DEVELOPMENT OF A REAL-TIME DEVS KERNEL: RT-CADMIUM

Ben Earle
Kyle Bjornson
Cristina Ruiz-Martin
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Dr., Ottawa, ON, Canada
{BenEarle, KyleBjornson, CristinaRuizMartin, GWainer}@cmail.carleton.ca

ABSTRACT

With the rise of Internet of Things devices, there is an increasing demand for embedded control systems. Discrete-Event Modeling of Embedded Systems (DEMES) is a Discrete Event System Specification (DEVS) based model driven development methodology that increases reliability and improves time to market by simplifying the development and testing of embedded systems. The existing toolchain used to implement DEMES, ECD-Boost, has some shortcomings that are addressed and improved in Real-Time Cadmium (RT-Cadmium); which is a Real-Time (RT) DEVS kernel developed on top of the Cadmium DEVS Simulator. RT-Cadmium allows users to switch between simulating and deploying their models with ease. RT-Cadmium is portable between target platforms and it already supports MBed Enabled ARM microprocessors and Linux based systems. RT-Cadmium can also handle asynchronous events, which are important for embedded system design and do not exist in standard DEVS simulators.

Keywords: DEVS, RT-DEVS, Embedded System, Embedded DEVS Simulator.

1 INTRODUCTION

With the rise of Internet of Things devices, there is an increasing demand for embedded control systems. These systems are increasing in complexity and often designed in an ad-hoc fashion resulting in large expensive code bases that are prone to errors and difficult to test. To resolve these issues, the Discrete-Event Modeling of Embedded Systems (DEMES) propose an agile model driven software design methodology for the development of embedded systems, whose goal of this methodology is to provide a formal development process that produce embedded software systems with reusable modules that can be tested in simulation before deployment on the target hardware (Wainer 2019).

The DEMES methodology is built on top of Discrete Event System Specification (DEVS), which is a formal discrete even modeling specification. The methodology requires a tool suite with a simulator to simulate discrete-event models and a kernel that will run the same models in Real-Time (RT) on an embedded target. In this paper we will refer to the kernel as the RT runner that executes the models on the target hardware. The latest DEMES tool suite uses CD-Boost as the simulator and ECD-Boost as the RT-DEVS kernel (Niyonkuru and Wainer, 2016).

This paper discusses the limitations to ECD-Boost found during a case study described in (Earle, et al., 2019). The lessons learned in this case study were used to designing and implement the newest RT-DEVS Kernel, RT-Cadmium. In the background section we provide a brief description of the DEVS formalism and the DEMES methodology. Next, we explain the design goals and improvements made on ECD-Boost in RT-Cadmium. Then we describe the implementation of a simple RT scheduler and the adaptations made to the DEVS abstract simulator to accommodate asynchronous events. Lastly, we will describe some case studies and conclude with future work.

2 BACKGROUND

DEVS is a modular and hierarchical formal modeling language used to describe discrete event systems. There exists an abstract DEVS simulator, which is an algorithm to simulate DEVS models. The algorithm has been proven to be correct and closed under coupling making DEVS an ideal modeling and simulation framework for this application (Chow, Zeigler and Kim., 1994). This algorithm is implemented in both DEVS simulators discussed in this paper. Cadmium uses the Parallel DEVS (P-DEVS) variant.

DEVS is composed two types of models: atomic and coupled models. Atomic models describe the behavior of a given component. Atomic models have a current state, a time that they will stay in that state if uninterrupted, input ports, and output ports. Coupled models are used to link groups of models (atomic or coupled). The outputs of one DEVS model can be passed into the inputs of another DEVS model and the coupled model is used make these links. The coupled models can contain both types of DEVS models and are used to create modular hierarchical designs. (Wainer, 2009)

Note that all data types and functions mentioned in this section are defined by the modeler. (E)CD-Boost and (RT-)Cadmium both C++ provide an interface for users to create DEVS models and run them in on their respective implementation of the abstract simulator.

DEMES is a formalized methodology for model driven development of embedded control systems. DEVS is the modeling paradigm of choice as it is the most general discrete event formalism (Zeigler et al. 2000). The methodology is iterative and agile, it capitalizes on the modularity and reusability of DEVS, and facilitates testing and validation of systems using simulation. The following figure describes the DEMES process.

