MDebugger: A Model-level Debugger for UML-RT

Mojtaba Bagherzadeh
Queen's University
Kingston, Canada
mojtaba@cs.queensu.ca

Nicolas Hili
Queen's University
Kingston, Canada
hili@cs.queensu.ca

David Seekatz
Queen's University
Kingston, Canada
seekatz@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Canada
dingel@cs.queensu.ca

EXTENDED ABSTRACT

Ideally, debuggers for Model-Driven Development (MDD) tools would allow users to 'stay at the model-level' and would not require them to refer to the generated source code or figure out how the code generator works. Existing approaches to model-level debugging do not satisfy this requirement and are unnecessarily complex and platform-specific due to their dependency on program debuggers. We introduced a novel approach to model-level debugging that formulates debugging services at model-level and implements them using model transformation. This approach is implemented in MDebugger, a platform-independent model-level debugger using Papyrus-RT, an MDD tool for the modeling language UML-RT.

https://youtu.be/L0JDn8eczwQ

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software testing and debugging**;

KEYWORDS

Model-based Debugging, Model-driven Development, UML-RT, Real-time and Embedded systems, MDD, MDE

ACM Reference Format:
Mojtaba Bagherzadeh, Nicolas Hili, David Seekatz, and Juergen Dingel. 2018. MDebugger: A Model-level Debugger for UML-RT. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183473

1 INTRODUCTION

Motivation. Existing Model-Driven Development (MDD) tools do not provide proper support for debugging system behavior at model-level. Instead, developers must typically refer to the generated source code and use a program debugger to debug their applications. Using generated source code for debugging contradicts MDD principles and goals because many of the benefits of the abstraction are lost and new unnecessary accidental complexity is introduced and understanding the generated code can be time-consuming and error prone, especially for developers unfamiliar with code generation or the target language, i.e., the programming language the generated code is written in.

State-of-the-art. Most of the existing work on model-level debugging in the literature (e.g., [4, 5, 7, 8, 10]) relies on program debugging techniques and maps debugging services and their results from code-level to model-level. This approach has the following downsides: the most important one is related to the need for integrating the model-level debugger with different program debuggers, if different target languages are to be supported. E.g., at least three different integrations with three different program debuggers are required for supporting three different target languages (e.g., C++, Java, Python). Besides, in order to map the debugging services from code- to the model-level, instrumentation is applied at code-level rather than model-level which introduces a dependency on the MDD tool and the generated code. This dependency causes portability issues and decreases the reusability of the instrumentation. Finally, the use of a program debugger for model debugging also exposes the 'semantic gap' between the modeling language and the target language. Typically, this gap is substantial and means that some modeling concepts cannot easily be mapped to corresponding programming language concepts and vice versa, and that the translation cannot be 'hidden' [19]. E.g., according to the UML-RT execution semantics, a capsule (i.e., component) instance is assigned to its own logical thread at model-level (i.e., it has its own independent flow of execution) and multiple logical threads can be assigned to a single physical thread in the target language. However, since the notion of logical threads is typically not supported by the program debugger, debugging capsule instances using program debuggers becomes unnecessarily complicated.

Proposed Solution. To overcome the existing limitations, we proposed a novel approach for realizing a platform-independent model-level debugger which does not depend on any architecture-specific program debugger [1, 2]. We use model transformation [11] to instrument a model to be debugged with information that allows it to support debugging activities. As a result of model instrumentation, the generated code from the instrumented model is a debuggable program which can provide debugging services. We developed a model level debugger that interacts with debuggable programs and provides debugging services at model level.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183473

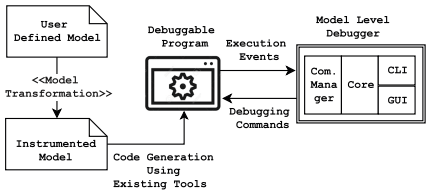


Figure 1: Overview of our Approach to Model Level Debugging (MDebugger)

2 UML FOR REAL-TIME

UML for Real-Time (UML-RT) [15, 17] is a modeling language for designing Real-Time Embedded (RTE) systems with soft real-time constraints. It has been the basis of a lot of academic work, industrial projects, and successful tools (e.g., IBM RSA-RTE [9] and Papyrus-RT [6]). UML-RT is a subset of UML with dedicated concepts for RTE design and only provides two diagrams: *capsule* and *state machine* diagrams. Both diagrams are specializations of the UML composite structure diagram and the UML state machine diagram respectively. UML-RT state machines are simplified versions of UML state machines, e.g., there is no AND-state (orthogonal) regions in UML-RT state machines.

