Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation

Nondini Das, Suchita Ganesan, Leo Jweda,
Mojtaba Bagherzadeh, Nicolas Hili, Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
{ndas, ganesan, juwaidah, mojtaba, hili, dingel}@cs.queensu.ca

ABSTRACT

This paper presents a vision that allows the combined use of model-driven engineering, run-time monitoring, and animation for the development and analysis of components in real-time embedded systems. Key building block in the tool environment supporting this vision is a highly-customizable code generation process. Customization is performed via a configuration specification which describes the ways in which input is provided to the component, the ways in which run-time execution information can be observed, and how these observations drive animation tools. The environment is envisioned to be suitable for different activities ranging from quality assurance to supporting certification, teaching, and outreach and will be built exclusively with open source tools to increase impact. A preliminary prototype implementation is described.

1. INTRODUCTION AND MOTIVATION

A component within a modern real-time embedded system (RTE) typically has to function in the context of possibly a large number of other components which produce input or consume output. For instance, modern cars have over a 100 Electronic Control Units (ECUs) which, e.g., regulate the car's braking systems by using input about the brake position, the vehicle speed, the speed of each wheel, the drive-mode (e.g., 4-wheel drive enabled), etc. In the telecommunications domain, a Private Branch Exchange (PBX) connects phones at a business site with each other and to the public telephone network. Modern PBXs handle many kinds of requests from external and internal phones including the setup of calls, conferencing, voice messaging, etc. In banking, the controller in an Automated Teller Machine (ATM) receives input from, e.g., the card reader, the keypad, the money slot, the host computer, and the bank computer, while sending output to the display, the money slot, the speakers, and the computers.

MODELS '16, October 02-07, 2016, Saint-Malo, France
 ISBN 978-1-4503-4321-3/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2976767.2976781

We will distinguish between two kinds of activities involving embedded systems, one occurring during development, the other typically occurring after development: quality assurance (QA) and communication.

Quality assurance: To ensure proper functioning of these embedded systems, quality assurance activities must check a range of properties including basic correctness properties (e.g., "if the withdrawal has been authorized, the requested amount is dispensed and recorded correctly"), performance properties (e.g., "responses to withdrawal requests should be given within 5 seconds"), and security properties (e.g., "only properly authorized users can access an account"). To check these properties, developers can use different techniques: (1) *execution*, which allows for initial, low-volume sanity checks; (2) *testing*, to ascertain that specific inputs trigger expected outputs; and (3) *monitoring and simulation*, in which the component must exhibit correct behaviour with respect to certain properties during a typically longer period of time under operating conditions that might approximate reality with different degrees of fidelity.

Note that the level of detail at which the component's behaviour is monitored and the accuracy and fidelity of the operating conditions must support the purpose of the simulation, i.e., the conclusions that are to be drawn from it. E.g., a simulation of an ATM controller to determine if it returns the bank card in less than two seconds after three failed pin input attempts must allow for the time delay to be observable and measured and must present the controller with a sufficient total number of failed pin input attempts.

All of these techniques can benefit from (4) *animation and visualization* in which relevant, possibly aggregated observations of the execution are displayed in a suitable format.

Finally, to locate and remove bugs, powerful *debugging* support is required, ideally on both the model- and the code-level.

Communication: Communication activities typically succeed successful development. Their purpose is to show relevant aspects of the structure and behaviour of the system to different audiences with the intent to educate and attract, train, provide evidence, or even to sell. For instance, in the context of our ATM example, suitably chosen communication activities might be useful to (1) *educate* students on different levels about software engineering or embedded systems and attract them to the area ("outreach"), (2) *train* maintenance operators, (3) *help convince* certification authorities, or (4) potential buyers. A central motivation be-

hind our work is the observation that communication activities and QA activities differ mostly in intent and audience, but otherwise have a lot in common. For instance, certification authorities and buyers might be impressed by software testing that exercise the system under input conditions that they consider realistic and that allow observations, perhaps using suitable technical animation tools, about system executions that suggest correct operation to them; similarly, high school students might be able to understand and appreciate key concepts of state machines (or networks) with a simulation in which states (or messages) are animated in an audience-appropriate way, perhaps even with "flashy" 3D animation tools such as Unity [9] or Minecraft [5].

Our vision.

We hypothesize that, given some RTE component C, the QA and communication activities described above can be supported by a tool environment that allows the use of a wide range of different (1) *input sources* that can generate inputs for C at different levels of detail and accuracy, (2) *notions of observability* of executions of C and observation consumers, and (3) *animation techniques and tools*. Further, we believe that such a tool environment can be built in the context of Model-Driven Engineering (MDE) via a highly customizable code generation process. Apart from the models describing C, this process would take as input a user-provided *configuration specification* describing the desired ways in which input is to be given to C and the resulting executions of C can be observed, and how these observations can drive different animation tools. The result of the code generation would be executable code allowing the execution and monitoring of C in the specified context and at the specified level of abstraction. Moreover, QA activities are further supported via *integrated debugging* allowing the developer to switch seamlessly between debugging activities on the model and the code level.

