debug unit. This is processed by the trace adaptor circuit for further usage at the host. The trace data indicating the debug results (i.e., an event-consumed



DWT: Data Watchpoint and Trace, DAP: Debug Access Port,
SWO: Serial interface [4],
JTAG: Industry standard JTAG interface [4]

Fig. 5.7 On-chip (software) monitoring arrangement

notification) comprises of an event (consumed on target), its source, destination and event parameters.

5.5.2.2 Software Instrumentation

Software monitoring is necessitated at the target for writing the trace data in data structures, such as predefined (debug) variables of the given application. These variables are monitored by the comparators of the DWT unit. The trace data is sent to the host, when the (debug) variables, monitored by the comparator units, are changed in the target. Thus, the major advantage in the on-chip monitoring approach is that no additional functionality is required to transfer the trace data from the embedded system to the host. Hence, the software instrumentation (in the case of the on-chip monitoring approach) is limited to a few write operation cycles at the target. Then, the trace data, which is now stored in variables/comparators at the DWT unit, is

available at the serial interface (SWO) (Fig. 5.7). Therefore, no additional software routines are required for sending the trace data to the host.

Similarly, to inject the debug-input data (comprising an event, its source, destination, and event parameters) sent from the host, software instrumentation is required. The debug-input data is written to the data structures such as predefined (debug) variables monitored by the DWT comparator units. An inject event flag is set after sending the debug-input data to the target. The RTOS used in the prototype detects that an event has to be injected by polling (during idle cycles) the status of the flag (*injectEvent*) at the target, as seen below.

```
1 if(injectEvent==1){/*check flag, generate event*/
2 EVT_gen(source, destination, eventId);
3 injectEvent=0;/*reset flag*/ }
4
```

Note that *EVT_gen(source, destination, eventId)* is a macro to generate an event in the underlying RTOS framework. Thus, additional software routines to convey the debug-input data from the host or debug results from the target are eliminated by the on-chip debug units such as DWT and DAP in the on-chip (software) runtime monitoring methodology.

5.6 Performance Metrics

Performance metrics such as memory and time overhead of the runtime monitoring mechanisms are discussed here.

5.6.1 Software Monitoring

The monitoring overhead for the software monitoring approach, such as target monitor size, event (bursts) handling, target monitor buffer overflow, time spent in the target monitor routine, and a comparison with the instrumentation overhead in the existing approaches, is presented here.

5.6.1.1 Debug Communication Interface and Target Monitor Size

The software monitoring mechanism introduced in this paper, for debugging, is intended to be independent of the application, its size and complexity. Moreover a modularized implementation of the monitoring routine is proposed in this paper such that the communication of the monitoring routine with the RTOS ((i.e., in *Monitor* routine) and the debug interface (*MonitorIO* routine) are available in two separate routines.

Table 5.1 Memory requirement on target for various debug interfaces using software-based runtime monitoring

Interface	Memory requirement	
	RAM (byte)	ROM (byte)
Generic-EIA-232	112	1290
JTAG-Keil (μVision)	84	1194
JTAG-Lauterbach (Trace32)	84	1396

The implementation of the *MonitorIO* routine is dependent on the debug interface under consideration (e.g., APIs available). In the prototype, the *MonitorIO* routine is implemented for the generic EIA-232 [1] serial interface and two industry standard JTAG-based interfaces, such as Keil-μVision [6] and Lauterbach-Trace32 [25]. The memory requirement for the monitor routine in the prototype for these three debug interfaces is shown in Table 5.1.

From Table 5.1, it is clear that the total memory (RAM, ROM) requirement is approximately 1 KiByte for all the three debug interfaces. Thus, this software-based monitoring routine, which is independent of the application, can also be accommodated in the final production code.

