Run-time Monitoring of a Rover: MDE Research with Open Source Software and Low-cost Hardware

Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel

School of Computing,
Queen's University, Canada
{ahmadi,hili,juwaidah,ndas,ganesan,dingel}@cs.queensu.ca

Abstract. This paper is an experience report on how we conducted research in run-time model monitoring with Open Source Software (OSS) and low-cost hardware. We demonstrate our experience using a Rover system case study, where we modelled its control logic, generated code, and collected traces of the running code to visually monitor the execution. We used open source tools throughout the project: Papyrus-RT for modeling and animation, LTTng for collecting execution traces, and Trace Compass for parsing the collected traces.

Keywords: Model Driven Engineering, Run-time Monitoring, Open Source Tools, Real-Time

1 Introduction

Despite the acceptance of MDE by the community, it is noteworthy that companies mostly use commercial and proprietary tools [18]. These tools cannot meet the very diverse and sometimes even conflicting expectations and requirements of both industry and academia at the same time [18]. The development of industrial-strength open source modelling tools has been suggested to remedy the situation and to provide support not only for research and teaching, but also for easing technology transfers to industry [20]. However, despite good progress recently, a lot more work is needed to build a healthy, sustainable open source community around mature and powerful tools that can be used by practitioners, researchers, and educators alike.

This paper details an experience report using and extending Open Source Software (OSS) and self-made, low-cost hardware in the context of an industrial research project on run-time model monitoring. It encourages and promotes the use of open source software and hardware for building real industrial case studies.

The rest of this paper is structured as follows: Section 2 gives an overview on the use of open source modelling solutions in the context of real-time embedded system design; Section 3 gives an overview of the research we conducted on run-time model monitoring; Section **??** details the modelling and the assembly of an industrial case study, the Rover model, using open source solutions only, and

the lessons we learned; Section 6 discusses other works based on open source for real-time embedded system design; Section 7 concludes.

2 Background

Using open source tools and platforms in the context of Model Driven Engineering (MDE) projects becomes critical. MDE tools developed by single companies cannot satisfy the very diverse and sometimes even conflicting expectations and requirements of the industrial and academic communities. Further, dependency on proprietary tooling platforms can limit the innovation in companies. Papyrus [19], as an industrial-grade IDE for UML, has recently drew attention of the industry and academia researchers due to being open source, highly customizable, and supporting the definition of Domain Specific Modelling Languages (DSMLs). Papyrus aims at facilitating collaboration and technology/knowledge transfer between industry and the academia [18, 19]. Papyrus for Real-Time (Papyrus-RT) [6] is another open source modelling environment, which is used to model complex real-time systems using UML-RT language. It comes with an editor for modelling, and a code generator for C++.

In addition to OSS, Open-Source Hardware (OSH) [11] is a new trend designating the use of hardware platforms with open architectures to facilitate the assembly of prototypes. Such platforms offer new opportunities to researchers. They are easily extendable and well documented, so other designers can adapt them for specific projects. Required information regarding the design of the hardware component in addition to the software is typically freely available.

Examples of such inexpensive platforms are Raspberry Pi [16] and Arduino UNO [8]. Raspberry Pi is a low-power credit card-sized computer. All models of Raspberry Pi come with a relatively powerful CPU (at least 700 MHz single-core in Model A), audio, USB, and HDMI ports. For instance, Raspberry Pi 3 is the newest model, which embeds a 1.2GHz 64-bit quad-core CPU running Linux, costs just around $35. It supports Bluetooth and Wireless LAN, and has 26 GPIO pins, that can be used to connect sensors and actuators. Arduino UNO, which comes with an open source extensible IDE and programming language, is customizable to be adapted for particular needs. The IDE supports basic C/C++ code and provides a simple one-click way to deploy a written program to the Arduino board. There are Arduino starter kits, which usually include a set of electronic components such as jumpers, wires, resistors and breadboard [13], which could be used along with some other simple items to build e.g. thermostats, simple robots, and motion detectors, among many others [10].