While not strictly necessary, we will assume that the development is based on Unified Modeling Language for Real-Time (UML-RT) [29], a language specifically designed for RTE systems with soft real-time constraints. UML-RT has a long, successful track record of application and tool support, via, e.g., IBM RSA-RTE [10] and PapyrusRT [6]. Moreover, to maximize the impact of our work, we propose to develop our tool environment entirely with *open source* tools including PapyrusRT for modeling and code generation, LTTng for m onitoring [4], and TraceCompass for trace animation [8].

Structure of the paper.

Related work is discussed in the next section. Section 3 describes the vision and its supporting tool environment in more detail. Section 4 sketches an initial prototype implementation. Section 5 concludes.

2. RELATED WORK

Our work touches on many different areas of research with testing, monitoring, simulation, animation, and visualization in the context of models and code, and model-level execution and debugging probably being the most relevant ones. We are not aware of any work on integrated debugging.

An important group of related work are existing proposals for model-level monitoring, simulation, and animation of UML models. The TOPCASED project reported on an early effort with a focus on simulation and animation of state machines [17]. More recently, different model execution environments such as Moka and Moliz have been proposed [13, 24]. In both cases, the environment contains a customizable model execution engine based on fUML [12], and support for breakpoints and execution animation. Our work differs in that it focuses on monitoring the execution of the *code generated from the model*, rather than on executing the model via an execution engine. This will make our work more suitable for the analysis of properties that can only be observed in the execution environment the developed software is to be shipped in such as properties involving performance and the use of hardware resources. Thus, our work addresses different concerns and is complementary.

Many other kinds of modeling formalisms have been used for simulation-based performance analysis including Petri nets, queuing networks, and DEVS [30]. Each of these formalisms is supported by a range of tools [2, 3, 7]. The Palladio approach allows the analysis and simulation of architectural models with respect to, e.g., performance and reliability [15]. To keep the analyses tractable, models in Palladio are not complete descriptions of the component's behaviour as in our work, but rather allow for a static description of the component's internal state only. None of these efforts are usable for MDE with UML.

Work on software testing such as Apache JMeter [1] also is relevant. As in our vision, the tool builds a complete, highly customizable testing environment with animation capabilities, however, not from models of an RTE system, but from existing code (in, e.g., Java). Nonetheless, we might be able to draw inspiration from these tools, e.g., with respect to the ways in which software tests can be set up.

On the industrial side, the development of electronic control systems in, e.g., the automotive and aerospace domains, is supported by a range of sophisticated, commercial modeling, monitoring, simulation, and visualization capabilities in tools offered by, e.g., IBM Rational (Statemate, RSA-RTE), National Instruments (LabVIEW), MathWorks (Simulink), or dSPACE. While our work is partially inspired by these tools, there are fundamental differences: our tool environment will be completely open source, be applicable to a wider range of component-based systems (e.g., with distribution), and support higher degrees of customization (because, e.g., monitoring tools, input sources, and animation tools do not need to be part of a particular tool set). Also, with the exception of Statemate and RSA-RTE, the above tools focus on systems with continuous, rather than discrete, control.

To conclude, our vision differs from existing work in several, fundamental ways.

3. OUR VISION

Traditional MDE processes for RTE systems typically involve activities such as design of the system, code generation, and execution of the generated code on the target platform and in the context of possibly a range of other components that the system interacts with. We posit that the key concept allowing the construction of a range of such execution scenarios supporting different purposes lies in the customization of the code generation process. We envision for this customizable code generation process to form the heart of a multi-purpose infrastructure for RTE system design that supports integrated debugging, monitoring, testing, animation, and incremental development.

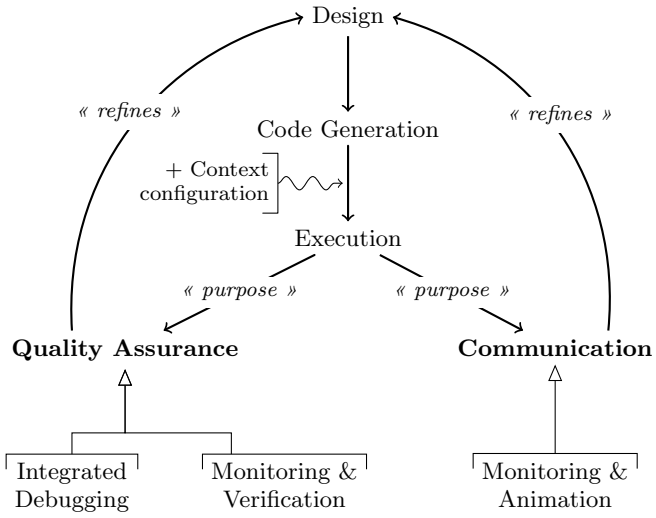

Figure 1: Overview of Use of Infrastructure

3.1 Infrastructure Overview

The use of the infrastructure is depicted in Fig. 1. Activities include *design*, *code generation*, *deployment*, and *execution* on a specific target platform and context. Executions can be used for two different purposes: *quality assurance (QA)* and *communication*. The infrastructure is to be used iteratively and allows for incremental development and refinement of the design; QA activities would form the primary source of feedback, but it is conceivable that communication activities also trigger a change in the design.