5.6.1.2 Time Spent in the Monitoring Routine

The time spent in the monitoring routine for sending an event-consumed notification obtained by measurement (using a logic analyzer), is shown in Table 5.2. It is clear that the time spent in the monitor routine can be predetermined and is independent of the application and its complexity. The differences in the time spent in the monitor routine for various debug interfaces is based on the implementation and the functionality supported by the APIs for the various debug interfaces [19]. Thus, by the proposed software monitoring technique, the memory (approx. 1 KiByte) and time overhead (in the order of μs), known beforehand, can be accommodated in the earlier phases of the development cycle.

To summarize, target behavior can be visualized online (at the host side), in resource constrained embedded systems without downloading any debug/test harness on the embedded system, using the proposed software-based runtime monitoring mechanism. This is a significant advantage over the existing approaches. The only

Table 5.2 Time spent in software-based target monitor for sending an event consumed notification

Interface	Time in monitor (μs)
EIA-232	74.52
μVision	265
Trace32	16.5

Table 5.3 Number of events handled per interface

Debug interface	Events per second
EIA-232	556
μVision	3770
Trace 32	3268

space requirement in the target is the memory requirement of the software-based runtime monitoring routine.

5.6.1.3 Event (Bursts) Handling

Events consumed at a higher frequency in a short period of time by the target can be termed as event bursts. Since the target monitor implementation depends on the debug interface used, the number of events (and event bursts) the target monitor can handle, also depends significantly on the debug interface under consideration. However, in order to handle the event bursts, the target monitor is implemented with a send/receive buffer interface (Fig. 5.5). The applicability of the target monitor buffer and its dimensioning is again dependent on the debug interface used.

The values shown in Table 5.3 provide a comparison of the number of events that each debug interface can handle theoretically before the use of target monitor send/receive buffers (in our prototype implementation). For example, for the EIA-232 interface, the theoretical maximum number of events that it can handle per second is 556 (115200 [baud rate]/9 [8bit+stop bit]/23 [number of bytes per event-consumed notification]). However, when there is a burst mode in the target system (i.e., number of events per second higher than the theoretical estimation in Table 5.3), this can be handled with the use of a target monitor buffer. Handling of burst-mode data can also be taken over by the APIs provided by the debug interface used. Thus, when there is a consistent burst of events, appropriate dimensioning of the target monitor buffer size and/or selection of a debug interface by the end user is necessary.

5.6.1.4 Target Monitor Buffer Overflow and Real-Time Characteristics of the Target

In the prototype, the target monitor is handled as a lower priority task in comparison with the system tasks. This implies that the target monitor is invoked during the "idle" state of the main loop of the RTOS framework. Let us consider a burst-mode scenario, in which there is a possibility that the target monitor buffer overflows. The target monitor buffer is implemented as a ring buffer. This implies that whenever there is a buffer overflow, the data in the ring buffer could be overwritten. This can lead to a loss of data (notifications about target behavior) stored in the monitor buffer. This is also because of the fact that the target monitor is assigned as a lower priority task and can access the system resources once they are freed by the higher priority

(system) tasks. In this case, whenever the target monitor buffer is full, the target debugger is notified about the possible loss of data.

For this scenario there are two possible configuration options, whereby the end user has to compromise between target monitor buffer size and the influence on real-time characteristics of the embedded target. For instance, since the buffer size is configurable, it is up to the end user to allocate a smaller/larger buffer size. As the target monitor implementation is dependent on the debug interface used, the buffer size and its usage is also dependent on the debug interface under consideration. On the other hand, the user could also assign the target monitor as a higher priority task. However, when the end user gives a higher priority to the target monitor (and/or increases the buffer size), he has to compromise between the influence on the real-time characteristics and the loss of target behavior data/notifications.

5.6.1.5 Traditional Versus Proposed Approach–Memory Overhead

The proposed approach has been evaluated for four example scenarios. Similarly, the existing model-based debugging feature (live animation) in the MDD tool [14] was applied to the same evaluation scenarios. Note that the four application scenarios consist of increasing system code (size) and complexity.