In the following, we show how we took advantage of OSS and OSH to build a prototype for monitoring the execution of real-time embedded systems.

3 Open Source Runtime Monitoring Framework

Nowadays, determining and evaluating the run-time behaviour and performance of models of embedded systems using commercial MDE tools is a challenging task. Such tools provide little support to observe, at model-level, the execution of the code generated from the model, and to collect the run-time information necessary to, *e.g.*, check whether timing constraints are met or not.

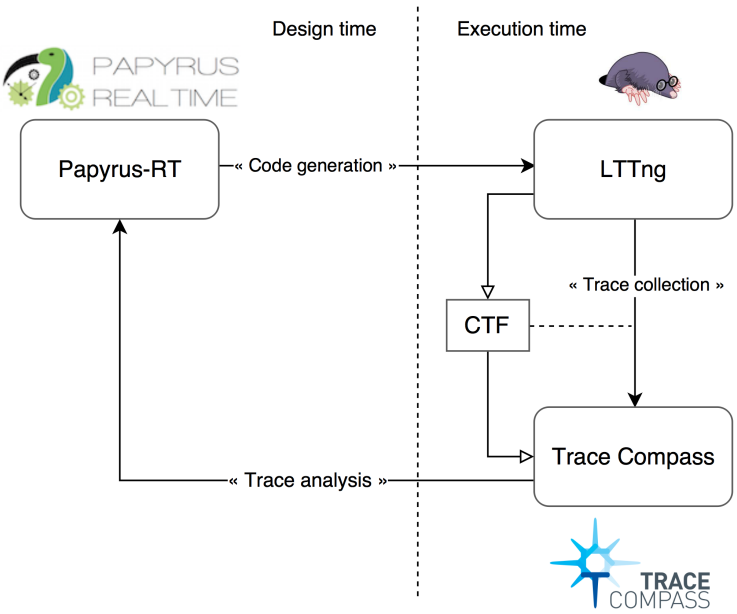

Fig. 1: Integration of open source tools for an MDE environment for modelling and monitoring of real-time systems

To address this issue, we proposed an approach to modelling and to monitoring real-time embedded systems at run-time [1]. Fig. 1 illustrates our approach. UML-RT models are first designed using Papyrus-RT. During this step, models can be enhanced with timing annotations and annotations to be checked during execution. From the annotated models, code and trace point information is generated, deployed, and executed on the target platform. Trace points are inputs used by Linux Trace Toolkit Next Generation (LTTng) [5], an open source lightweight instrumentation framework, to monitor the execution, and to generate corresponding trace files. Finally, the generated traces files are parsed by Trace Compass [7], and imported into Papyrus-RT where traces can be animated and timing constraints can be checked.

As illustrated in Fig. 1, the entire approach relies on the integration of OSS and open source standards. LTTng produces Common Trace Format (CTF) files, which are sent to Trace Compass for trace analysis. Trace Compass is a framework which provides views and graphs to depict extracted information from the produced traces. This shows an example of successful collaborations enabling the use of open source solutions for run-time model monitoring. Our monitoring framework consists of several Papyrus-RT plugins, which were appealing to our industry partners and we will soon merge them into the main source code repository of Papyrus-RT. In the following, we detail our experience and lessons learned during this project.

