Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems

Mojtaba Bagherzadeh, Nicolas Hili, Juergen Dingel
{mojtaba,hili,dingel}@cs.queensu.ca

ABSTRACT

Providing proper support for debugging models at model-level is one of the main barriers to a broader adoption of Model Driven Development (MDD). In this paper, we focus on the use of MDD for the development of real-time embedded systems (RTE). We introduce a new platform-independent approach to implement model-level debuggers. We describe how to realize support for model-level debugging entirely in terms of the modeling language and show how to implement this support in terms of a model-to-model transformation. Key advantages of the approach over existing work are that (1) it does not require a program debugger for the code generated from the model, and that (2) any changes to, e.g., the code generator, the target language, or the hardware platform leave the debugger completely unaffected. We also describe an implementation of the approach in the context of Papyrus-RT, an open source MDD tool based on the modeling language UML-RT. We summarize the results of the use of our model-based debugger on several use cases to determine its overhead in terms of size and performance. Despite being a prototype, the performance overhead is in the order of microseconds, while the size overhead is comparable with that of GDB, the GNU Debugger.

1. INTRODUCTION

Due to the growing complexity of Real-Time and Embedded (RTE) systems, the need for development methods and tools that provide effective means for the efficient development of correct embedded software is increasing. To meet these demands, Model-Driven Development (MDD) has been proposed. MDD uses models rather than source code as primary development artifact in an attempt to raise the level of abstraction. MDD supports the generation of code and can facilitate testing and verification activities [9].

Typically, developers spend a significant part of their time debugging applications and fixing bugs [29] supported by sophisticated program debugging tools. However, modelling tools currently do not provide proper support for debugging and understanding the run-time behaviour at model-level [21, 20, 37]. Instead, developers must use a program debugger on the generated source code to debug their models, which contradicts MDD principles and goals because many of the benefits of the abstraction are lost. Additionally, understanding the generated source code can be challenging and error prone for developers who use only models for development and are not versed in the target language, i.e., the language the generated code is in [37]. Inadequate debugging support is one of the central barriers to a broader adoption of MDD.

Recent efforts have provided a good starting point for understanding the challenges and requirements for model-level debugging [21, 20, 12, 18, 15, 14, 11, 41, 46]. The most frequent proposals are to (1) realize model-level debugging through a model interpreter, or to (2) maintain traceability information between model and code and leverage an existing program debugger. Interpreter-based proposals require the implementation of an interpreter for the modeling language, the action language, and the run-time services library; this is not only time-intensive, but also creates a harmful discrepancy between the environment that the model is debugged in and the environment that the code generated from the model will execute in; this discrepancy might create spurious bugs or mask real bugs that depend on, e.g., the choice of target language or hardware platform; also, we would not expect a large degree of portability of the debugger from one MDE tool to the next. Program debugger-based approaches are very dependent on the target language, the operating system, and the hardware platform, meaning a change in any of these is likely to necessitate substantial changes to the debugger.

In this paper, we propose a novel approach for realizing model-level debugging of models of RTE systems. Our approach overcomes the above-mentioned limitations of existing proposals by being *completely* platform-independent, i.e., it does not depend on any architecture-specific program debugger. Instead, our approach relies on *model transformation* to instrument the model to be debugged with information that allows it to support debugging activities. Since the debugging support is realized on the model level, debuggable code can be generated from the model with the standard code generator and without the need for additional, code-level instrumentation or a program debugger.

We have implemented our approach in a model-based debugger called *MDebugger*. To maximize the impact of our work, our implementation only uses *open source* tools including the Papyrus-RT MDE tool for modelling and code

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

generation, and the Epsilon tools for model instrumentation. Just like Papyrus-RT, MDebugger is applicable to models expressed in UML for Real-Time (UML-RT) [36]. However, our approach is transferable to other modeling languages and tools used for the MDD of RTE systems.

The remainder of this paper is organized as follows. The next section reviews the most relevant parts of UML-RT and the current practices in model-level debugging; Section 3 presents our approach; Section 4 describes our implementation; Section 5 evaluates our approach using performance and code size metrics; Section 6 presents related work and compares our approach; Section 7 concludes.

2. BACKGROUND

The objective of this section is twofold. First, it outlines current research in model-level debugging and its limitations. Second, it introduces UML-RT using a running example.

2.1 Model-Level Debugging

Over the past years, several approaches have been proposed to improve the current state of support and integration of debugging features in MDD tools [21, 20, 12, 18, 15, 14, 11, 41, 46]. The most advanced proposals implement model-based debugging features on top of an existing program debugger (e.g., GNU Debugger) as shown in Fig. 1. These approaches usually rely on *code instrumentation* in order to generate the meta-data required to keep a bi-directional mapping between code artifacts and model elements (states, transitions, etc.) [12, 45]. Binary code is then generated and contains debugging symbols used by existing program debuggers to debug the application. An additional component at model-level interfaces with the program debugger in order to display the debugging information directly on the model and to trigger debugging commands from the modeling tool.

While this solution works in practice, it suffers from several limitations. The most important one is caused by the need to integrate the model debugger with different program debuggers that are specific to a target language. For instance, at least three different integrations with program debuggers are required for supporting three different target languages (C++, Java, etc.). The definition of different mappings is furthermore impacted by the number of architectures (Intel x86, ARM, etc.) to support. The resulting dependencies makes the task of maintaining comprehensive debugging capabilities at model-level difficult and time-intensive for tool vendors.

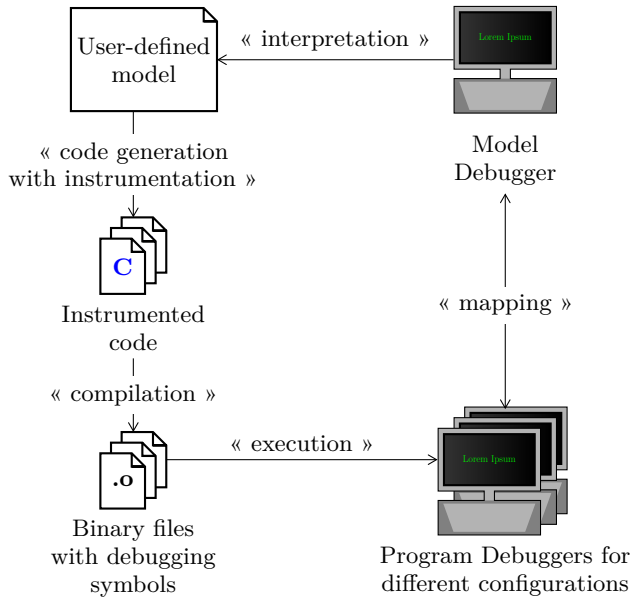


Figure 1: Model-Level Debugging Workflow

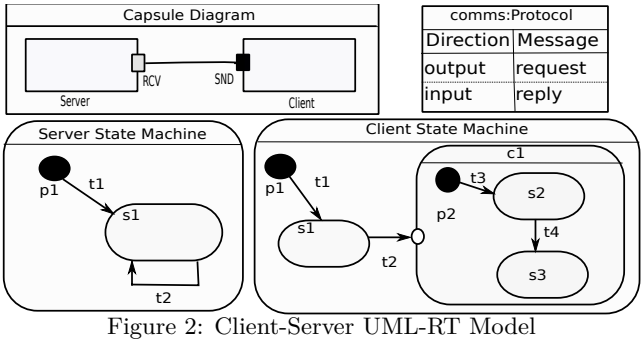


Figure 2: Client-Server UML-RT Model

A second limitation is the lack of reusability. Existing approaches focus on instrumenting the code instead of the modeling language which introduces a dependency on the MDE tool. For instance, UML-RT is supported by IBM RSA-RTE and Papyrus-RT. However, the need for instrumenting the code generator for debugging purposes would prevent a model-based debugger used in one tool from being easily reused in the another tool. Consequently, proposed solutions for model-level debugging are hard to port from one MDE tool to another.