For instance, application scenarios 1, 2, and 3 consist of 2, 4, and 8 classes respectively (based on the small "blinky" example [20]). Scenario 4 is based on a more sophisticated case study involving a MIDI system (15 classes). Detailed description of the MIDI system evaluation for the proposed approach is available in [18, 20]. The complexity of the system also varies based on the number of events handled and dependencies on other modules.

The memory overhead incurred (in the target) using both the approaches for the four scenarios (for model-based debugging) are shown in Fig. 5.8. From Fig. 5.8, it is evident that the memory overhead increases with an increase in the application size using the model-based (live animation feature) approach in an MDD tool such as Rhapsody [14]. On the other hand, the size (and the percentage increase) of the target monitor memory footprint is negligible in comparison with the increasing application size as seen in Fig. 5.8 for our proposed approach.

5.6.2 On-Chip (Software) Monitoring

In this case, the only monitoring overhead (time & memory) is that of the additional software instrumentation used to write the test input stimuli/trace data in the debug variables (of the application) monitored by the comparators in the DWT. The additional memory required for the software instrumentation in this approach is approximately 100 bytes (Table 5.4). The time taken to write the trace data for an event-consumed notification (with 23 bytes of trace data denoting the test result), is 360 ns (obtained by measurement using a logic analyzer). The on-chip

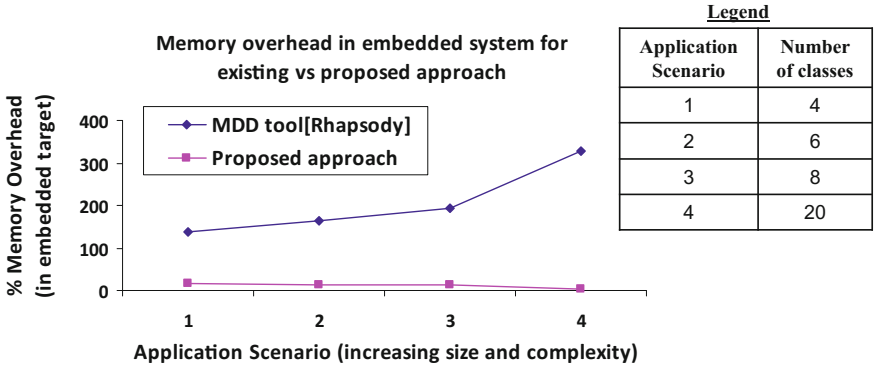


Fig. 5.8 Memory overhead (in target) for model-based debugging of various application scenarios

Table 5.4 Memory requirement on target for software instrumentation

RAM (byte)	ROM (byte)
24	74

monitoring mechanism is also independent of the application and its complexity. Thus, the overhead parameters for executing the test cases using this monitoring methodology (memory approx. 100 bytes and time = 360 ns) is also known and measurable beforehand.

5.7 Discussion and Evaluation

Debugging RTESS is a challenging task in comparison with debugging desktop systems. While there are some traditional and model-based tools for debugging RTESS, these have limitations. Model-based debugging techniques in the existing tools usually involve dynamic source code instrumentation. This instrumented code increases with the increase in the application size and necessitates sophisticated debug interfaces. The protocol and performance overhead incurred during debugging could also result in modifying the temporal behavior of the embedded system. All these factors make the existing model-based debugging techniques unsuitable for the memory-size constrained RTESS.

In order to overcome the aforementioned limitations, a model-based debugging methodology for small RTESS was outlined in this paper. Using the proposed methodology, RTESS behavior can be visualized in real time using UML sequence and timing diagrams. Some salient features in the proposed approach, which overcome the limitations of the existing approaches, are discussed below.