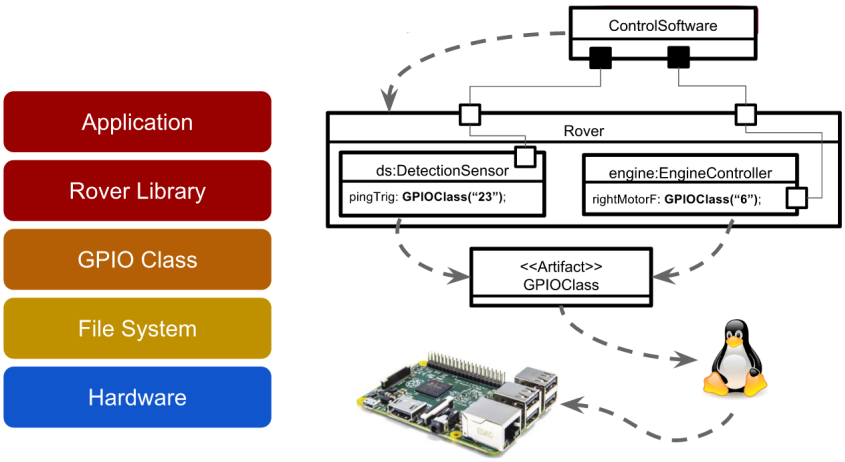

Fig. 2: Rover Architecture

4 Low Cost Rover Demonstrator

This section is an experience report on the use of open source software and platforms to conduct research in the context of MDE. It is divided up into four parts. The first part introduces a running example, the Rover system case study, we modelled and assembled in order to perform run-time model monitoring; the second part presents the rationals behind the modelling of this system; the third part describes the assembly of the Rover platform using low-cost hardware; the last part draws the conclusions and presents the lessons learned.

4.1 The Rover Case Study

The rover is a small vehicle driven by two motors to move forward and backwards, as well as to rotate. It embeds a set of sensors to detect obstacles and to collect data from the environment (temperature and humidity). The main processing element is a Raspberry Pi platform running a real-time version of Linux, where the generated code from the UML-RT models is deployed and executed.

The behavior of the Rover system is the following. In the initial state, it moves forward until an obstacle is detected. To avoid the obstacle, it turns 90 degrees, and then starts again moving forward. During all the execution, temperature and humidity information are collected from the environment.

4.2 Modelling the Rover

The behavior of the Rover has been modelled in UML-RT using Papyrus-RT. The UML-RT model is composed of inter-connected capsules, ports, and protocols to model the structure of the system, and statemachines encapsulated in each capsule to model its behavior.

The whole architecture of the Rover system consists of five layers of abstractions (see Fig. 2). From the bottom, the Hardware layer corresponds to the

Raspberry Pi, which embeds 26 GPIO pins, among them 17 are used to connect external devices (sensors and actuators). The File System layer is the file system powered by a real-time version of Linux. Each GPIO pin in the Raspberry Pi corresponds to a file in the file system. The user can interact with a pin by reading the files value or writing into the file. The GPIO layer is a C++ wrapper class to ease the mentioned file accesses for controlling GPIO pins. The wrapper constitutes a GPIOClass, which includes methods to set and get the values of pins, set the direction of GPIO pins (make them as in or out) as well as clearing the value of pins [9]. On top of that, the remaining layers correspond to the UML-RT modelled using Papyrus-RT. To respect the client/server model of platforms [2], the UML-RT modelled was divided up into two parts. The Rover Library layer contains UML-RT capsules which correspond to different components of the physical Rover (e.g. engine controller, temperature sensor), while the uppermost layer, Application, contains the business logic of the application. Both layers are connected through relay ports and a Rover capsule (which embeds all the capsules of the platform) ensures encapsulation [1]. The models took about 40 hours, which included experimenting to learn how to interact with GPIO pins from within Papyrus-RT.

4.3 Assembling the Rover Platform

Choosing the right components for assembling the physical platform is an important step. It depends on several requirements including processor speed, memory, and interfaces used to connect digital and analog devices. For the Rover system, we considered two potential candidates: Arduino UNO and Raspberry Pi.

Arduino UNO is a micro-controller, which can be easily connected to sensors and actuators. For our scenario, one advantage of Arduino over Raspberry Pi was the support for Pulse Width Modulation (PWM), which allows for connecting analog devices. This is particularly useful for controlling the speed of the motors, which would allow the rover to make more gradual turns. However, Arduino UNO does not support Linux, which it was a requirement in our project as it has to support the Papyrus-RT RunTime Service (RTS) library which is only developed for Linux environments. For this reason, and despite its lack of support for analog devices, we chose to use a Raspberry Pi 3.