A third limitation is due to the semantic gap between model and code elements, making the translation of debugging information to the model-level difficult. In the context of UML-RT, the following concrete example can be given: According to the UML-RT execution semantics, a capsule (i.e., component) instance executes in its own logical thread, i.e., even if the capsule code is not assigned to a physical thread in the target language, the controller in the UML-RT run-time system will create the illusion to the user that the instance is executing concurrently; however, since the notion of logical threads is not supported by the program debugger, debugging capsule instances using program debuggers becomes unnecessarily complicated.

Overall, given its significance, debugging appears to be an insufficiently researched topic, not just in the MDD context, but also for more traditional software development using general-purpose or domain-specific languages.

2.2 UML for Real-Time

UML for Real-Time (UML-RT) [38, 32] is a language specifically designed for RTE systems with soft real-time constraints. Over the past two decades, it has been used successfully in industry to develop several large-scale industrial projects, and has a long, successful track record of application and tool support, via, e.g., IBM RSA-RTE [2] and PapyrusRT [1]. UML-RT is designed as a UML profile with a simplified notation. It only provides two diagrams: *capsule* and *state machine* diagrams.

The central modelling entity in UML-RT is called *capsule* (cf. Fig. 2). A capsule is similar to an active class in object-oriented programming languages or to a process in process algebra. Being active implies that each capsule may have autonomous behaviour. Capsules own a set of internal and external *ports* that are typed with *protocols*. A protocol defines the different incoming and outgoing *messages* a capsule can receive or send through its ports. A port is the sole interface for the communication between the capsules which guarantees high encapsulation. Ports of two capsules can be connected through *connectors* only if they are typed with the same protocol. Furthermore, capsules can have *attributes*, *operations*, and *parts* (aka. sub-capsules) [38, 36].

Fig. 2 shows a simple *Client-Server* system modeled with UML-RT.The top capsule is composed of two capsule parts, `client` and `server`. Both are connected through their ports `SND` and `RCV`, typed with the protocol `comms`.

Capsule behaviour is modeled using *hierarchical state machines*. A UML-RT state machine consists of several *states* connected using *transitions*. States can be of three kinds: basic states, composite states (containing sub-states), and pseudo-states (e.g., initial pseudo-state, choice point). A basic or composite state can have *entry* and *exit* action code that is executed when the state is entered or left, respectively. A *transition* connects a *source* state to a *target* state. It may contain a *triggering event*, a *guard*, and an *effect*. A transition is taken when the triggering event is fired and the guard evaluates to true. When it is taken, the code representing the transition effect is executed.

Fig. 2 illustrates the state machines of the `server` and the `client` capsules. The `server` state machine only contains a pseudo-state `p1` and a basic state `s1`. A regular transition `t1` connects both states, and an external self-transition `t2` loops around `s1`. The `client` state machine contains a composite state `c1`, which is entered when the transition `t2` is taken.

The semantics of UML-RT state machines is similar to that of UML state machines with some restrictions, including: (1) there is no AND-state (orthogonal regions), (2) the UML concepts *fork*, *join*, *shallow history*, and *final states* are prohibited in UML-RT, (3) transitions cannot cross state boundaries, and (4) states do not have idle (*do*) actions [32]. In addition, the execution semantics of UML-RT is managed by a Run-Time System (RTS) library, which defines one or more *controllers* to monitor the concurrent execution of each capsule. A controller is assigned to a thread and controls the execution of a set of capsules. An important characteristic related to the execution semantics of UML-RT is *run-to-completion*, which guarantees that an incoming message will be fully processed before the processing of the next message starts.

As an example, the `server` state machine (cf. Fig. 2) contains two transition chains. When the server starts, the transition `t1` is taken. The transition effect of `t1` and the entry action code of `s1` are executed in this order. State `s1` remains active until an event matching the trigger of transition `t2` is fired. When this event is fired, `t2` is taken and the full transition chain, consisting of the exit action code of `s1`, the transition effect of `s2`, and the entry action code of `s1`, is executed. Run-to-completion guarantees that the execution of the entire transition chain is not interrupted.

In our approach, we classify transitions based on whether or not their source or target states are pseudo-states. We distinguish the following four groups of transitions: (1) P2P: transitions in this group connect two pseudo-states. Assuming that there is not any guard associated with the transition, a P2P transition is taken as soon as its source state becomes active. (2) P2N: transitions in this group connect a pseudo-state to a non-pseudo state (e.g., `t1` from server state machine). Similar to P2P transitions, a P2N transition is taken as soon as the source state becomes active. (3) N2N: transitions in this group connect two non-pseudo states (e.g., `t2` from server state machine). In contrast to P2P and P2N transitions, an N2N transition is only taken when an event matching its trigger is raised by the RTS or another capsule. (4) N2P: transitions in this group connect a non-pseudo state with a pseudo-state (e.g., `t2` from the client state machine). Similar to N2N transitions, an N2P transition is only taken when an event matching the transition's trigger is raised by the RTS or another capsule.

3. APPROACH

As stated before, existing approaches suffer from several limitations many of which result from the dependency on program debuggers specific to a particular language or architecture. For instance, in addition to having to generate the code for different configurations, existing model-level debuggers have to provide wrappers per program debugger.

Our approach relies on a model instrumentation process which is completely platform-independent, as shown in Fig. 3. Differences with existing approaches (cf. Fig. 1) are highlighted in the figure. The first difference is that our approach uses *Model-to-Model (M2M)* transformation techniques for creating an instrumented version of the user-defined model supporting debugging activities. This support is added via model transformation rules that are defined for each construct of the modelling language. This step is essential to our approach. It allows, without having to instrument the code, the generation of applications providing debugging services by themselves, i.e., without having to rely on a program debugger. The fine control provided by the model transformation techniques allows us to implement at model-level not only the communication between the debuggable system and the model debugger, but also advanced capabilities such as component introspection and attribute value change. Finally, as the generated code is debuggable, the compilation step does not require the addition of language- or architecture-specific debugging symbols into the binary files, reducing their size.

Compared to existing approaches, ours is more general, less dependent of the specifics of code generators or deployment configurations. As model instrumentation is done at model-level, the resulting debugger is easier to port from one tool to another. In addition, their is no gap between debug information provided at code- and model-level as the debugger does not rely on any program debuggers.

The remainder of this section shows how models are instrumented with respect to them. It then gives an example of an application and discusses to what extent our approach guarantees the preservation of behaviour.