In UML-RT, a system is designed as a set of interacting capsules. A capsule is an active object which has *attributes*, *operations*, *ports*, and a *behaviour* modelled by means of a hierarchical state machine [17].

Eclipse Papyrus for Real-Time (Papyrus-RT). Papyrus is an open source modelling tooling framework for UML which provides advanced facilities for making customized modelling tools. It has been customized for different UML profiles and purposes (e.g., SysML, MARTE, and information modeling) [3] and has recently attracted considerable attention in the modelling communities [12]. Papyrus-RT [6] is built on top of Papyrus and provides modeling facilities based on UML-RT and supports code generation in C++ from the UML-RT models. Also, it provides a run-time service library.

3 MDEBUGGER

3.1 Overview

Fig. 1 outlines our approach to model-level debugging as implemented in MDebugger. The approach consists of two steps:

Creation of Debuggable Programs. Rather than using program debuggers and other related techniques, MDebugger uses the debuggable program that is generated from the instrumented model. As shown in Fig. 1, we use Model-to-Model (M2M) transformation techniques for creating an instrumented version of a user-defined model. The program generated from the instrumented model therefore provides the debugging services by itself as follows: (1) *Execution control* service to control the execution of a system being debugged using commands for setting breakpoints, suspending, resuming, stepping in the execution, and so on; (2) *View and change attributes* services to inspect and modify the system status; (3) *Generation of execution traces* service that provides a foundation to backtrace and analyze the execution; (4) *Debugging interface* to enable model-level debuggers to interact with the system being debugged. The instrumentation process has been developed using the Epsilon Object Language (EOL) [14].

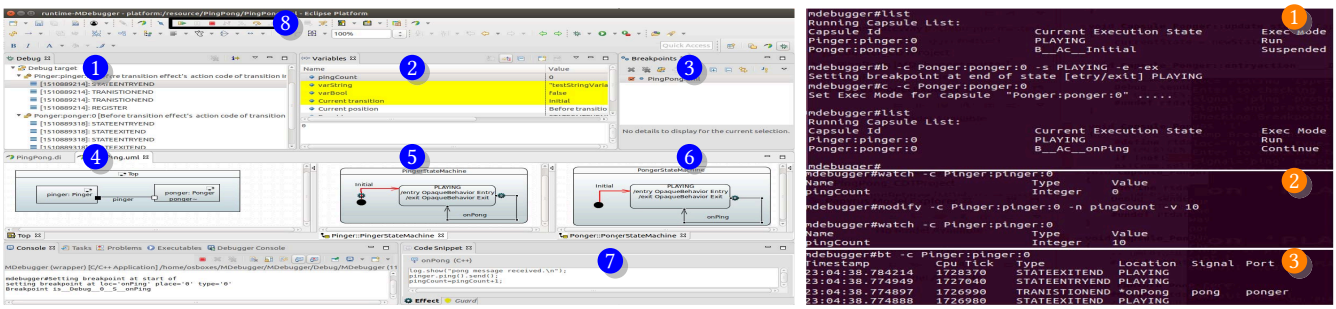
Model Level Debugging using Debuggable Program. By relying on debugging services provided by the debuggable program, we designed and developed a model-level debugger that interacts with the debuggable program and provides model-level debugging support without using a program debugger. As shown in Fig. 1, it consists of several components including a *core component*, a *communication manager*, and command-line and graphical user interfaces (i.e., *CLI & GUI*). The *core component* implements the main logic of the debugger. Initially, it queries the list of the system capsule instances from the debuggable program. For each capsule instance, it keeps track of certain data, such as the most recent events generated by the instance and the active state or transition. The *communication manager* manages the communication with the debuggable program. Both CLI and GUI provide interfaces for debugging using the command-line or a graphical interface. The core of the model-level debugger and the CLI has been developed using C++ and the Boost library. Both CLI and GUI interfaces are interchangeable and one can either choose to use command lines only, or choose to use the graphical interface that has been implemented as an Eclipse perspective an integrated into Papyrus-RT.