The use of models for the *design* facilitates the expression of the system and its behaviour at different levels of abstraction. The suitability of the level of detail in a design depends on what the generated code is to be used for. E.g., a high-level design for less technical audiences might ignore some low-level details pertaining, e.g., to the way a component computes its output, or how time can influence the execution; conversely, if the generation of shippable code is the goal, then the design needs to include all details.

Models created during the design phase are inputs to the code generator together with a *context configuration* which specifies any additional components that the generated code is to run in concert with and how the code interacts with them. For instance, the configuration may specify (1) to what extent the additional components are able to observe the code during execution (e.g., which externally visible events the execution might generate), (2) how this information is collected (e.g., the monitoring tool used), (3) how this information is fed to other components as input (e.g., to drive animations), and (4) which components can provide input to the code and how that input is delivered. The context configuration is described in more detail in Section 3.2.

After generation, the code is compiled, deployed and executed on the target platform. The code will interact with additional components by generating and consuming events; output events may be collected, displayed, analyzed, or drive animations; input events may be generated from different sources including a user, a sensor, or a simulation tool based on random choice, historical data, or a probability distribution. All of these choices are specified in the context configuration and determine the suitability of the execution for a range of different activities and purposes.

3.1.1 Integrated Debugging

Debugging real-time embedded systems is hard [21]. Model-level execution and debugging is a topic of ongoing research and some tools such as Moka are beginning to emerge [28]. While undoubtedly useful, model-level execution and debugging can only represent the "first line of defense" in the fight against bugs. More precisely, it is well suited to uncover faults in the basic functionality of the model, but it cannot detect requirement violations due to the way the system uses computational resources such as violations of real-time or memory constraints, because these issues only manifest themselves when the generated code is run in its intended execution environment and on the intended platform. After detecting these kinds of violations via code-level monitoring as supported by our envisioned infrastructure, inspection and debugging on the code level is necessary as "second line of defense".

However, instead to model- and code-level debugging to be disjoint activities implemented by separate tools unable to share information, we envision integrated debugging, which allows the developer to perform debugging activities on either level and which automatically "lifts" code-level debugging information (e.g., stack traces) onto the model-level and displays it there in an appropriate form (e.g., as state chains). To achieve this integration, relevant code-level elements must be traceable to their model-level counterparts.

3.1.2 Monitoring & Verification for QA

Run-time monitoring and testing are important activities in software development [14, 18, 22, 26]. They can help designers validate model quality under different "operating conditions", and detect and debug faults in the models by observing the code execution on the target platform. Observation could aim at high-level correctness properties (e.g., *"it is never the case that the statemachines of these two components are in these two states at the same time"*), or at low-level resource consumption or timing properties (e.g., *"after a request has been received, a response is issued within 2 seconds"*). Observing and tracking high-level, non-timing-related information such as the currently active statemachine state is not too difficult. Similarly, most operating systems support performance counters that can be used to determine the consumption of resources such as heap-space or network bandwidth; examples include `perf` in Linux and `perfmon` in Windows. However, the collection of meaningful timing information that allows the validation of real-time constraints is a lot more difficult and real-time monitoring tools should satisfy specific requirements [25]. More precisely, the tool must incur low overhead to limit the impact on the execution of the observed system; it should be non-obtrusive and be able to operate as an observer alongside the system execution; moreover, to support resource-constrained systems, the monitoring tools' memory footprint should be as small as possible. While remote monitoring, in which events are sent over a communication link to the monitor, is useful for tracking high-level, non-timing related information, its utility for timing analyses is limited to the overhead caused by the communication; local monitoring is much more suitable here.

3.1.3 *Monitoring & Animation for Communication*

As before, the suitability of an animation depends on the purpose and the audience: (1) Model execution tools such as Moka [28] allow for model-level animation of the system's execution and appear most suitable for a more technically inclined audience hoping to understand the inner workings of the system across relatively short executions with limited environment interactions. They might also serve to illustrate the use of statemachines to describe RTE systems in training and teaching contexts. (2) However, to show the system's behaviour across longer-running executions involving perhaps complex environment interactions that approximate realistic operating conditions with a high degree of fidelity to an audience of, e.g., potential buyers and regulators, animation tools capable of, e.g., aggregating data and displaying it graphically would be more suitable. (3) Finally, to maximize appeal and "wow"-factor for, e.g., outreach and recruitment events, the use of, e.g., 3D-animation tools such as Unity or Minecraft [5,9] allowing users to visualize a system and possibly even interact with it via avatars might be the best choice.