3.1 Model Instrumentation

Model-level debuggers should provide a wide range of operations, such as attribute view and change and crash analysis. These operations can be built upon three composite operations including: (1) *stop and resume* operations for controlling the execution of the system via breakpoints, pause,

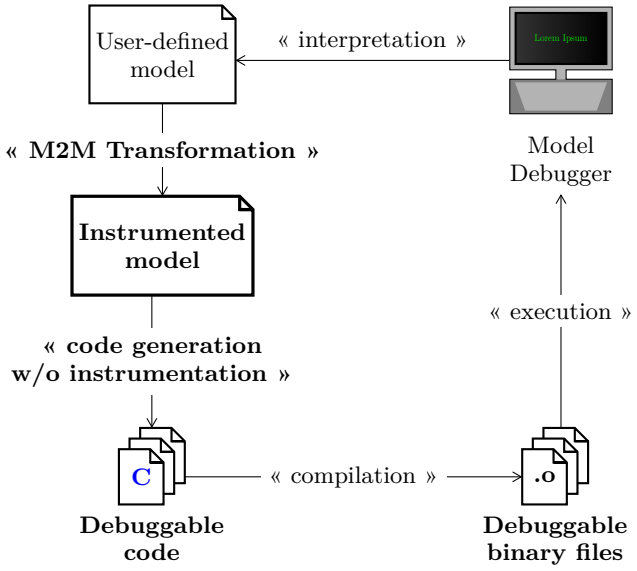


Figure 3: Model-Level Debugging Workflow

resume, step in, and so on; (2) *view and change attribute* operations to inspect and modify the system status; (3) *detailed tracing* operations providing a foundation for crash analysis, event chain analysis, punctuality analysis, utilization analysis, and so on.

To formalise our approach, we first observe that instrumentation can be strictly bounded to *transition chains* as computation only occurs in three places: transition effects, state entry actions, and state exit actions. Indeed, the semantics of UML-RT state machines excludes any *do* activities. Therefore, we define a *transition chain* of a transition t_1 between two states s_1 and s_2 as the sequence of actions that are executed when the transition is fired. Since pseudostates do not contain exit or entry actions, the actions in the transition chain of a transition thus depend on which one of the four groups defined at the end of the previous section, the transition is in. Let t_{P2P}, t_{P2N}, t_{N2P}, and t_{N2N} denote transitions with source state s_1 and target state s_2 in each of these four groups. Then, their transition chains are given by

$$\begin{aligned}
\text{chain}(t_{P2P}) &= \langle \text{effect}(t_{P2P}) \rangle \\
\text{chain}(t_{P2N}) &= \langle \text{effect}(t_{P2N}), \text{entry}(s_2) \rangle \\
\text{chain}(t_{N2P}) &= \langle \text{exit}(s_1), \text{effect}(t_{N2P}) \rangle \\
\text{chain}(t_{N2N}) &= \langle \text{exit}(s_1), \text{effect}(t_{N2N}), \text{entry}(s_2) \rangle
\end{aligned}$$

The execution of a capsule in UML-RT can be entirely represented by the composition of all transition chains that lead the system from one state to another. Consequently, by focusing on transition chains only, our approach supports the instrumentation of the entire model for debugging purposes.

The following shows the formalisation of different transformation rules we use. We formulate them at the model-level and then employ model query and transformation techniques to refine the user-defined model. Fig. 4 shows how a N2N transition t_1 between states s_1 and s_2 is instrumented in order to provide debugging support for the transition

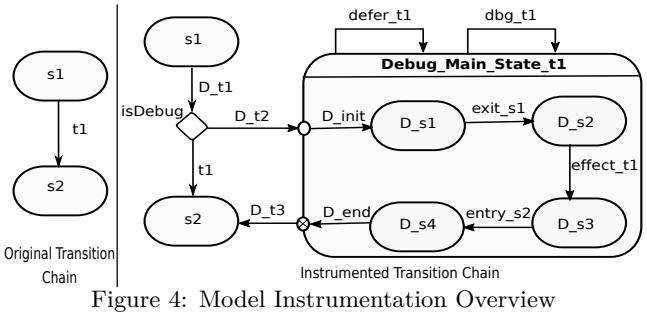


Figure 4: Model Instrumentation Overview

chain. The left side of the figure shows the original transition t_1. The right side shows the refined transition after instrumentation. The refined model introduces a choice point and a composite state a capsule may enter whenever a breakpoint is reached. We will now describe how the refined model allows the three different kinds of debugging operations mentioned above.

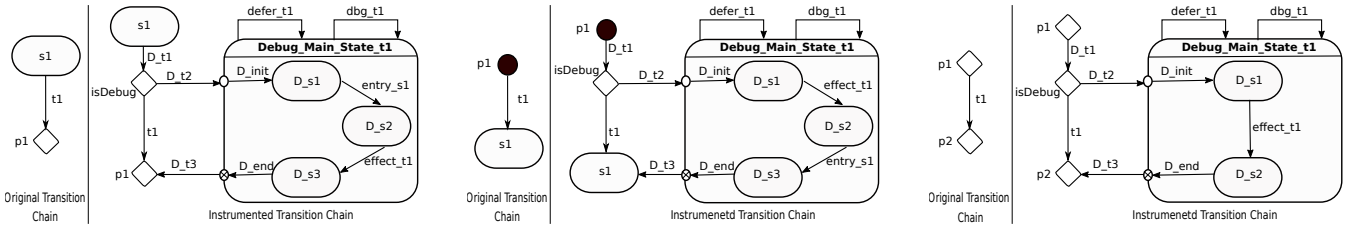
Suspend and Resume Operations.

Fig. 4 shows how a capsule execution can be suspended and resumed. The presence of the choice point allows the application to check whether a breakpoint on the transition is set or not. If it is not set, the execution normally continues by executing the transition chain. Otherwise, it enters the composite debugging state and the transition chain is executed step-by-step. To do so, a mapping is created between the different actions composing the initial transition chain and the local transitions in the composite debugging state. It provides support for common *stop and resume* operations such a *resume* and *step over*. Stepping over the transitions `exit_s1`, `effect_t1`, and `entry_s2` allows for executing the different actions (exit action of `s1`, transition effect of `t1`, entry action of `s2`) of the transition chain separately.

Initially, the execution goes into the debugging state `D_s1`, where it is suspended before the exit action code of the original state `s1` is executed. In this state, the execution is pending, waiting for commands from the model debugger to resume its execution. When the command is received, the transition `exit_s1` is taken, causing the original exit action code to be run, and suspending the execution in `D_s2`. Another command results in taking the transition `effect_t1` associated with the execution of the transition effect of `t1` and in suspending the execution in `D_s3`. A third command causes the execution of the entry action of the original state `s2` associated with the transition `entry_s2`. A last command is required to leave the composite state and resume the regular execution.

Attribute View and Change Operations.