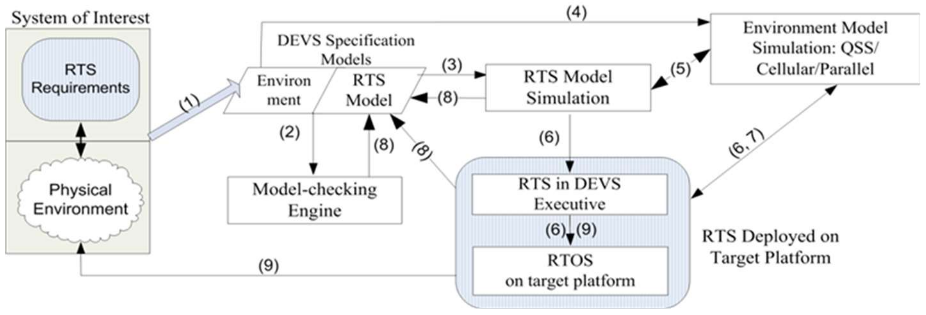

Figure 1: DEMES Workflow Diagram (Wainer 2019).

Figure 1 shows the architecture of DEMES. A designer starts (1) by modeling the System of Interest (a RTS and its environment) using formal specifications (for instance DEVS, Bond Graphs, etc.). These models subsequently transformed into TA and verified using model-checking tools (2). In parallel with this formal verification phase, we use the same models to test the components in a simulated DEVS environment (3). The physical environment can also be simulated (4) together with the RTS model under particular loads (5). These tested submodels can be deployed incrementally into the target platform (6). Most of the testing phase (7) can be done using simulation (with faster than RT performance), even if the hardware is not available, if there are risks, or practical issues. Design changes are done incrementally in a spiral cycle (8), providing a consistent set of apparati throughout the development cycle. The cycle ends with the RTS fully tested, and every model deployed in the target platform (9). Stubs for these drivers are created during the simulation phase, however, they will need to control the physical I/O ports on the target platform. The stubs are then replaced with the real sensor drivers as they sensors become available and software is developed. This will decouple the control system from the I/O drivers, allowing them to be developed in parallel, reducing the potential for errors, and facilitating an agile development style.

3 RT-CADMIUM DESIGN

The RT-Cadmium kernel was designed with six requirements: (1) it must be backward compatible with all existing Cadmium models, (2) it must conserve the DEVS simulator structure, (3) the models must have identical behavior when simulated and deployed on hardware, (4) it must be easy to compile the project for the simulator and the RT target, (5) the tool should be portable between target platforms, and (6) the tool must support asynchronous events. The tool is open source and publicly available (Earle & Bjornson, 2019a). Requirements 1-3 are to allow for reusability of models and maintain the integrity of the DEVS simulator. The latter requirements are improvements on ECD-Boost and will be discussed in the remainder of this section.

3.1 Interchangeability of simulation and deployment

When following the DEMES workflow, DEVS models are developed incrementally in an agile fashion. Therefore, an ideal tool for DEMES must be able to switch between testing a project on the simulator and deploying it to hardware with ease. Unfortunately, in the ECD-Boost toolset this switch is not simple and quick. It was designed to be used in a waterfall development fashion, where the initial modeling and simulation is done in CD-Boost. Then, when the user is satisfied, they make a new ECD-Boost project that will reuse these models adding some features to make them compatible with the Target Platform. This workflow is not realistic and drives most users to skip simulating their system all together. RT-Cadmium is designed in such a way that encourages users to follow the DEMES workflow; this is done by programing it, so the same project is both simulated and deployed. From the perspective of the user, the only difference is a compiler flag. RT-Cadmium also comes with visualizations tools to help debug simulated models, and runtime model verification (inherited from the Cadmium base).

3.2 Platform portability

To facilitate the growing number of embedded platforms, the RT-DEVS tool should be portable to many platforms. ECD-Boost was written solely for ARM platforms without considering portability. RT-Cadmium uses abstraction to remain hardware independent, requiring only three components to port it to a new a target system. The following list shows the three things that are needed to port the tool to a new platform.