5.7.1 Salient Features in the Proposed Approach

- Dynamic source code instrumentation is eliminated with the introduction of an optimized monitoring software routine in the target monitor (implemented in the programming language 'C'). The target monitor (library) is bundled with the RTOS used (RXF [33]). The target monitor is now independent of the application, its size, complexity, and source code. The target monitor occupies approximately 1 KiByte of memory, which is accommodative for small embedded platforms. Moreover, because of its size, the target monitor can be bundled along the final production code as well.
- In addition to the optimized target monitor size, the information exchange between the target debugger and the target monitor is handled via a custom-defined protocol. The protocol design is extensible, compact, and requires a minimum number of operations. For example, the minimum frame-size for the protocol is 2 bytes and an event consumed notification requires 23 bytes of data. The frame format of this protocol is described in detail in Sect. 5.5.
- A major factor influencing the real-time behavior of the embedded system is the communication overhead between the target system and the host computer (i.e., huge debug data being sent back and forth between the target and the host computer). In order to minimize this, an AWK script parses the source files, header files, and linker map file (for a given project) and creates a symbol table at the host computer in our approach (Figs. 5.5 and 5.6). This symbol table enables identifying each element in the system code, such as class, instance, event, etc., by an ID. This symbol table data is stored in an intermediary format, such as an XML file, at the host computer. On receiving the trace data from the target, the target debugger decodes the trace data with the aid of this XML file. The trace data from the target is translated and the animation program in the target debugger GUI re-constructs the target behavior in the form of UML sequence and timing diagrams in real time.
- Based on the runtime monitoring mechanisms discussed above, it can be stated that the proposed techniques are time-and-memory aware (supported by the performance metrics discussed in Sect. 5.6). This is primarily because, the overhead (time & memory) introduced by the two variants of the monitoring technique is measurable beforehand, minimal, bounded, and independent of the application. On the other hand, tools such as [14] introduce unbounded and variable overhead for debugging and/or visualizing the target behavior (using UML diagrams) in real time for different application scenarios. These features eliminate the risks due to the change in program behavior before and/or after debugging the system code. Therefore, the proposed time-& memory-aware runtime monitoring mechanisms provide a significant improvement over the existing techniques.
- The proposed mechanisms also address the aspect of scalability and applicability for resource constrained targets and industrially relevant examples. For example, the solution proposed in this paper already concentrates on resource constrained embedded systems, thereby addressing the question of scalability. For industrial applications involving several complex interactions and entities, the overhead

parameters can be accommodated in the earlier phases of the development cycle. A robust implementation of the software instrumentation may further reduce the overhead parameters.

5.8 Conclusion

While monitoring/testing an embedded system and acquiring the trace data, one often faces the so-called "Heisenberg's effect": *Inspecting a system tends to influence the system's behavior* [10]. Whereas a time & memory-aware runtime monitoring mechanism is introduced in this paper, a runtime monitoring mechanism which introduces (ideally) no overhead in the target is an ambitious goal. Therefore, while employing a generic software-based runtime monitoring approach, the goal should be to minimize the monitoring overhead as far as possible. An example of this approach is discussed with a prototype in this paper. For embedded systems with microcontrollers supporting the real-time trace functionality [10] (i.e., on-chip debug units), the overhead parameters can be further minimized using an on-chip mechanism such as the one introduced in this paper. When the nature of the embedded software described requires the system to meet real time requirements in debugging/testing mode, the overhead parameters from monitoring can be included in the earlier stages of the development cycle. By doing so, the influence on the real-time characteristics of the embedded system because of the overhead introduced by monitoring can be eliminated.

Application of the proposed time-& memory-aware runtime monitoring, to industrial case studies, adding UML state chart diagrams in the target debugger GUI for visualizing the target behavior and evaluation on other target platforms are some items for future work.

Acknowledgements This work was supported by a grant from BMWi-ZIM, DAAD, and industrial partner Willert Software Tools GmbH. We would like to thank the project teammates at Willert Software Tools GmbH and UAS-Osnabrueck for their cooperation.