In addition to providing support for monitoring the execution of a capsule, the transformation rule also adds one self-transition `dbg_t1`. It provides a support for inspecting and changing the system status at run-time. Program debuggers such as GDB directly access the program stack and heap to provide these operations, which results in a tight coupling with the architecture. In contrast, our implementation relies on self-reflection techniques [23]. Therefore, when the execution is suspended, the model debugger can directly inspect and change the attributes of the suspended capsule. Note that specific languages, such as C++, provide



(a) Second rule: P2N

(b) Third rule: N2P

(c) Fourth rule: P2P

Figure 5: Alternative Transformation Rules

limited, if any, support for self-reflection (in contrast with, e.g., Java [28]). However, this limit is overcome by our approach as the support for self-reflection is directly provided by the instrumented model.

Detailed Tracing Operations.

Having access to the history of the system execution during the debugging process is important for localizing bugs efficiently. For example, analysing execution traces can help to find the root of an unexpected behaviour. Execution traces can also provide information for other analysis activities which are crucial for RTE systems (e.g., Worst Case Execution Time analysis, Run-time verification). To support execution traces, we add tracepoints to capture all possible events generated by the system. Similar to the work on schedulability analysis in [16, 27], we define an event class and event type so that the captured execution traces contain the necessary information.

Coverage of all Transitions.

To cover the transformation of all transition types (i.e., N2N, N2P, P2N, P2P) which discussed in Section 2, we define a transformation rule per each transition type. The transformation rule illustrated in Fig. 4 is applied to N2N transitions; since the chain of N2N transitions contain all possible action code four debugging substates and related transitions are required. Fig. 5 shows the transformation rules for the other three transition types. In other words, the rules in Figs. 5a, 5b), and Fig. 5c are applied to P2N, N2P, and P2P transition respectively.

3.2 Behaviour Preservation

Behaviour preservation is an important concern to address when developing debuggers. While debuggers always impact the performance on a debugged application, the added overhead resulting from in their use should be minimized as much as possible in order to prevent a significant change in the system behaviour. In this part, we discuss in which sense our approach to debugging is behaviour preserving.

Fig. 6 shows how an execution trace is altered when using our approach. Top and bottom parts respectively show the execution trace of the transition `t1` before and after model instrumentation. Nominal and debugging paths are respectively shown with solid (—►) and dashed lines (- -►). In both cases, the transition chain is started when the event triggering the transition `t1` is fired. In the instrumented model, the capsule checks (`isDebug`) if the debugging mode is set. In that case, the execution follows the debugging path. As explained before, states `D_s1` to `D_s4` are states

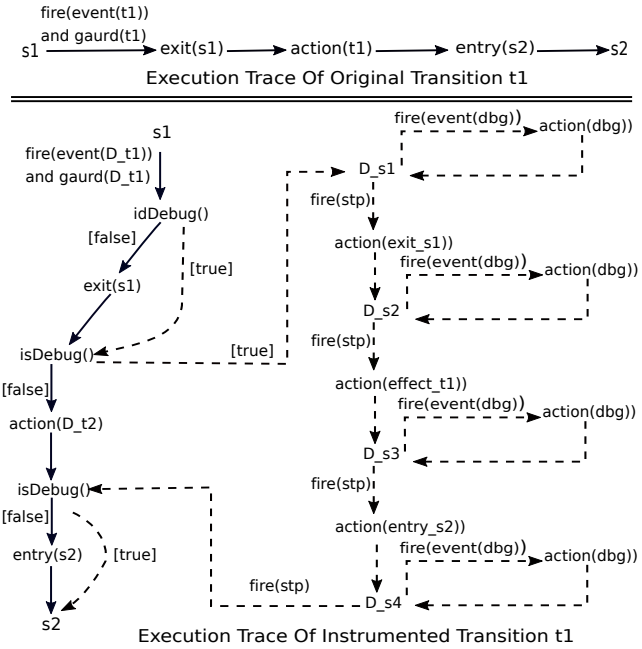


Figure 6: Execution Trace

in which the capsule waits for debugging commands for resuming its execution or changing its state. Stepping over these states is done when receiving the *step* command from the model debugger. Each step results in the execution of a part of the transition chain of `t1`. After executing the entire transition chain, the execution is resumed.

When the debugging mode is not active (*isDebug* returns false), we can observe that the instrumented transition chain executes the same sequence of actions as the original. The *isDebug* function is only used to check whether the system is in or should enter the debugging mode (e.g., after reaching a debugging breakpoint). It is read-only, very small (1 line of code) and executes very quickly. Unless the system is sensitive to slight delays, the execution of the *isDebug* function will not change the behaviour of the debugged system.

Under debugging mode (*isDebug* returns true), the execution can be delayed or modified with the help of inspection and step commands. However, the order in which the steps in the transition chain are executed is preserved. Moreover, unless the user alters the system state (by, e.g., changing attribute values), the execution of the transformed transition chain will terminate in the exact same state as the original transition chain. Stepping operations from the model de-

bugger do not alter the system state, and are only used for advancing through the original transition chain.

In the context of concurrent executions, delays introduced by the debugger can have serious side-effects in the system behaviour and its preservation cannot be guaranteed. Indeed, while the execution of a debugged component is interrupted after, e.g., a breakpoint is reached, other components are still running and may send messages to it. It may result in loss of messages or timeout errors, the suspended component being unable to process incoming messages or to acquit from their reception.

While there is no universal solution in existing approaches, this issue is partially addressed in our approach. By relying on the *defer* and *recall* mechanism of UML-RT, our approach prevents messages from being lost. To do so, we modeled a transition defer_t1 (cf. Fig. 4) which captures all incoming messages sent by other capsules to recall them after the debugged capsule returns to its nominal execution. This mechanism preserves the order of the received messages. Therefore, the transition rec_t1 helps ensures that the behavior of the debugged capsule is preserved. However, it does not guarantee that the behavior is preserved for time-sensitive systems. Note that the debugging of timed distributed systems suffers from this problem in general.

4. IMPLEMENTATION

This section describes our current implementation. We use Papyrus-RT as the primary tool for modelling UML-RT models and Epsilon [22] to implement the transformation rules required for instrumenting the models. This section is divided into two parts. The first part details the implementation of the transformation rules. The second part details the prototype implementation of a model-level debugger called *MDebugger*.

4.1 Model Instrumentation using Epsilon

Transformation rules used for instrumenting the models are implemented using the Epsilon Object Language (EOL). It supports a set of instructions to create, query, and modify models expressed in languages described with the Eclipse Modeling Framework (EMF).