1. A C++17 Compiler for the target
2. A RT Clock class relating simulation time to microseconds
3. DEVS models that encapsulate IO drivers

To demonstrate the portability, these components have been developed for ARM MBed and standard Linux systems. The ARM MBed package includes many standard port drivers and examples on our GitHub (Earle & Bjornson, 2019a). The Linux package is less developed, it is missing DEVS models encapsulating port drivers. In the future we hope to develop some I/O drivers to allow for the development of interactive applications using RT-Cadmium. Additionally, the Linux drivers would lend naturally to a Raspberry Pi system.

The RT Clock implementation can be found in the '/include/cadmium/real_time/<platform>/ rt_clock.hpp' folder in the Cadmium source files. The implementation of a basic real time scheduler is described in section 4. The portability of the simulator allows for users to create custom schedulers and add them to the Cadmium tool. In the future we plan to create a selection of RT schedulers allowing the user to select the appropriate one for their project.

3.3 Asynchronous event handler

Any platform for embedded system development must support asynchronous events. Many standard DEVS simulators (ECD-Boost included) do not support this. All DEVS events are deterministic from the perspective of the simulator (i.e. they need to be defined at run time before the model goes to sleep). Any time a DEVS model goes to sleep, it will always tell the simulator the time when it should be woken up again. An asynchronous model, for example a model of an interrupting input, cannot tell the RT engine when it will generate its next output. RT-Cadmium uses an observer pattern to add support for asynchronous events, while maintaining the top down coordinator tree structure of the DEVS abstract simulator. This solution lets the asynchronous atomic model notify the DEVS engine of the event at the top coordinator level. The asynchronous event handler will be described in detail in section 5.

4 REAL-TIME CLOCK

The real-time clock is a fundamental component of Real-Time Cadmium that relates the software handling of time to actual time delays. Cadmium's execution loop is in an object called the runner; the standard runner takes the following steps:

a. Collect outputs from atomic and coupled model ports.
b. Advance the simulation by executing the appropriate DEVS functions.
c. Update the current time to the time of the next event.
d. Repeat until termination.

Note that in in the third step Cadmium instantly advances time to the time of the next event. RT-Cadmium requires the engine to run in actual time steps; therefore, it cannot advance time instantaneously. For every time advance in RT-Cadmium, there must be an actual delay equivalent to the time advance. The RT runner relies on the RT Clock to time the waits between scheduled events. The clock is decoupled from the core Cadmium engine because it is target specific. This section will walk through the architecture of the RT Clock, iteratively improving on the design, then finish by describing the two available implementations for MBed and Linux targets.

The first step taken was to add a delay to the runner's execution loop that would wait until the time of the next event in the engine. This simple implementation is shown in Figure 2.

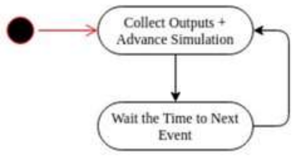
Collect Outputs + Advance Simulation
Wait the Time to Next Event

Figure 2: Simple Real-Time Clock Algorithm.

Figure 2 shows the two main jobs of the Cadmium runner in the top activity bubble: collect outputs and advance the simulation. The lower bubble represents the delay of exactly the time to the next event. The problem with this implementation is that it does not consider the time passed while collecting outputs and advancing the engine.

On the target platform code will execute in a non-zero amount of time. The additional delay introduced by the engine is significant and must be considered in the RT waits. A better solution, considering the engine's runtime is shown in Figure 3.

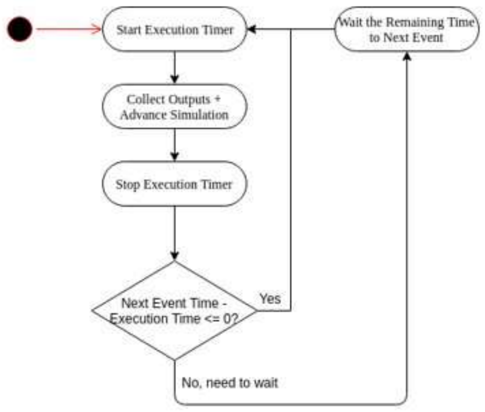
Start Execution Timer
Collect Outputs + Advance Simulation
Stop Execution Timer
Next Event Time - Execution Time <= 0?
Yes
No, need to wait
Wait the Remaining Time to Next Event

Figure 3: Real-Time Clock Measuring Execution Time.