Fig. 3 illustrates the assembly of the Rover platform. The core component is the Raspberry Pi 3 platform (component A). It embeds two steps motors (which are electrical motors that convert electrical pulses into mechanical shaft rotations) attached to wheels (components D), whose the control is ensured by a motor controller (component E) connected to the Raspberry Pi. An ultrasonic distance sensor (component G) is used to measure the distance with obstacles and a breadboard is used to connect the different components. Finally, two sets of batteries (component B and F) are used to power both the Raspberry Pi and the two motors, a voltage regulator (component C) ensures the Raspberry Pi is

[1] All developed source codes and models could be found in `https://github.com/reza-ahmadi/rover`

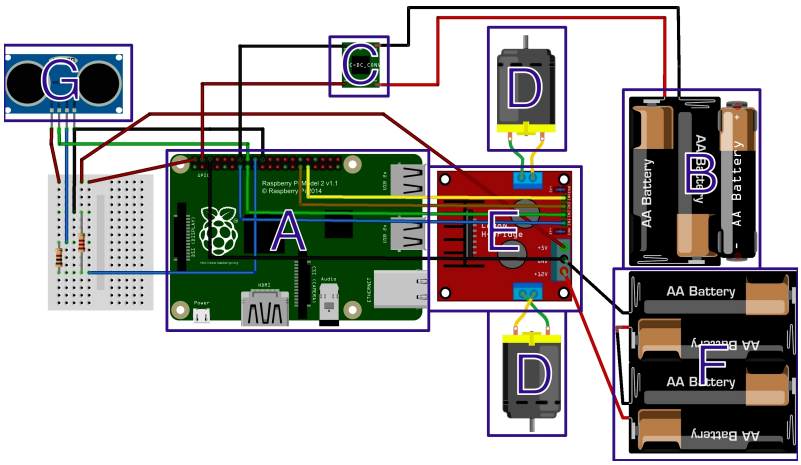

Fig. 3: Assembling the Rover [14]

getting the 5 volts it needs to be powered on. Regarding the time investments for the project, the hardware took about 5 hours to assemble.

The rationale behind our multilayered architecture is to facilitate modularity and as a result maintainability in the future. For instance, the GPIO library is there to abstract away file operations needed to interact with GPIO pins through the file system. It is specific to GPIO pins but it is much easier to modify the GPIO library than it is to modify every occurrence of a file operation in the Rover library.

5 Lessons Learned

During our project we gained valuable insights into using open source MDE tools and low-cost hardware. To build our demonstrator, we spent just less than $50 for all components of the rover (mentioned before), which could show the possibility of MDE research using low-cost hardware. Due to an active community and available online resources we could find answers to our many questions. In particular we found Bran Selic's papers [2, 3] very useful for designing the architecture and modelling the logic of our rover. We used [14, 15] to assemble the rover, and to learn how to access the GPIO pins in C++.

There are some other ongoing research projects with a focus on MDE research with low cost hardware, such as PolarSys rover [21], which we are communicating with its researchers to enhance our project.

6 Related Work

There are prior work done in the area of model monitoring and animation. For instance, Moka is a Papyrus module which integrates with Eclipse and enables model animation and debugging [12]. What differs in Moka in comparison with our work is its execution engine; we monitor the execution of the generated code from the model, while in Moka model execution happens via a simulation engine

rather than generated code. Our work suits better for analyses of properties such as performance, which could be analyzed in the context in which the program is executed in [1]. Dvai et al. [17], proposed a framework on top of Moka and Eclipse Debug Framework for model execution and debugging, but via code generation [17]. For animation, in their framework, they generate "debug symbols" which are mappings between the generated code and the animated model. In our work, for our model monitoring and animation we use generated LTTng traces.