3.2 MDebugger Features

Similar to program debuggers (e.g., GDB [18]), MDebugger supports debugging features such as setting breakpoints, listing and modifying variables. However, in contrast with program debuggers which usually work on the code-level and allow for debugging the execution at thread-level, MDebugger works on the model-level and allows for debugging the execution at the capsule-level, i.e., it allows for viewing, controlling, or modifying the execution of specific capsule instances. In the following, we discuss features we have implemented to debug a system developed in UML-RT using our approach along with the limitations of alternative approaches, such as relying on existing program debuggers. To do that, we first introduce an illustrative example. Then, we detail each part of our implementation illustrated in Fig. 2. Graphical symbols (e.g., 1, 2) are provided to show to which parts of Fig. 2a or Fig. 2b the description refers to.

An Illustrative Example. 4 shows the structure of a `PingPing` system in a UML-RT capsule diagram. The system includes two capsule instances `pinger` and `ponger` that interact using ports typed with a `PingPong` protocol defining two messages `ping` and `pong`. The behaviour of the `pinger` capsule instance is shown in 5 which first takes the `initial` transition, sets an integer variable `pingCount` to one, sends a `ping` message to the `ponger` capsule instance, and enters the `PLAYING` state. In the `PLAYING` state, it waits for a `pong` message. Upon reception of the `pong` messages, the `onPong` transition is taken which sends another `ping` message to the `ponger` and increases the `pingCount` (7). The behaviour of the `ponger` capsule instance is similar to the `pinger` instance (6). It first takes the `initial` transition and enters the `PLAYING` state. In the `PLAYING` state, upon reception of a `ping` message, the `onPing` transition is taken which sends a `pong` message to the `pinger` during the transition.

View running capsules Usually, viewing the current state of a program and its components at runtime is the preliminary step to start debugging. MDebugger provides the `list` command (1) that



(a) MDebugger GUI

(b) Excerpts of MDebugger CLI

Figure 2: An Overview of MDebugger Features

Figure 3: Excerpts of GDB

shows the list of running capsule instances, their current execution state that corresponds to the last executed action code, and their execution mode. Possible modes are `suspended` (i.e., the execution is suspended and waits for a debug command), `run` (i.e., the execution will run without any interruption), and `continued` (i.e., the execution will stop at the next breakpoint). Also, the current execution state is highlighted with red in the related state machines as shown in 5 and 6.

Due to the semantic gap between model- and code-level, providing similar features using program debuggers is not possible without extra instrumentation at code-level in addition to the required mapping. To illustrate, consider the use of the command `info thread` in GDB to determine the currently running threads. Fig. 3 shows the output of this command for the running example at two different times. Since both `pinger` and `ponger` are assigned to the same thread, each time only one of the capsules and its current state is shown. However, based on the semantics of UML-RT both capsules are running and their current state should be shown as MDebugger does it (1). The fundamental problem is that the program debugger is, by default, unaware of modeling concepts.

Control execution commands. The commands `breakpoint`, `run`, `next`, and `continue` allow for controlling the execution of capsule instances. Using the `breakpoint (b)` command, breakpoints can be set before or after action code. The `run` command executes the instance without interruption, disregarding of any breakpoints set. The `next (n)` command steps the execution of the capsule instance. The `continue` command executes until a breakpoint is reached. Intuitively, if no breakpoint is set, the two commands `continue (c)` and `run (r)` behave the same. These commands work at the capsule-level, e.g., a specific capsule instance can be stopped while other capsules are running. E.g., as shown in 1, setting a breakpoint at the end of the exit action code of the `PLAYING` state of the `ponger` capsule and then using the `continue` command steers the execution of the capsule to `B__Ac_onPing` which corresponds to the end of the exit action code of `PLAYING` and the beginning of the action code of the `onPing` transition. Execution control in the GUI is managed in the main toolbar (8) and breakpoints are set by selecting the related elements in the state machines. Also, a list of breakpoints is shown in the *breakpoint* view (3).