Listing 1 is an example of a transformation rule written in EOL. It shows the main function for instrumenting the state machine of all capsules of the user-defined model. The `addGateway` function is responsible for enabling the model to interface with the debugger. It adds a UML-RT port to each capsule. These ports are typed with a specific protocol used for debugging purposes. The `refineStructure` function adds required attributes and methods to each capsule to support debugging. An example of attributes is a map used for maintaining breakpoint information, required by the `isDebug` method during debugging. Examples of methods added are the `isDebug` method and a set of methods for supporting attribute view and change operations. The `isDebug` method returns a boolean value indicating whether a debugging session is opened or needs to be opened according to the current state and the next transition about to be taken. The generation of self-reflection methods was inspired by work on physics engine development [25]. These methods provide support for viewing and changing attribute values. To do so, the `refineStructure` function iterates over all attributes of each capsule and generates the corresponding helper functions, such as getters and setters. The `refineForSRO` function applies the four transformation rules explained in Section 3. For each transition chain, a helper function is used in order to determine which instrumentation rule needs to be applied, based on the kinds of source and target states. Based on the result, the `refineForSRO` calls the proper transformation rule. The `addTrace` function adds support for detailed tracing operations to each state. Finally, the `guardCodes` function adds a guard to every entry and exit code to prevent them from being executed when the capsule is being debugged.

```
addGateWay();
refineStructure();
for (SM in allStateMachines){
    refineForSRO(SM);
    for (s in allStates){
      s.addTrace(traceType);
      s.guardCodes();
      }
    }
```

Listing 1: Main Transformation Functions

4.2 Model-Based Debugging

After applying the instrumentation, code is generated from the instrumented model and an executable and debuggable binary is built from the code. Fig. 7 gives an overview of our current implementation. The debuggable binary consists of two parts: the instrumented binary itself, and a *debugging agent* responsible for managing debugging sessions and for interfacing with the model debugger. The debugging agent interfaces on one side with the instrumented binary through messages exchanged using the Papyrus-RT RTS library, and on the other side with the model debugger through two queues: an event queue for sending detailed traces and a command queue for receiving debugging commands for, e.g., suspending the execution or viewing an attribute. Shared memory was favored as it provides efficient and fast asynchronous communication so that the model debugger can interface with the system without disrupting its execution. The model debugger, called *MDebugger*, provides debugging services to users. Its API is accessible either via the command line or via TCP to connect external debugging environments. Using the TCP connection, we developed a graphical user interface by extending the *Eclipse Debugger* perspective.

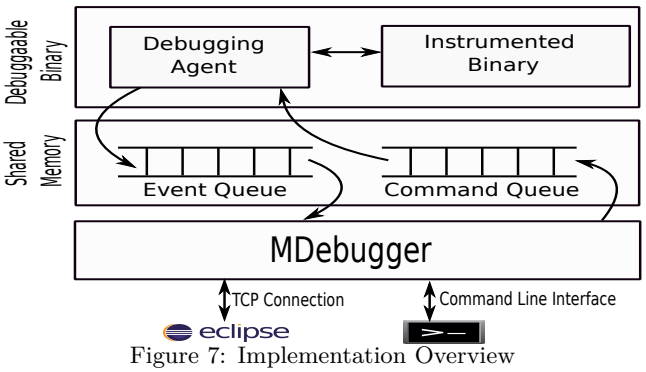


Figure 7: Implementation Overview

4.2.1 Debugging Agent

The debugging agent plays a central role to provide effective debugging services. It was designed as a customized version of a more generic run-time monitoring architecture currently being developed [3, 4]. The debugging agent communicates with the system using the execution semantics of UML-RT. This brings several benefits. It cannot make the system unschedulable or cause unpredictability or synchronization issues.

Initially, the debugging agent first creates the event and command queues for communicating with the model debugger, then waits for every instrumented capsule instance to register. The registration phase is essential so the debugging agent can maintain a list of every capsule instance. It allows the model debugger to identify and send commands to a specific capsule instance to debug. After the registration phase, the debugging agent performs two tasks: 1) it receives and forwards trace events from the registered capsule instances to the event queue, and 2) it periodically polls the command queue and processes any received commands. When a received command is valid, it is transmitted by the debugging agent to the target capsule instance.

4.2.2 MDebugger

MDebugger is a model-level debugger which provides debugging services by interacting with the debuggable binary. It consists of several components including a *core component*, a *command interface*, an *event manager*, and a *communication layer*.

The core component implements the main logic of the debugger. Initially, it queries the list of the registered capsule instances from the debugging agent. For every capsule instance, it keeps track of certain data including: 1) last events generated by the capsule, 2) the active state or transition, 3) the list of attributes, their type and value, 4) the execution mode (e.g., *stepping* or *suspended*), 5) the list of breakpoints set, and 6) the debugging command history related to the capsule instance. In contrast to program debuggers, the list of events represents the stack of a program, the active state corresponds to the instruction pointer, and the list of variables corresponds to the heap of a program.

In addition to the core component, the communication layer provides services for reading from and writing to the shared memory, and for handling TCP connections. The command interface is responsible for receiving and parsing the debugging commands from the command line interface or from external applications connected via TCP.

4.2.3 Supported Features

Our current implementation supports the following features:

Set, remove, and list breakpoints: Similar to program debuggers, breakpoints can be set to suspend the execution at certain points. However, in our implementation, breakpoints are set at the model-level, i.e., before and after entry and exit action code of states, or on transitions. In addition, breakpoints can be set on a specific capsule instance, meaning that only this specific instance is suspended while other instances of the same capsule keep on running. This is even possible in the case where capsules are assigned to the same physical thread.

Stepped execution: Our current implementation supports three execution modes: stepping, run to next breakpoint, and run to completion. As for setting breakpoints, stepping is possible at instance-level, so it is possible to run the system with different debugging scenarios.

View and change attributes: Whenever a capsule instance is suspended, commands can be used to view and change attribute values. In addition, the generated detailed traces allow the user to "rewind" the execution to see where and when attribute values are modified.

View and store event chain: It is possible to view the last event traces generated by each capsule instance. It is very useful for root-cause and crash analysis. In addition, MDebugger allows users to save the existing event chain into a file for off-line analyses. Off-line debugging is useful for RTE systems when no proper user interface or TCP connection is available for live debugging.

4.2.4 Eclipse Debugging Interface

We extended the Eclipse debugging interface along with Papyrus-RT in order to provide a graphical user interface for model-based debugging of UML-RT models. This interface supports all implemented features and was customized to support the semantics of UML-RT. Each capsule instance corresponds to a main execution unit and is displayed in the *debug* view along with its five last events. When a capsule instance is selected, the *variables* view displays all the attributes and their value corresponding to a specific event. When a specific event is selected, all attributes whose value was affected by the event are highlighted. In addition, it is possible to change the values from the interface. Finally, this interface also supports stepped execution.

4.3 Exceptions and Limitations

Currently, instrumentation only applies to explicitly defined transitions. Since transitions from the history state to possible target states are implicitly implemented, the current approach does not provide support for suspending the execution before the entry action code of a state when it is reached from the history state. The same limitation exists for exit code on sub-states during a group transition.

Our current implementation only supports value change of primitive attributes. Changing the value of an attributes typed with a specific class is not currently possible.

5. EXPERIMENT

This section details the experiment we conducted in order to assess the applicability of our approach. The first part presents the experiment protocol and hypothesizes we formulated. The second part details results showing the benefits of our approach in terms of performance and scalability.

5.1 Experiment Protocol

In order to assess the applicability of our approach, we formulated two hypotheses w.r.t. the expected gains compared to existing approaches relying on general-purpose program debuggers. The two hypotheses are formulated below:

HYPOTHESIS 1. *[Performance] We hypothesize that our approach relying on model instrumentation causes reasonable performance overhead, negligible enough so it can be applicable to RTE systems*

HYPOTHESIS 2. *[Code Size] While our approach based on model instrumentation undoubtedly increases the size of the*

generated code (as it significantly increases the complexity of the models), we hypothesize that the size of the generated binary file is within the size range of binary files containing debugging symbols used by general-purpose debuggers.