Pesu et al. [4] have used low-cost hardware, e.g. Raspberry PI and motor actuators, to build a slightly more complex robot than ours. They have created a framework which allows modelling the behaviour of the robot in UML and, then run it. In their work, though they do not monitor the behaviour of the robot.

7 Conclusion

In this work, we assembled a rover using inexpensive hardware components to conduct MDE research in the context of an approach for run-time model monitoring for Papyrus-RT models we formalized in a previous project [1].

We envision that our work could be leveraged in some ways. We think we can add symbolic execution to our system to generate test inputs and use the test inputs to exercise the model and ultimately uncover various violation of properties. Another extension point would be to add a methodology to our run-time monitoring to validate model changes such as certain kind of model refinements.

Some other future work would be to support model-level debugging and animation using 3D/2D environments such as Unity. The former is to support adding breakpoints on the model level, watching values, and transition between the generated code and model. The latter one allows a two-way interaction with the model, so to not only watch the model in the animation, but also triggering rover movements while watching the animation.

References

1. N. Das, S. Ganesan, L. Jweda, M. Bagherzadeh, N. Hili, J. Dingel : Supporting the Model-Driven Development of Real-time Embedded Systems with Simulation and Animation via Highly Customizable Code Generation. In: 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016).
2. Selic, B., 2008. Accounting for platform effects in the design of real-time software using model-based methods. IBM Systems Journal, 47(2), pp.309-320.
3. Selic, B., 1998. Using UML for modeling complex real-time systems. In Languages, compilers, and tools for embedded systems (pp. 250-260). Springer Berlin Heidelberg.
4. Pesu, M., Model Based Engineering: Robot Car (Master's thesis). Tampere." Master thesis, Tampere Uni. of Applied Sciences, 2015.
5. LTTNG documentation. http://lttng.org/docs. Accessed: 2016-06-01.
6. Papyrus for real time (Papyrus-RT). https://www.eclipse.org/papyrus-rt. Accessed: 2016-06-01.
7. Trace compass. https://projects.eclipse.org/projects/tools.tracecompass. Accessed: 2016-06-01.
8. Arduino. https://www.arduino.cc/en/Guide/Introduction. Accessed: 2016-06-01.

9. GPIO C++ Library. http://hertaville.com/introduction-to-accessing-the-raspberry-pis-gpio-in-c.html. Accessed: 2016-06-20.
10. Arduino Wiki. https://en.wikipedia.org/wiki/Arduino. Accessed: 2016-06-20.
11. Open Source Hardware. https://en.wikipedia.org/wiki/Open-source_hardware. Accessed: 2016-06-01.
12. Papyrus: Moka overview. http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution. Accessed: 2016-06-06.
13. Arduino Starter Kit. https://www.arduino.cc/en/Main/ArduinoStarterKit. Accessed: 2016-06-20.
14. Wiring a Rover. https://www.hackster.io/peejster/rover-c42139. Accessed: 2016-06-06.
15. Accessing Raspberry Pi GPIO pins in C++. http://hertaville.com/introduction-to-accessing-the-raspberry-pis-gpio-in-c.html
16. Raspberry PI. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Accessed: 2016-06-06.
17. Dvai, G., Karcsony, M., Nmeth, B., Kitlei, R. and Kozsik, T., UML Model Execution via Code Generation.
18. Bordeleau, F. and Fiallos, E., 2014. Model-Based Engineering: A New Era Based on Papyrus and Open Source Tooling. In OSS4MDE@ MoDELS (pp. 2-8).
19. Barrett, R., Bordeleau, F.: 5 years of Papyrusing migrating industrial development from a proprietary commercial tool to Papyrus (invited presentation). In: Workshop on Open Source Software for Model Driven Engineering (OSS4MDE 2015), pp. 312 (2015).
20. Eclipse Papyrus Industry Consortium Charter: https://www.eclipse.org/org/workinggroups/papyrusic_charter.php. Accessed: 2016-06-06.
21. PolarSys Rover. https://www.polarsys.org/projects/polarsys.rover