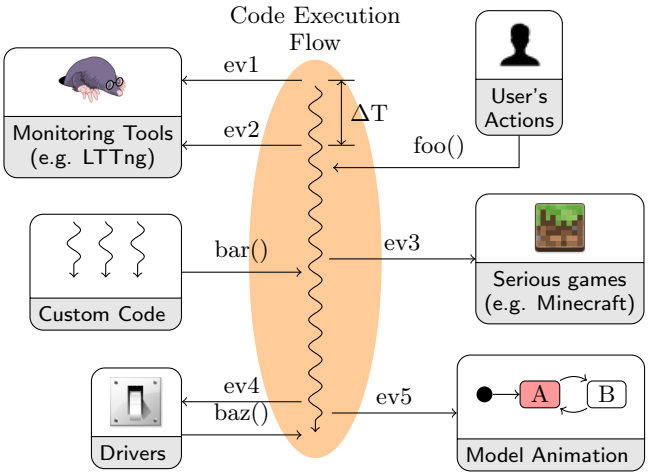

Figure 2: Interaction between Code and Context

3.2 The Context Configuration Model

As discussed above, the generated code usually has to interact with a number of different components. All these components constitute the *context of execution* of the system. Fig. 2 provides an illustration of how the running code might interact with its context. It may include devices, debugging, monitoring, testing, or animation tools, or drivers to access external devices and sources of input. We note that all components have one thing in common: They can either consume or produce events, or both. Consequently, the code generation process has to be adapted so that the generated code can interact with them.

To this end, we define a *context configuration model*. It lists all the components the generated code is to interact with and which events they can produce or consume (left and right sides in Fig. 2). The presence of a component in the configuration would not only impact the shape of the code generated from the model, but may also cause the generation of additional artifacts the component requires.

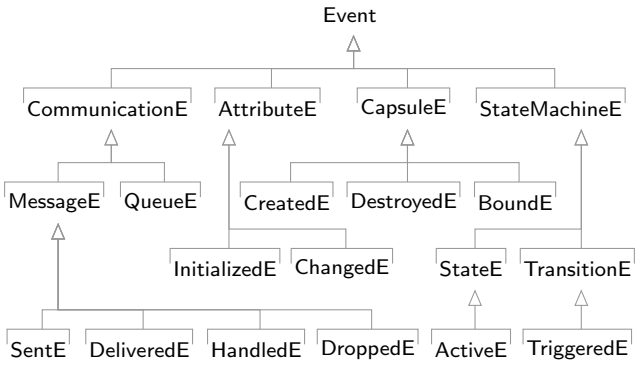

Figure 3: Taxonomy of Monitorable Events

To facilitate the specification of the interaction between the code and its context, we define a hierarchy of monitorable events (Fig. 3). The hierarchy is currently specific to UML-RT, but could be adapted to other component-based modeling languages. It distinguishes four main categories of monitorable events, namely *CommunicationE*, *AttributeE*, *CapsuleE* and *StateMachineE* events each of which with subcategories. For example, events occurring in a statemachine model are related to states and transitions (*StateE* and *TransitionE*). Sub-events are *ActiveE* and *TriggeredE*, which indicate, respectively, a change in the currently active statemachine state and the triggering of a transition.

4. IMPLEMENTATION

Fig. 4 illustrates our current prototype implementation of the infrastructure and how the different activities discussed in Fig. 1 are supported through open source tools. For design and code generation, PapyrusRT [6] is used. PapyrusRT is built on the Papyrus modeling environment for UML 2.5 [11, 23]. It provides a complete environment for UML-RT models design, C++ code generation capabilities and a Run-Time Support (RTS) library for Linux platforms.