The execution is surrounded by a timer to record the execution time of every step. The execution time is then used to offset the wait time for the next time advance. The wait time is calculated as the next event time subtracting the execution time. If this evaluates to zero or a negative number, then the wait activity is skipped. If this occurs, it indicates that the engine took longer to execute than it had time allowed to it by the previous time advance. Here, this issue is ignored but if this situation regularly occurs the real-time clock will drift. This problem is addressed in the next iteration.

The following solution, shown in the Figure 4, addresses possible clock drift with the concept of a scheduler slip time. Unlike above, if a scheduled time advance step is missed, the amount of time it was missed by is recorded. A maximum allowed slippage is defined by the modeler which the recorded slip time is checked against. If the slip is too high, the simulation engine fails. If not, the simulation engine continues as before.

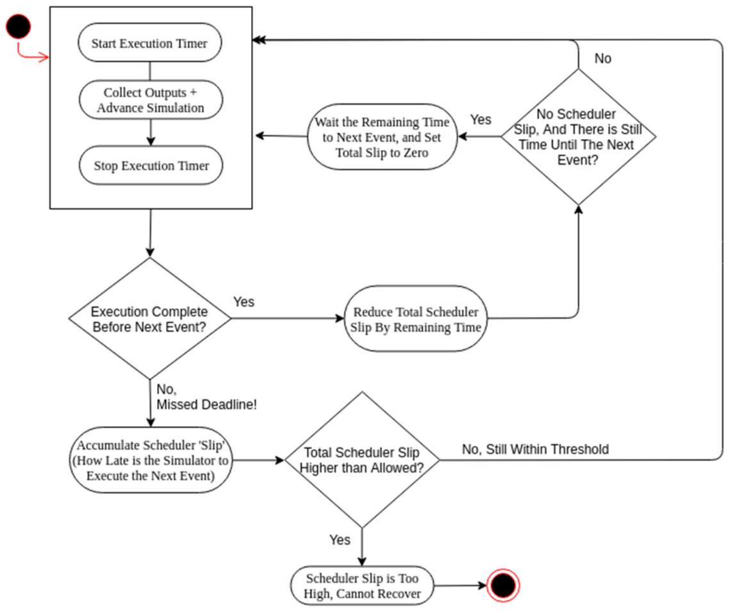
Start Execution Timer
Collect Outputs + Advance Simulation
Stop Execution Timer
Wait the Remaining Time to Next Event, and Set Total Slip to Zero
Yes
No Scheduler Slip, And There is Still Time Until The Next Event?
No
Execution Complete Before Next Event?
Yes
Reduce Total Scheduler Slip By Remaining Time
No, Missed Deadline!
Accumulate Scheduler 'Slip' (How Late is the Simulator to Execute the Next Event)
Total Scheduler Slip Higher than Allowed?
No, Still Within Threshold
Yes
Scheduler Slip is Too High, Cannot Recover

Figure 4: Real-Time Clock with Adjustments for Scheduler Slip.

Unlike the execution timer, the scheduler slip is accumulated over every execution step of the engine. Therefore, small amounts of slip can accumulate over many advances of the engine, which could still cause the engine to fail. Although, this also works in reverse. If there is excess time between when the engine is done executing and the next time step, time will be subtracted from the total slip. This is done to make up the time from missed deadlines. If the next time advance is greater than the total accumulated slip, then the slip is reset to zero, and the simulation engine waits the remaining time.

4.1 MBed Implementation of Real-Time Clock

For any real-time clock implementation, there needs to be a way to relate the model time to real time. Real-time cadmium clocks are designed for use with the Natural Deep Time class included with Cadmium. This class separates time values into hours, minutes, seconds, milliseconds, microseconds, and so on. The implementation converts these values into one long integer that represents the natural deep time in microseconds.