Again, due to the semantic gap, it is almost impossible to provide this feature at capsule level using program debuggers unless each capsule is assigned to a separate physical thread (which may change system behaviour and thus lead to other issues). In the context of the example, as discussed, both capsules `pinger` and `ponger` are assigned to the same physical thread, so stopping/resuming the thread will respectively stop/resume the execution of both capsules. However, using MDebugger each capsule execution flow can be controlled separately. Since this feature is essential for interactive debugging but cannot be supported easily with a program debugger, it illustrates why the use of a program debugger to develop a model-level debugger is problematic.

View and change variables. The `modify (m)` and `watch (w)` commands are used to view and change the values of the variables (attributes) of a capsule. `watch` shows the variable values based on the last execution trace even when the capsules are executed in `run` mode which is useful for runtime analysis activities. `modify` only works when the execution of a capsule is suspended. 2 shows a typical scenario involving these commands. 2 shows how these commands are supported in the *GUI*.

Backtracing: The `backtrace (bt)` command can be used to 'unwind' the execution of a capsule by showing the list of the *n* last execution steps. 3 shows sample output of the `backtrace` command for the `pinger` capsule instance where the traces are ordered based on their occurrence time. Each trace contains detailed information that allows the user to understand where and when specific attribute values are modified, which message caused a transition to be taken. In the GUI, MDebugger shows the last 5 execution traces as shown in 1. Also, Mdebugger provides a `seq` command which generates a sequence diagram based on the execution traces. The generated sequence diagram can be viewed using *PlantUML* [16].

Table 1: Model Complexity and Instrumentation Time

Model	Model Complexity						Instr. time (ms)
	Original			Instrumented			
	C	S	T	C	S	T	
Counter	1	2	2	2	14	20	445
Car Door Lock	4	8	18	5	95	144	943
Parcel Router	8	14	25	9	158	244	1488
Rover	6	16	21	7	134	200	1397
FailOver System	9	28	43	10	254	396	1528

C: Capsule, *S*: State, *T*: Transition

Integration with other tools While MDebugger provides the foundational features for model-level debugging, many other activities and value-added services can be developed by using MDebugger services. Other applications can use the MDebugger services in three ways: (1) Execution traces can be saved in a textual format using the `save` command and analyzed in offline mode. (2) The core component also provides a `connect` command for connecting to external applications using *TCP* and publishing all execution traces in real-time. (3) Finally, external applications can interact with the debuggable program and use its services directly.

4 EVALUATION

A detailed evaluation of our approach is discussed in [2]. We have applied the instrumentation to several system models which are listed in Table 1 and evaluated our approach using three metrics: size overhead, instrumentation time, and performance overhead.

Instrumentation Time. To measure the instrumentation time and verify that our approach is scalable, we measured the time required by the tool to create the instrumented version of the model from which code is generated. For each use case, we measured instrumentation time 20 times and calculated the average. The last column of Table 1 shows that this average varies between 445 and 1,523 milliseconds depending on the system model, which is within the range of a second and a good indication of scalability of approach.

Size Overhead. We assumed that existing solutions relying on program debuggers require a program with debugging symbols and mapping data to support model-level debugging. For the evaluation, we generated three binaries for each use case illustrated in Table 1: two from the original model (with and without debugging symbols used by GDB) and one from the instrumented model (without debugging symbols). We compared the size overhead of the binary containing the debugging symbols (required by GDB) and the instrumented binary, w.r.t. to the original size of the system (without debugging symbol nor instrumentation). The result shows that the size of the instrumented binary is almost equal to the size of the binary with debugging symbols required by GDB. Thus, we can safely conclude that the size overhead of our approach is lower than that of alternative approaches which require mapping data in addition to debugging symbols.