To verify the two hypotheses, we compared MDebugger with the normal execution and, when applicable, with GDB, one of the most widely used program debuggers. Despite being semantically different as they operate at different levels, the comparison is viable as existing model-based approaches rely on program debuggers, hence inherit the overhead induced by them, in addition to other possible overhead resulting from the mapping between code- and model-level. It may be argued that GDB provides more advanced features and debugging services at binary-level, making the comparison unfair. While this is true, it is worth mentioning that most of the services provided by GDB or other program debuggers are not beneficial at model-level, hence merely impede the performance of model-based debuggers.

5.1.1 Use cases

In order to verify our approach, we applied the instrumentation to several RTE system models. Models have different complexities that range from simple models containing two states to models with more than 250 states. Simple models include the *Counter* and the *Car Door Central Lock* systems. *Counter* is a simple system which counts the elapsed seconds. *Car Door Central Lock* is a control system for locking and unlocking car doors.

The *Parcel Router* [40, 24] is an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The system is time-sensitive and jam can appear due to the variation of time spent by a parcel to transit through the different chutes. We created two different versions of the same system. The complete version checks whenever parcel jam occurs and prevents a parcel from being transfered from one chute to another one until it is empty. The simplified version ignores jams.

The *Rover* system model [6] allows an autonomous robot to move through different directions. It is equipped with three wheels driven by two engines. It can move forward, move backwards, and rotate. In addition, it embeds several sensors, such as temperature and humidity sensors to collect data from the environment, as well as an ultrasonic detection sensor to detect and avoid obstacles.

The *FailOver* system [10] is an implementation of the failover mechanism. It involves a set of servers processing client requests. To meet high availability, the system supports two replication modes, *passive* and *active* [17]. In passive replication, one server component works as the master, handling all the client requests while backup servers are largely idle, except for handshake operations. Whenever a malfunction occurs, resulting in a failure of the master server, a backup server is ranked up as the new master. In active replication, client requests are load-balanced between several servers. In addition to processing client requests, each server has to update its status to inform other servers of its availability. Therefore, each server can be notified whenever a malfunction causes the failure of one of its peers.

5.1.2 Quality Metrics

Beside assessing the coverage of our implementation, we evaluated our approach based on different quality metrics, such as the size overhead, instrumentation time, and performance overhead. To perform the experiment, we used a computer equipped with a 2.7 GHz CPU and 8GB of memory. All hardware and software configurations and workload of the system during the entire experiment were identical.

Performance Overhead: Performance overhead is one of the main factors impacting the applicability of debugging tools. This factor is even more important in RTE systems where time sensitiveness and behaviour preservation are the main concerns during debugging.

To evaluate this metric, we set up a benchmark for evaluating the performance of the FailOver system 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 were sent by the clients and processed by the servers. Then, based on the system logs, we collected the computation time for replying to a server request (i.e., RequestReply transition), processing message response by the clients (i.e., ProcessingResponse transition), and notifying the availability of each server to its peers (i.e., SendKeepAlive transition). Besides, we also measured the overall execution time, from the first message sent to the last message received and processed.

Size Overhead: This metric shows the impact of model instrumentation on the model complexity, generated code, and binary size. As part of the first hypothesis, the size of the generated code is directly proportional to the increased complexity of the instrumented model. However, as the debugging features are embedded in the model itself, there is no need to generate debugging symbols as used by existing model-based approaches. Therefore, we focus our experiment on the binary size overhead.

To evaluate this metric, we generated for each use case illustrated in Table 1 the code of both the original and the instrumented models. From the code of the original model, we created two binaries: without debugging symbols (to get the real size of the system) and with debugging symbols used by GDB. Then, we created a third binary by compiling the code generated from the instrumented model. We compared the three binaries to compare the size overhead of GDB and our approach. This comparison allows us to compare our approach with general-purpose program debuggers, such as GDB. It does not take into consideration extra overhead caused by existing model-based approaches, e.g., to maintain a mapping between binary-, code-, and model-levels.

Instrumentation Time: To verify that our approach is scalable, we also 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 calculated an average time of the instrumentation over 20 measures in order to reduce margins of error. It would have been interesting to compare the time required for instrumenting models in our approach with the time required for instrumenting code in existing model-based approaches. Unfortunately, no data is available for establishing a comparison.

5.2 Results

Table 1 shows the added model complexity of each model, as well as the time required to instrument them. Depending on the use case complexity, models grow proportionally to the number of transition chains to instrument. Therefore, the number of states is increased by 5 to 11 times and the number of transitions by 7 to 10 times. As for the struc-

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 Central Lock	4	8	18	5	95	144	943
Simplified Parcel Router	8	12	14	9	96	140	1346
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

tural part of the models, one capsule is always added, which corresponds to the debugger capsule. The added model complexity impacts the generated code size which is increased by 3 to 6 times depending on the use case.As for the time required to instrument the models, it varies between 445 and 1,523 milliseconds, depending on the use case, which is within the range of a second.

In order to assess the scalability of our approach, we applied the model instrumentation process twice, successively on the FailOver system model and on the instrumented version of the model. While there is no need to apply the instrumentation on an already instrumented model, this experiment allowed us to check that the instrumentation time does not skyrocket when the model size grows exponentially. Note that, although intensive efforts are put in Papyrus-RT to create an industrial-grade development tool, it is still in incubation phase, hence no industrial model is available to assess the scalability of our approach.

The resulting model includes 2363 states and 3725 transitions and the instrumentation time is an average of 41 seconds. It shows that the approach is scalable enough for industrial models.

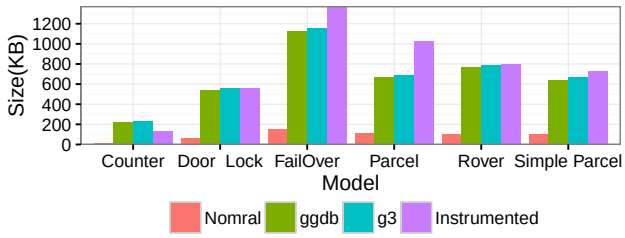


Figure 8: Binary Size Comparison

Both instrumented models and generated code are intermediate in our approach, and their size growth does not impact user experience. However, they do impact the size of the binaries created from them. Fig. 8 shows a comparison of the size of the binary files using our approach and GDB (compiled with -ggdb and -g3 flags). It shows that for each use case, the debuggable binary resulting from our approach is an average of 8 times as large as the original binary. We argue that this is reasonable and even comparable with program debuggers as the size overhead is within the range of values when compiling the code with debugging symbols (between 7.98 and 8.17 times as large as the original binary files). This validates our second hypothesis.

Compared to traditional model-level debugging approaches, our approach does not add additional overhead other than the one caused by the model instrumentation process. Besides creating an overhead due to the integration of debugging symbols into the binary files, existing methods cause an additional overhead in order to generate and maintain a mapping between the executed binaries and the models. As the description of existing approaches does not include the measurement of this overhead, we could not compare our approach with existing ones.