References

1. Axelson J (2007) Serial port complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems, 2nd edn. Lakeview Research
2. BridgePoint UML Tool (2016) http://www.mentor.com/
3. Bunse C, Gross H-G, Peper C (2007) Applying a model-based approach for embedded system development. In: 33rd EUROMICRO conference on software engineering and advanced applications
4. Cortex-M3 Processor (2016) http://www.arm.com/
5. Delgado N, Gates AQ, Roach S (2004) A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans Softw Eng 30(12):859–872
6. Embedded development tools (2016) http://www.keil.com/

7. Enterprise Architect tool (2016) http://www.sparxsystems.com/
8. Fischmeister S, Lam P (2010) Time-aware instrumentation of embedded software. IEEE Trans Ind Inform 6(4):652–663
9. France RB, Ghosh S, Dinh-Trong T, Solberg A (2006) Model-driven development using UML 2.0: promises and pitfalls. Computer 39(2):59–66
10. Ganssle J (2008) The art of designing embedded systems, 2nd edn. Newnes
11. Graf P, Muller-Glaser KD, Reichmann C (2007) Nonintrusive black- and white-box testing of embedded systems software against UML Models. In: Proceedings of the 18th IEEE/IFIP international workshop on rapid system prototyping, pp 130–138, Washington, DC, USA, 2007. IEEE Computer Society
12. Harmon T, Klefstad R (2007) Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In: 13th IEEE international conference on embedded and real-time computing systems and applications, RTCSA 2007, pp 209–216
13. Huang X, Seyster J, Callanan S, Dixit K, Grosu Radu, Smolka Scott A, Stoller Scott D, Zadok Erez (2012) Software monitoring with controllable overhead. Int J Softw Tools Technol Transf 14(3):327–347
14. IBM Rational Rhapsody Developer, Ver 8.2 (2016) http://www.ibm.com
15. IBM Rational Rhapsody Test Conductor Add-on (2016) http://www.btces.de/
16. IBM Rational Test RealTime (2016) http://www-01.ibm.com/software/awdtools/test/realtime/
17. Iyenghar P, Wuebbelmann J, Westerkamp C, Pulvermueller E (2013) Model-based test case generation by reusing models from runtime monitoring of deeply embedded systems. IEEE Embedded Syst Lett 5(3):38–41
18. Iyenghar P (2012) A test framework for executing model-based testing in embedded systems. PhD thesis, University of Osnabrueck
19. Iyenghar P, Pulvermueller E, Westerkamp C, Uelschen M, Wuebbelmann J (2011) Model-based debugging of embedded software systems. Gesellschaft Informatik (GI), softwaretechnik (SWT), pp 31–33
20. Iyenghar P, Westerkamp C, Wuebbelmann J, Pulvermueller E (2010) A model based approach for debugging embedded systems in real-time. In: Proceedings of the tenth ACM international conference on Embedded software, EMSOFT '10, NY, USA
21. Jiao Y, Zhu K, Yu Q, Wu B (2006) Towards model-driven methodology: a novel testing approach for collaborative embedded system design. In: 10th International conference on computer supported cooperative work in design, 2006. CSCWD '06, pp 1–5
22. Karsai G, Sztipanovits J, Ledeczi A, Bapty T (2003) Model-integrated development of embedded software. Proc IEEE 91(1):145–164
23. Kashif H, Mostafa M, Shokry H, Hammad S (2009) Model-based embedded software development flow. In: 4th International design and test workshop (IDT), pp 1–4
24. LabVIEW System Design Software (2016) http://www.ni.com/labview/
25. Lauterbach-Microprocessor development tools (2016) http://www.lauterbach.com/
26. Matlab and Simulink (2016) http://www.mathworks.com/
27. MCB1700 evaluation board (2016) http://www.keil.com/mcb1700/
28. Object Management Group (2016) http://www.omg.org
29. Plattner B (1984) Real-time execution monitoring. IEEE Trans Softw Eng SE-10(6):756–764
30. Qt. User interface framework (2016) http://qt.nokia.com/
31. Tsai JJP, Fang K-Y, Chen H-Y, Bi Y-D (1990) A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Trans Softw Eng 16(8):897–916
32. Watterson C, Heffernan D (2007) Runtime verification and monitoring of embedded systems. IET Softw 1(5):172–179
33. Willert Software Tools GmbH (2010) http://www.willert.de/