Performance Overhead. We set up a benchmark for evaluating the performance of the FailOver system [13] under normal mode (to show the real performance of the system) and debugging mode using our approach. In both cases, we executed the system until 10,000 messages had been sent by the clients and processed by the servers. The results showed that the average performance overhead of MDebugger is on the order of microseconds per request which is acceptable for many application domains.

5 CONCLUSION

We have presented MDebugger and discussed its use to debug UML-RT models. It uses debuggable code resulting from model instrumentation rather that program debuggers. Currently, we are extending our approach to support debugging of partial models (i.e., the behavioral parts are specified partially or not specified at all). To best of our knowledge, there is no model-level debugger that supports debugging models without behavior, and we hope our work will be one of the first work that will address this problem.

REFERENCES

[1] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2017. MDebugger Repository. https://github.com/moji1/MDebugger. (2017). Retrieved June 5, 2017.

[2] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2017. Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems. In *11th Joint Meeting on Foundations of Software Engineering*. 419–430.

[3] CEA. 2017. Papyrus Documentation. https://www.eclipse.org/papyrus/documentation.html. (2017). [Online; accessed 01-July-2017].

[4] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2016. Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 36–43.

[5] Dolev Dotan and Andrei Kirshin. 2007. Debugging and Testing Behavioral UML Models. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, 838–839.

[6] Eclipse Foundation. 2016. Eclipse Papyrus for Real Time (Papyrus-RT). https://www.eclipse.org/papyrus-rt. (2016). Retrieved: 2016-03-10.

[7] Philipp Graf and Klaus D Muller-Glaser. 2006. Dynamic Mapping of Runtime Information Models for Debugging Embedded Software. In *17th IEEE International Workshop on Rapid System Prototyping, 2006*. IEEE, 3–9.

[8] Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. 2010. Model-Level Debugging of Embedded Real-Time Systems. In *10th Int. Conference on Computer and Information Technology (CIT'10)*. IEEE, 1887–1894.

[9] IBM. 2015. Rational Software Architect RealTime Edition, v9.5.0 Product Documentation. http://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0. (2015). Retrieved: 2016-03-10.

[10] Padma Iyenghar, Clemens Westerkamp, Juergen Wuebbelmann, and Elke Pulvermueller. 2010. A Model Based Approach for Debugging Embedded Systems in Real-Time. In *10th ACM Int. Conference on Embedded Software*. 69–78.

[11] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. 2018. Survey and Classification of Model Transformation Tools. *Software and Systems Modelling* (2018). to appear, accepted February 2018.

[12] Nafiseh Kahani, Mojtaba Bagherzadeh, Juergen Dingel, and James R. Cordy. 2016. The Problems with Eclipse Modeling Tools: A Topic Analysis of Eclipse Forums. In *ACM/IEEE 19th Int. Conference on Model Driven Engineering Languages and Systems*. ACM, 227–237.

[13] Nafiseh Kahani, Nicolas Hili, James R Cordy, and Juergen Dingel. 2017. Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering*. IEEE Press, 12–18.

[14] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The Epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*. Springer, 46–60.

[15] Ernesto Posse and Juergen Dingel. 2016. An Executable Formal Semantics for UML-RT. *Software and Systems Modeling* 15, 1 (2016), 179–217.

[16] Arnaud Roques. 1999. PlantUML. http://plantuml.com/. (1999). Accessed July, 2017.

[17] Bran Selic, Garth Gullekson, and Paul T Ward. 1994. *Real-Time Object-Oriented Modeling*. Vol. 2. John Wiley and Sons New York.

[18] Richard Stallman, Roland Pesch, and Stan Shebs. 2016. Debugging with GDB. http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz. (2016). Accessed August, 2016.

[19] Willem Visser, Matthew B Dwyer, and Michael Whalen. 2012. The hidden models of model checking. *Software & Systems Modeling* 11, 4 (2012), 541–555.