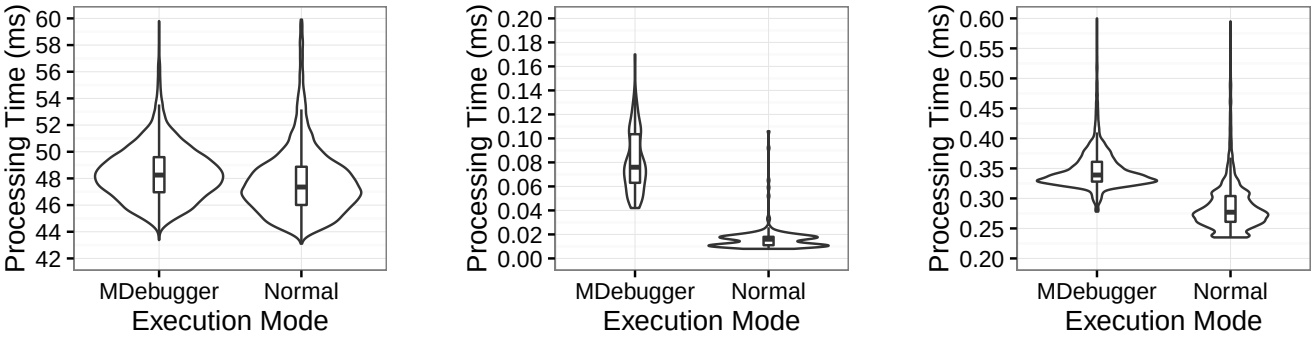
Performance is evaluated over the FailOver system. To do so, we configured the system in the active replication mode with five clients and two servers, and we set up a simple scenario where each client checks the available servers, sends a message, and waits for a reply. Upon receiving the reply, the client processes the response and send a new message after a specific time. The two servers process client requests and send replies. In addition, every five seconds, each server has to update its state to the second server, so other servers are noticed whenever a failure affects a running server.

We set up a benchmark and measured the overhead of our approach compared to the normal execution. To do so, we focused on three specific transition chains: *RequestReply*, during when a client sends a message to both servers and waits for a reply; *ProcessResponse*, during when the client receives a response from one of the servers and process it; *SendKeepAlive* during when a server updates its status to its fellow. Fig. 9 shows violin plots of computation times for the three transition chains. The box plots within the violin plots show the median of data. Computation times are recorded until 10,000 messages are processed. The wideness bars shows the density of computation time in the specific range. As shown in Fig. 9, for all three transitions the system performance is impeded by the use of MDebugger. Using MDebugger, the majority of the ProcessResponse message are processed within 0.28 to 0.41ms, with an average time of 0.35ms and a median time of 0.33ms respectively, which is close to the processing time when system in normal mode (average and median times of 0.29 and 0.27ms respectively). The overhead for ProcessResponse transition is within the range of microseconds and therefore negligible. The overhead is similar for *RequestReply* and *SendKeepAlive* messages. While the median and average of computation time for *RequestReply* is 48.26ms and 48.45ms using our approach respectively, the median and average in normal mode are 47.41 and 48.35ms. For the SendKeepAlive transitions, the median and average using our approach are 0.07ms and 0.08ms respectively and is 0.01ms for both in normal mode. In summary, we can argue that for each transition, the overhead of our approach is small, which is quite acceptable for many RTE systems.

We also measured the overall processing time of 10,000 messages. The overall processing time is 507 seconds under normal mode and 519 seconds using our approach. It represents an added overhead of 2.31%. This overhead is quite low and validates our first hypothesis. It might be worth mentioning that the added execution time for each transition appears to be constant, meaning that this result may vary depending on the complexity of the system to debug.

6. RELATED WORK

A certain number of related work are relevant for model-based debugging. They can be divided up into three cate-



(a) RequestReply (b) SendKeepAlive (c) ProcessResponse

Figure 9: Processing Time of Transitions and Overall Processing Time of 10,000 Messages

gories: debugging through simulation and model interpretation, off-line debugging via trace analyses, and live debugging on a target platform.

Debugging via simulation is done by interpreting the models at design-time. Debugging features, such as setting breakpoints and stepping over the execution are usually supported. Simulation and model interpretation are supported by several tools, e.g., Matlab StateFlow [42], AF3 [13], xtUML [5] and YAKINDU [19]. However, while simulation is necessary and useful for finding bugs at early design-time based on functional requirements, it is insufficient for debugging RTE systems where bugs can be caused by the sensitiveness of the system to timing constraints [18, 44, 35, 15]. Such bugs can be found using complex representations of the systems where timing constraints and the environment are modeled, but this requires more sophisticated environments to interpret the model. Another proper solution is to run the system on the target platform, allowing users to perform off-line or live debugging.

Trace analysis techniques are a mean to support debugging capabilities on target platforms where debugging tools cannot be embedded due to the limited resources the platform provide. They rely on the off-line interpretation of useful traces generated by the platform. In order to generate these traces, one can rely on existing probes embedded on the target platform, or on customizing the code generator in order to instrument the code. An example of using probes is described in [8], where traces are generated by an HW multi-core platform and are used to estimate the power consumption of a HW node. Examples of existing work and MDE tools supporting trace analyses via code instrumentation include [18, 41, 21, 20, 14]. For instance, Iyengar et al. [21, 20, 15] propose an optimized model-based debugging technique for RTE systems with limited memory. They use a monitor on the target platform to collect the generated traces and a debugger (executed on a host with sufficient memory) to analyse the traces off-line and to display results on the model elements. Das et al. [11] propose a configurable tracing tool based on LTTng. They rely on code instrumentation in order to produce tracepoints useful for LTTng. The tool supports timing constraint analysis via trace replay.

The main disadvantage of this method is that the connection between target platform and debugger is one way, which does not provide the control on the execution which is required for rich debugging features such as stepping over the execution, setting breakpoints, or changing attributes. To the best of our knowledge, only the work described in [14] provides a limited support for controlling the execution via signal injection. Also, the different work still requires to maintain a mapping is between the source code and the model elements. It can be addressed by instrumentation [11] or stored in mapping files [21, 20].

Live debugging on target platforms is the richest debugging service and is our main focus in this research. Despite its importance / significance, only a few tools, e.g., ProgramDev [33], IBM RSARTE [2], and Timing Architects [43] supports live debugging capabilities. However, they rely on existing program debuggers such as GDB [34] and Universal Debug Engine [30], and suffer from the different limitation discussed in Section 2. Also, some research tried to addressed the model-level debugging and facilitate the mapping in RTE and other domains which all of them are using traditional approaches [12, 39, 31, 7, 46, 15]. Martin et al. [39, 31, 7] develop an integrated debugging plug-in for equation-based models created by Modelica [26]. They use GDB [34] to debug the generated C code, and then map the debugging results to the equation-based model element. Graf et al. [15] present a framework for dynamic mapping from binary- to the model-level.

7. CONCLUSION

In this paper, we formalised and implemented a model-level debugging approach for RTE systems. Compared with existing approaches that rely on program debuggers to work, our approach relies on model instrumentation techniques. To do so, we formulated and applied the necessary instrumentation at the model-level, bringing debugging capabilities to the model itself. As a consequence, our approach does not require any additional instrumentation at code-level. It is therefore more generic and portable, meaning that the debugger can support a range of code generators, target languages, and HW platforms without change. Along with the approach, we implemented a model-level debugger called *MDebugger*. It supports most of the common debugging features, and can be used directly in command line, or via a graphical debugging interface we developed in Eclipse. We also conducted an experiment to assess the applicability or our approach. The experiment showed that the size overhead of the generated binary files is comparable to other approaches and the use of our implementation slightly impacts the performance of the system under debugging.