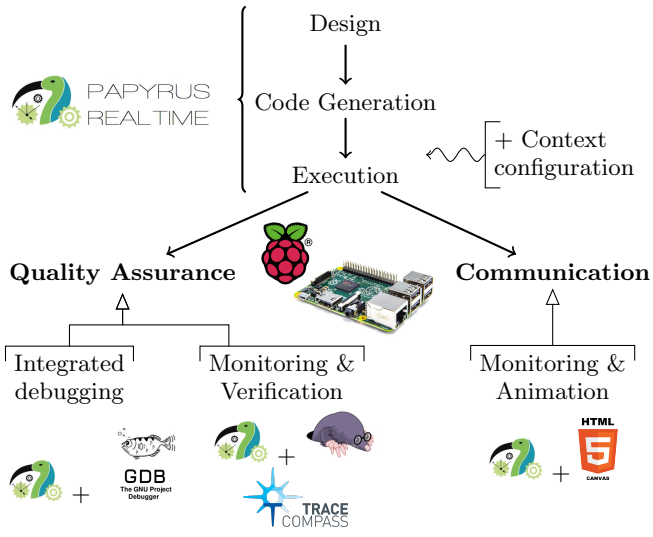

Figure 4: Architecture Overview

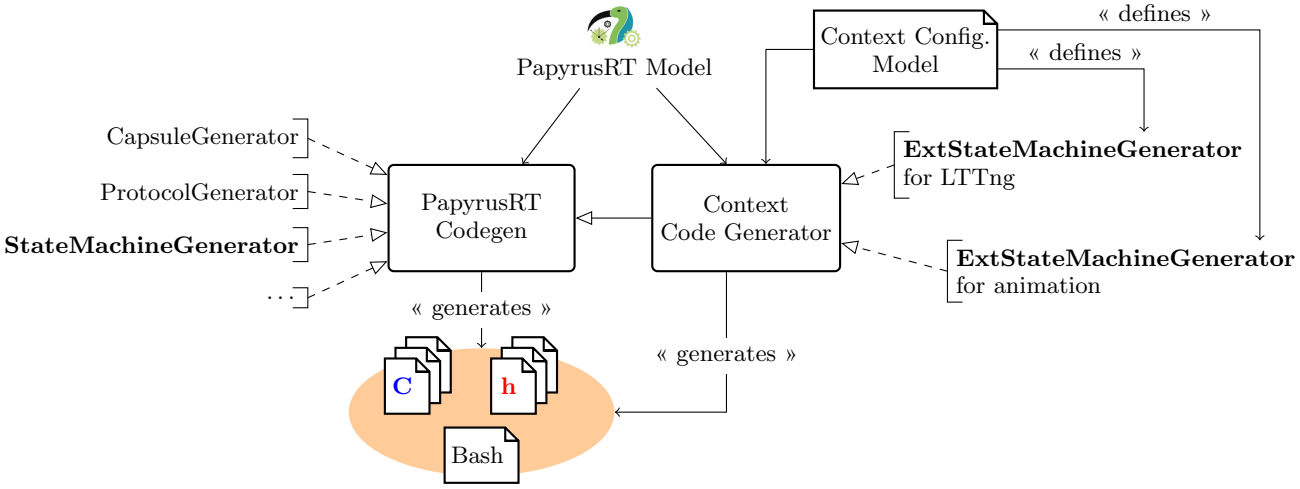

Figure 5: Papyrus Code Generator Extension

4.1 Extension of the PapyrusRT Codegen

A central objective of the PapyrusRT Codegen project was to design a code generator for UML-RT models that can be easily extended. The architecture of the generator is described in [27] and our work leverages the extensibility of this architecture greatly. On the left-hand side, Fig. 5 sketches the structure and use of the standard generator and shows the collection of independent generators that are registered with the PapyrusRT Codegen plugin. Each generator focuses on the generation of a specific high-level element type in UML-RT models. For example, the *CapsuleGenerator* generates C++ artifacts related to the definition of a capsule with its ports, attributes, and internal parts. The *StateMachineGenerator* generates code corresponding to a UML-RT statemachine.

The right-hand side of Fig. 5 shows how our implementation subclasses standard generators with customized generators that are registered using a specific Eclipse extension point. To increase generality, we implemented a *Context Code Generator* that takes the *context configuration model* as an input and defines which specialized code generator should be used to subclass a standard one. For example, we defined two customized statemachine code generators: one to support monitoring with LTTng (described in Section 4.3) and another one for web-based animation (Section 4.4). Each generator contributes the code required by the additional components.

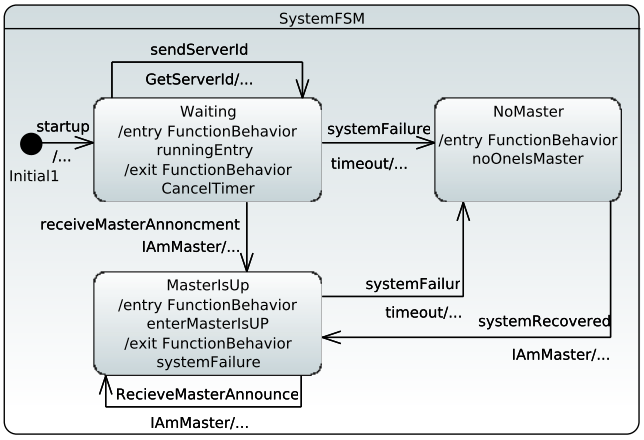

Figure 6: The Failover System

4.2 Running Example: the Failover System

To validate our implementation, we modeled a *Failover* system and executed the generated code on a Raspberry PI platform. The Failover system uses passive replication, a key technique to maximize availability in distributed real-time systems. In passive replication, only one server (called *master*) handles all client requests, while the backup servers are largely idle, except for, e.g., receiving state updates from the master. If the master fails, a failover is triggered and one of the backups becomes the new master.

Fig. 6 shows the behavior of the root capsule of a system containing two or more servers. During the initialization, the system is idle, waiting for a server to proclaim itself to be the master. Then, the master server has to regularly notify the system that it is alive. If no master is discovered during the initialization of the system, or if the current master is not available for a specific period, the system enters a failure mode and waits for another server to become the master.

4.3 Timing Constraint Validation using LTTng

So far, our implementation supports timing constraint validation (e.g., "if the master fails, selecting a backup server to be the new master should be done within 2 seconds"). Future extensions could allow for additional analyses such as CPU performance evaluation.

Fig. 7 summarizes the main monitoring steps. Using UML-RT models designed with PapyrusRT, trace monitoring is performed on the target platform using Linux Trace Toolkit Next Generation (LTTng), an open source tracing framework for Linux [4,20]. Traces are collected to validate timing constraints and further refine the design.