MBed provides a few methods of interfacing with onboard hardware timers. The first is the ‘Timer’ object. This is used for measuring the execution time. It can be started, stopped, reset, as well as read the number of elapsed microseconds. The second is the ‘Timeout’ object. This is used when the simulation engine must delay until the next time advance. To use, a callback function is attached, and the Timeout object is instructed to wait the provided microseconds. Both timers can only be guaranteed to utilize a 32-bit hardware counter. Therefore, it is only possible to delay approximately 35 minutes with one of these timers. Due to this, if the time until then next advance is greater than the maximum value the timer can wait, multiple timer waits are set up removing the 35-minute dependency.

4.2 Linux Implementation of Real-Time Clock

The Linux implementation of the real-time clock is the same as MBed, except for the timer. Here a C++ ‘chrono’ object is used to keep track of time on the x86 environment. It uses a ‘chrono’ object both measure execution and provide a delay for the time advance function.

5 ASYNCHRONOUS EVENTS

To allow RT-Cadmium to be used to build embedded control systems, we should include services to handle asynchronous events, which by definition could occur at any time. Standard DEVS simulators, including Cadmium, are not designed to handle events of this nature. A modeler may believe that external inputs to the top model or any model using random time advances are asynchronous, but that is not the case. Although these events can occur ‘whenever’ from the perspective of the models, they are all deterministic for the engine at runtime. For instance, it can be the case that external inputs to the top model are predefined in files, which are parsed and loaded into a generator atomic model. The generator model knows the exact time it needs to generate its next event. This allows to schedule the generator as it would any other atomic model. Secondly, any probabilistic time advance function will be run before advancing the simulation, thus collapsing the random value at run time. The random delay will be returned to the simulator, allowing it to be scheduled the same as any other atomic model. Therefore, all DEVS events, even those seemingly unpredictable to the modeler, are deterministic from the perspective of the simulation engine at runtime.

To understand the changes made to support asynchronous events one must first look at Cadmium’s original implementation of atomic models. The Cadmium simulator encapsulates models in objects. Coupled models are unpacked and loaded into coordinator objects. Atomic models are instantiated and stored in simulator objects. The simulator object manages the simulation time, stores the model’s unique ID and calls the user’s DEVS functions at the appropriate times. The atomic model relationship structure is shown in the following UML Diagram (figure 5).

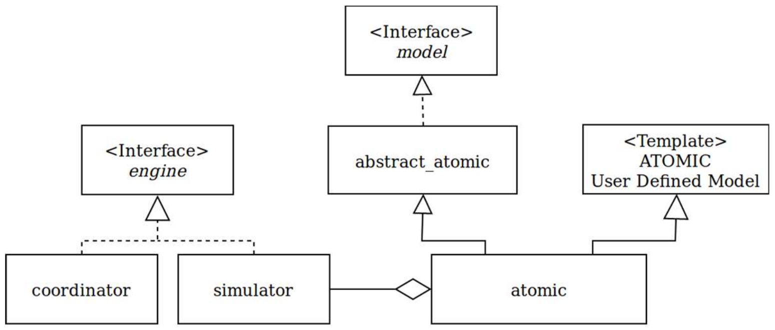

Figure 5: Cadmium Atomic UML Diagram.

Coordinator and Simulator objects share a common interface. The atomic model inherits from the model written by the user and an abstract atomic class. When a coordinator object is instantiated, it will receive a coupled model as a parameter. It then loops through all the models in the coupled model and instantiates a simulator object for each of the atomics it finds.

There are four places in the simulator that required updates to add support for asynchronous events: the atomic model, the simulator, the coordinator, and the RT clock. The primary design goals considered while programing changes to the cadmium simulator are backward compatibility and preserving the DEVS structure used to design the simulator. This structure sates that all external events to the simulation engine must enter at the root coordinator level. To prevent backward compatibility issues the simulator and atomic model classes were not changed; instead new classes were created to have these functionalities. All the standard Cadmium classes that were reimplemented for the asynchronous event handler have the 'asynchronous_' prefix. The coordinator and the runner changes were not done in this way because the coordinator changes were minor, and the runner is already RT DEVS specific.