8. REFERENCES

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

[2] IBM Rational Software Architect RealTime Edition, v9.5.0 Product Documentation. http://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0, 2015.

[3] [Blinded for review]. In *Model-Driven Engineering Languages and Systems (MODELS'16)*, 2016.

[4] [Blinded for review]. Submitted to Model-Driven Engineering Languages and Systems (MODELS'17), 2017.

[5] xtUML - eXecutable Translatable UML with BridgePoint. https://xtuml.org, accessed July 19, 2016.

[6] R. Ahmadi, N. Hili, L. Jweda, N. Das, S. Ganesan, and J. Dingel. Run-time Monitoring of a Rover: MDE Research with Open Source Software and Low-cost Hardware. In *Workshop on Open Source for Model-Driven Engineering (OSS4MDE'16)*, 2016.

[7] A. Asghar, A. Pop, M. Sjölund, and P. Fritzson. Efficient Debugging of Large Algorithmic Modelica Applications. *IFAC Proceedings Volumes*, 45(2):1087–1090, 2012.

[8] Y. B. Atitallah, J. Mottin, N. Hili, T. Ducroux, and G. Godet-Bar. A Power Consumption Estimation Approach for Embedded Software Design Using Trace Analysis. In *41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA'15)*, pages 61–68. IEEE, 2015.

[9] C. Atkinson and T. Kuhne. Model-Driven Development: a Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[10] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt. Adaptive Failover for Real-Time Middleware with Passive Replication. In *15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 118–127. IEEE, 2009.

[11] N. Das, S. Ganesan, L. Jweda, M. Bagherzadeh, N. Hili, and J. Dingel. Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM, 2016.

[12] D. Dotan and A. Kirshin. Debugging and Testing Behavioral UML Models. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 838–839. ACM, 2007.

[13] FORTISS GMBH. AF3. http://af3.fortiss.org/, accessed July 19, 2016.

[14] E. Gery, D. Harel, and E. Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. In *International Conference on Integrated Formal Methods*, pages 1–10. Springer, 2002.

[15] P. Graf and K. D. Muller-Glaser. Dynamic Mapping of Runtime Information Models for Debugging Embedded Software. In *Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006*, pages 3–9. IEEE, 2006.

[16] S. Graf, I. Ober, and I. Ober. A Real-Time Profile for UML. *International Journal on Software Tools for Technology Transfer*, 2006.

[17] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *Computer*, 30(4):68–74, 1997.

[18] W. Haberl, M. Herrmannsdoerfer, J. Birke, and U. Baumgarten. Model-Level Debugging of Embedded Real-Time Systems. In *10th international conference on Computer and Information Technology (CIT'10)*, pages 1887–1894. IEEE, 2010.

[19] ITEMIS AG. Yakindu StateChart Tools. https://www.itemis.com/en/yakindu/statechart-tools, accessed July 19, 2016.

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

[21] P. Iyenghar, C. Westerkamp, J. Wuebbelmann, and E. Pulvermueller. A Model Based Approach for Debugging Embedded Systems in Real-Time. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 69–78, New York, NY, USA, 2010. ACM.

[22] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[23] P. Maes. Concepts and Experiments in Computational Reflection. *SIGPLAN Not.*, 22(12):147–155, Dec. 1987.

[24] J. Magee and J. Kramer. *State Models and Java Programs*. Wiley, 1999.

[25] S. Migdalskiy. Physics Engine Development - Steam. http://media.steampowered.com/apps/valve/2014/Sergiy_Migdalskiy_Debugging_Techniques.pdf, 2014. [Online; accessed 21-Feb-2007].

[26] Modelica Association. The Modelica Language Specification Version 3.2 revision 2, July 30th 2013. https://www.modelica.org/documents/ModelicaSpec32Revision2.pdf, accessed August 5, 2016.

[27] OMG. UML Profile for Schedulability, Performance, and Time Specification, 2005.

[28] Oracle and/or its affiliates. Trail: The Reflection API. https://docs.oracle.com/javase/tutorial/reflect/, 2017. [Online; accessed 26-Jan-2017].

[29] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[30] PLS Programmierbare Logik & Systeme GmbH. UDE Microcontroller Debugger. https://www.pls-mc.com/universal-debug-engine-ude-and-microcontroller-debugger-for-aurix-arm-7911-xe166xc2000-xscale-sh-2a_c166st10/universal_debug_engine-a-802.html, 2016. [Online; accessed 21-Feb-2007].

[31] A. Pop, M. Sjölund, A. Asghar, P. Fritzson, and F. Casella. Static and Dynamic Debugging of

Modelica Models. In *Proceedings of the 9th International MODELICA Conference; Munich; Germany*, number 076, pages 443–454. Linköping University Electronic Press, 2012.

[32] E. Posse and J. Dingel. An Executable Formal Semantics for UML-RT. *Software & Systems Modeling*, 15(1):179–217, Feb. 2016.

[33] PRAGMADEV SARL. ProgmaDev. http://www.pragmadev.com, accessed July 19, 2016.

[34] S. S. Richard Stallman, Roland Pesch. Debugging with GDB. http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz, accessed August 5, 2016.

[35] T. Schwalb, P. Graf, and K. D. Müller-Glaser. Monitoring Executions on Reconfigurable Hardware at Model Level. In *5th International MODELS Workshop on Models@runtime*, 2010.

[36] B. Selic. Using UML for modeling complex real-time systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 250–260, 1998.

[37] B. Selic. What Will It Take? A View on Adoption of Model-based Methods in Practice. *Software & Systems Modeling*, 11(4):513–526, Oct. 2012.

[38] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*, volume 2. John Wiley & Sons New York, 1994.

[39] M. Sjölund, F. Casella, A. Pop, A. Asghar, P. Fritzson, W. Braun, L. Ochel, and B. Bachmann. Integrated Debugging of Equation-Based Models. In *Proceedings of the 10 th International Modelica Conference; Lund; Sweden*, number 096, pages 195–204. Linköping University Electronic Press, 2014.

[40] W. Swartout and R. Balzer. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, 1982.

[41] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 1–8. IEEE, 2003.

[42] The MathWorks, Inc. Stateflow - MATLAB & Simulink. http://www.mathworks.com/products/stateflow/, accessed July 19, 2016.

[43] Timing-Architects Embedded Systems GmbH. Timing Architects. http://www.timing-architects.com/, accessed July 19, 2016.

[44] Willert Software Tools GmbH. Embedded UML Target Debugger. http://www.willert.de/assets/Datenblaetter/DatS-Embedded-UML-Target-Debugger-V9.0-EN-2016.pdf, 2016. [Online; accessed 21-Feb-2007].

[45] H. Wu, J. Gray, and M. Mernik. Grammar-Driven Generation of Domain-Specific Language Debuggers. *Software-Practice and Experience*, 38(10):1073, 2008.

[46] K. Zeng, Y. Guo, and C. K. Angelov. Graphical model debugger framework for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 87–92. European Design and Automation Association, 2010.