4.3.1 Monitoring Configuration

The first step consists of two activities: 1) defining with respect to which events shown in Fig. 3 the system is to be monitored; and 1') specifying timing constraints that must

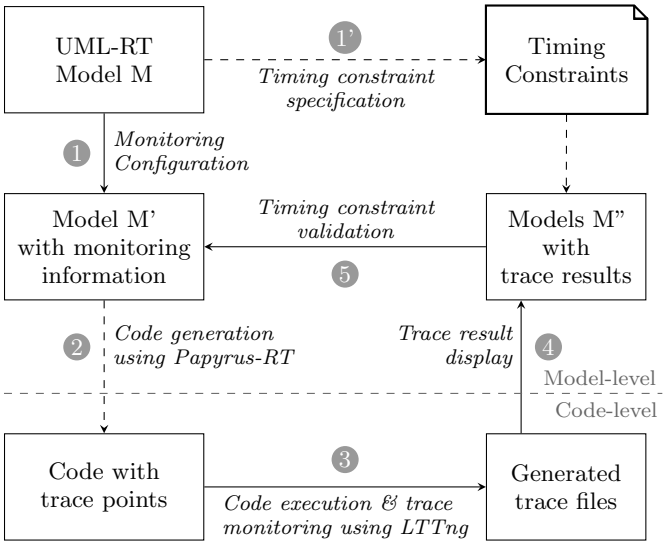

Figure 7: Time Constraint Monitoring Overview

be validated over the execution (e.g., going from state `S1` to state `S2` must not exceed 2 seconds). For 1), a UML-RT model M is annotated with monitoring information. For the moment, our prototype only supports statemachine events. Information we can currently monitor includes when the system enters a specific state, when a transition is triggered, and the amount of time between two user-selected events. Selected elements are marked for the code generator using a "LTTngElement" stereotype we defined (c.f. Fig. 8). For 1'), timing constraints are specified in a separate file which is used to validate the collected traces.

4.3.2 Code Generation with LTTng trace points

LTTng is well suited for monitoring timing constraints as it is efficient, non-obtrusive, and allows for both local and remote monitoring. Local monitoring has low overhead and is thus more suitable for timing analyses. In addition, LTTng relies on the Common Trace Format (CTF) [19], an optimized format for producing and analyzing large amounts of data, to produce trace files with a low overhead [20].

From the annotated model M' code is generated. It includes trace points required by LTTng to monitor the execution. To this end, we extended the standard code generator of PapyrusRT as described in Section 4.1. Alongside the code, a script containing all the LTTng commands necessary for tracing and which can be run to monitor the execution is generated.

Our implementation is fully automated, meaning that the generation of the code, the execution of LTTng on the target platform, the production of traces and their display in PapyrusRT do not require user interaction.

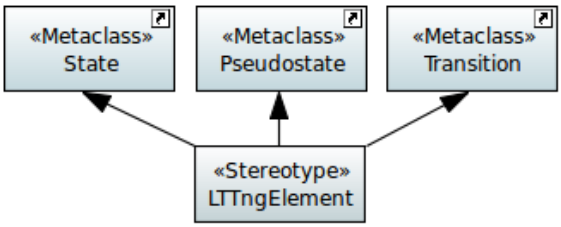

Figure 8: Monitoring Profile

4.3.3 Tracing using LTTng and Trace Display

We developed an Eclipse-based plugin to display and analyze traces. The goal is to display the trace in both a textual as well as a graphical format[1]. Fig. 9 depicts an Eclipse view comprising three columns showing trace files, trace details[2], and four buttons for user interaction. The plugin allows users to select a model and the plugin will then automatically list any trace files associated with that model. Upon selecting a trace file produced by LTTng on the embedded platform, the Trace Compass CTF parser is used for parsing all the traces. The most significant feature of an LTTng trace is that it contains timestamp information for every monitored event [16, 20]. In our plugin, the timestamp information is displayed along with the string field of all traced events in the left part of Fig. 9. Users can step through the trace in the model by using a "Display" and a "Step" button, shown on the right-hand side of Fig. 9; the currently active state and transitions taken are highlighted in the model and the textual view. Finally, timing constraints can be validated on a trace by using a "Validate Time" button on the right-hand side of the view (cf. Fig. 9).

[1]A screen cast of the tracing of the Failover model with LTTng is available at https://youtu.be/h2uNLHg001I

[2]Fig. 9 shows a trace file resulting from the execution of the Failover model (cf. Fig.6).