When an asynchronous event occurs, there are two objects that need to be notified: the *asynchronous_simulator* for the atomic model that will service the event, and the RT clock that handles the wait between events. The Observer Design Pattern (also known as the Publisher / Subscriber Pattern) was chosen because it is used to "[d]efine a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." (Sarcar, 2019) In this scenario the asynchronous model needs to notify two completely different objects about an event occurrence, therefore it fits very well. The observer pattern is made up of two abstract classes called the Subject and the Observer, the UML diagram for these classes is shown in figure 6.

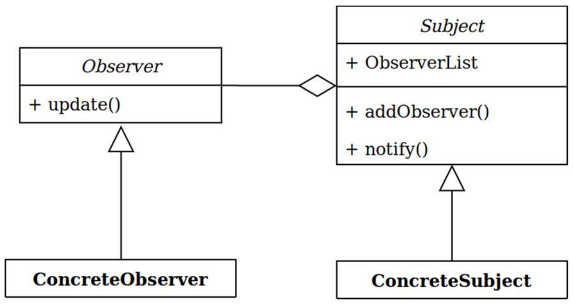

Figure 6: Observer UML Diagram (FuzzDuck 2020).

The subject maintains a list of observers that have subscribed themselves to its events. The observer's constructor will accept one or more subjects to subscribe itself too. The observer has an abstract function called update, which will be implemented by the concrete class inheriting it. The subject has a notify method that loops through all subscribed observers calling their update function. The objects that wish to participate will inherit from these classes, call their constructor, and implement abstract functions as described.

In RT Cadmium's implementation of the Observer Pattern the asynchronous model is the subject, and the asynchrnous_simulator and RT_Clock are the observers. The asynchronous atomic model's UML diagram is shown below.

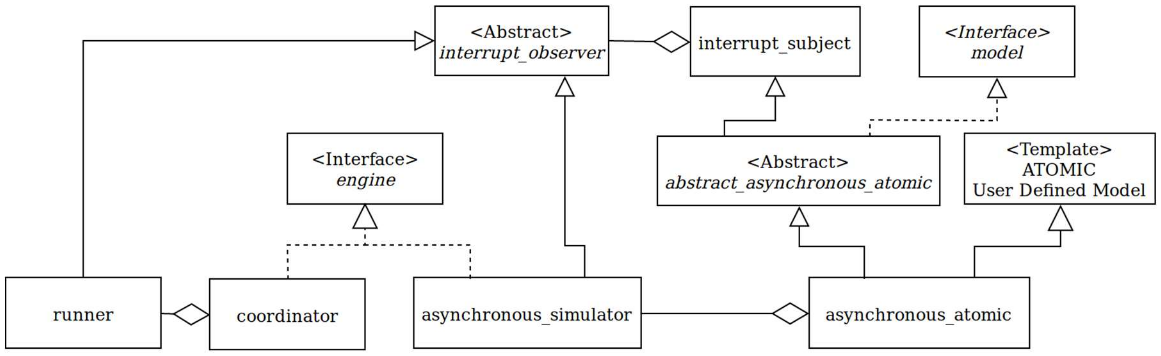

Figure 7: Cadmium Asynchronous Atomic UML Diagram.

When the asynchronous_simulator is instantiated, it will pass a reference to the abstract observer class's constructor, which will subscribe it to the model. Whenever a coordinator object is instantiated it queries its sub-coordinators for a list of asynchronous subjects. The runner will get the complete list of asynchronous subjects from the top coordinator and pass them to the RT_Clock so it can subscribe to all asynchronous events. This is all done internally in the runner, this process will be transparent to the modeler. The only difference from their perspective is that their atomic models will be passed an interrupt_subject object in the constructor. The modeler must call the notify function on this object when their asynchronous event occurs. For example, if the event being monitored is an interrupt on a pin, the notify function should be run in the Interrupt Service Routine. Figure 8 depicts a simplified version of the asynchronous event state diagram.