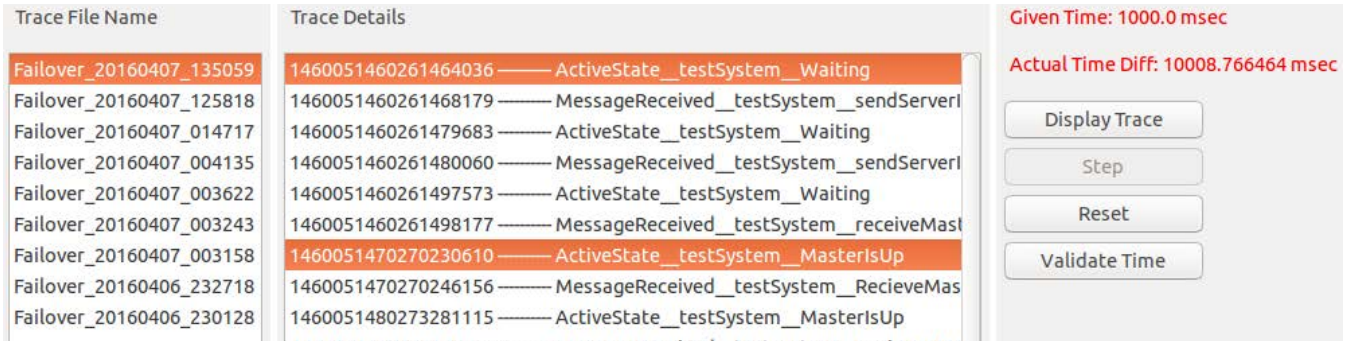

Figure 9: Trace Display

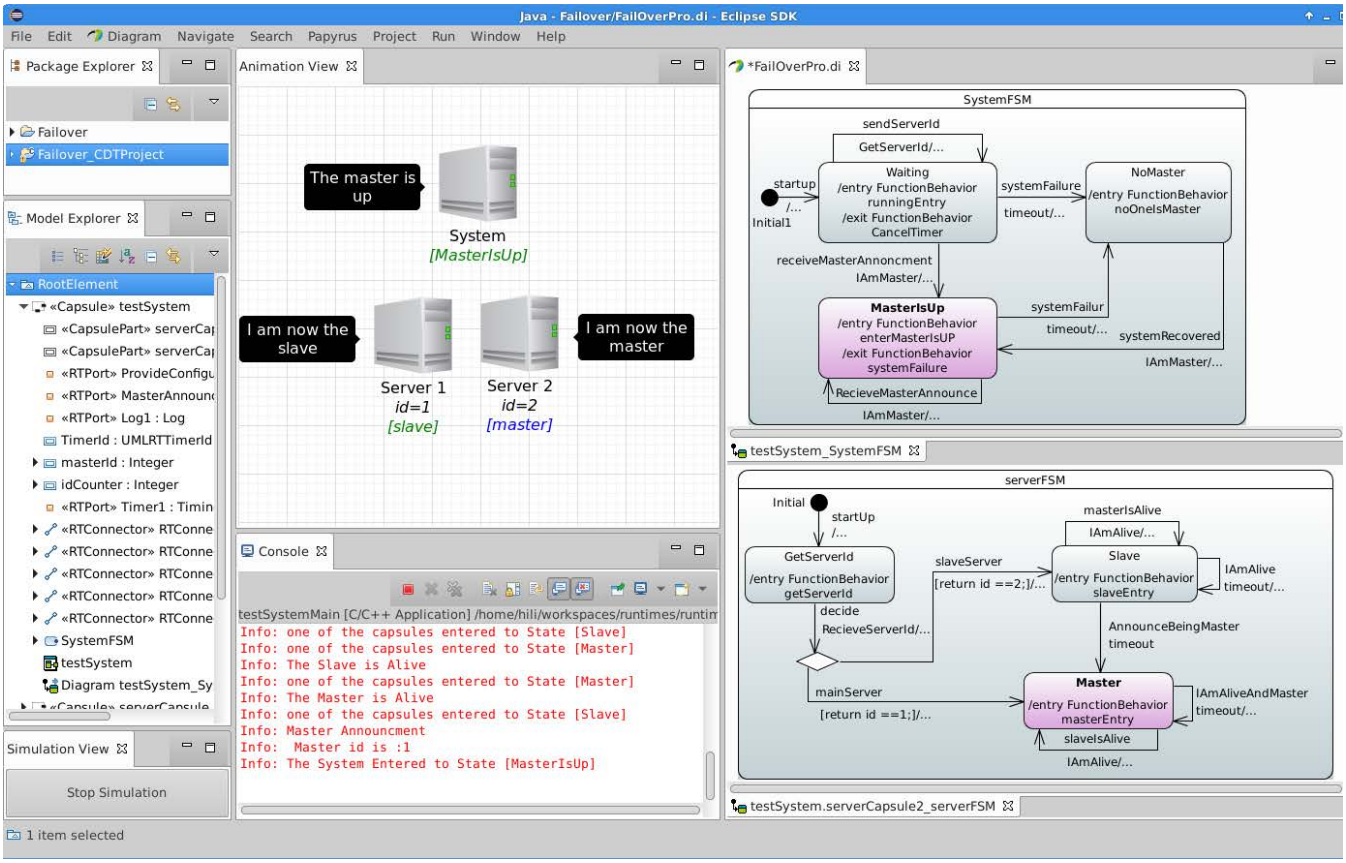

Figure 10: Live Monitoring & Web-Based Animation Environment for the UML-RT System and its Surrounding Context

4.4 Live Monitoring & Web-based Animation

We also implemented a live monitoring and animation tool which interacts with PapyrusRT to animate the execution of the generated C++ code at run-time on the statemachines the code was generated from. Our implementation differs from existing simulation tools such as Moka [28], which interpret the model using executable action semantics like ALF or fUML. In our case, statemachine model monitoring is directly driven by the code running on the embedded platform. Therefore, it directly monitors the actual execution of the system. To monitor the execution flow from Eclipse, we implemented a bi-directional socket communication.

In addition to the statemachine model monitoring, we also implemented an animation engine to animate the execution of the model on a user-friendly, web-based environment, using JavaScript and HTML5 Canvas. In terms of Fig. 2, both the model and the environment animation engines are part of the *execution context* of the platform. Consequently, they can be plugged in to monitor the code execution flow on the real platform.

Fig. 10 illustrates the monitoring of the *Failover* model. The animation environment depicts the system and two servers. Each server is identified by an identifier and can become either the main server or the slave. The two statemachines on the right-hand side of the animation environment show the behavior of the system and of the master server[3].

[3]A screen cast of the animation of the Failover model in our tool is available at https://youtu.be/p5emJoswWpk

4.5 Discussion and Future Directions

How to best realize and support the customization is a research topic. Leveraging code generation customization for such a broad range of purposes and tools appears novel. At model-level, it allows for the support of customizable communications with external tools. At code-level, several questions have to be answered in order to choose the right techniques in order to implement a customizable code generator. For example, supporting techniques could be using aspects, 'monkey patching' from Python and Javascript, byte code manipulation, etc.

The originality of the vision relies on the adaptability of the proposed framework with respect to different RTE activities and targeting different intents and levels of detail. Our current prototype is based on the customization of the PapyrusRT code generator. While the extension of the code generator is the pivotal technological concept, the vision is mostly supported by the ideas that 1) each external component can be seen as a producer / consumer of events, 2) the taxonomy of events defined in Fig. 3 allows the annotation of UML-RT models to make them suitable for monitoring, and 3) the code generation is impacted by the choice of which components the system has to interact with.

In our vision of integrated debugging, the traceability between the model and the generated code must be ensured. It requires the generation of appropriate mapping information during the code generation. This work is planned to be done in the future

5. CONCLUSION

In this paper, we described a unified infrastructure to address many specific challenges of real-time embedded system design and development. It encompasses integrated debugging, monitoring, verification, and continuous development. It is built upon traditional activities such as design, code generation and execution, and it supports an iterative development. In addition, it is highly customizable through a *context configuration model* providing support for different activities with different purposes and audiences.

We also described an initial implementation of our infrastructure. Central characteristics of the implementation are that it covers the entire design flow, and it only relies on open source tools. PapyrusRT is used for the design, code generation, and model monitoring activities ; LTTng and TraceCompass are used for tracing and monitoring the execution flow for detecting real-time constraint violations. Finally, web-based technologies such as JavaScript and HTML5 Canvas were used to propose a user-friendly environment to visualize the system in its surrounding environment.

Acknowledgment

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Ontario Ministry of Research and Innovation.

6. REFERENCES

[1] Apache JMeter. http://jmeter.apache.org.

[2] DEVS tools. http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm.

[3] List of queueing theory software. http://web2.uwindsor.ca/math/hlynka/qsoft.html.

[4] LTTng documentation. http://lttng.org/docs.

[5] Minecraft video game. https://minecraft.net.

[6] Papyrus for real time (PapyrusRT). https://www.eclipse.org/papyrus-rt. Accessed: 2016-03-10.

[7] Petri nets tools database quick overview. https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html.

[8] Trace compass. https://projects.eclipse.org/projects/tools.tracecompass. Accessed: 2016-03-15.

[9] Unity: 3D engine and game development environment. https://unity3d.com.

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

[11] Object Management Group (OMG). Unified Modeling Language (UML) 2.5 Specification, Mar. 2015.

[12] Object management group (OMG). Semantics of a foundational subset for executable UML models (fUML). http://www.omg.org/spec/FUML/1.2.1, 2016.

[13] Papyrus: Moka overview. http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution, 2016.

[14] N. Asadi, M. Saadatmand, and M. Sjödin. Run-time monitoring of timing constraints: A survey of methods and tools. In *International Conference on Software Engineering Advances (ICSEA'13)*, 2013.

[15] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[16] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on Google-sized clusters. In *Proceedings of the Linux Symposium*, pages 29–40, 2007.

[17] B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, and M. Pantel. Introducing simulation and model animation in the MDE Topcased toolkit. In *4th European Congress on Embedded Real Time Software (ERTS'08)*, 2008.

[18] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12), 2004.

[19] M. Desnoyers. Common trace format (CTF) specification (v1.8.2). *Common Trace Format GIT repository*, 2012.

[20] M. Desnoyers and M. Dagenais. LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224, 2006.

[21] L. Dohmen and L. Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In *4th International Conference on Product Focused Software Process Improvement (PROFES'02)*, pages 475–484, 2002.

[22] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67:634–658, 2010.

[23] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. Papyrus UML: an open source toolset for MDA. In *Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, pages 1–4, 2009.

[24] T. Mayerhofer and P. Langer. Moliz: A model execution framework for UML models. In *Int. Master Class on Model-Driven Engineering: Modeling Wizards (MW'12)*, 2012.

[25] A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real Time Technology and Applications Symposium*, pages 252–262, 1997.

[26] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, 14, 1981.

[27] E. Posse. PapyrusRT: modelling and code generation. In *Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)*, 2015.

[28] E. Seidewitz and J. Tatibouet. Tool paper: Combining Alf and UML in modeling tools–an example with Papyrus. In *Workshop on OCL and Textual Modeling (OCL'15)*, 2015.

[29] 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.

[30] B. Zeigler. *Theory of Modelling and Simulation (2nd Ed.)*. Academic Press, 2000.