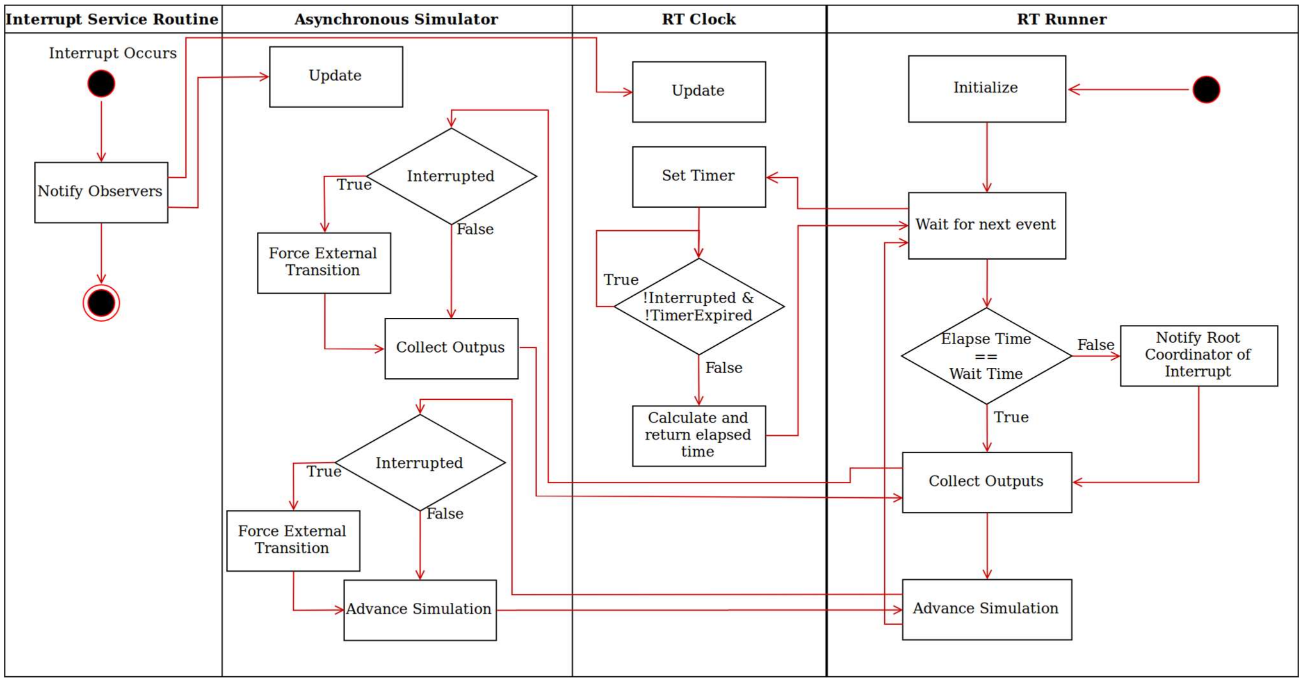

Figure 8: Cadmium Asynchronous Event Execution Diagram.

When notified the asynchronous simulator and RT clock objects will update a local Boolean named 'interrupted'. The asynchronous simulator uses this variable to force the behavior of an external transition at the next simulation step. The RT clock's wait loop monitors the interrupted variable and will break out of its waiting loop and return the elapsed time to the runner when it is set. Using the elapsed time, the runner will update the current time of the simulation. If the elapsed time is less than the requested wait time the runner knows that an asynchronous event has occurred. The runner notifies the top coordinator of the event, then advances the simulation as per usual. The top coordinator will give all its sub-engines a chance to run if desired, which is when the asynchronous_simulator will generate the external transition in the asynchronous atomic model.

As the engine runs DEVS models (which are hierarchical and modular), the tool only handles asynchronous atomic models at top layer. If we need to affect a model in a lower layer, the message will traverse the coordinator tree to wake all the parent coordinators of the simulator that must run, as in a standard DEVS model (but we need to include the links between layers to make that happen). In the future, we propose to extend the list of asynchronous subjects maintained in each coordinator will turn into a dictionary, containing the asynchronous subject and the sub-coordinator that told it about this object. This will allow each coordinator to notify the correct sub-coordinators until the proper simulator's coordinator is awoken.

5.1 MBed Implementation of Asynchronous Events

There is one asynchronous event handler included with RT Cadmium written for the ARM MBED platform; it is the *InterruptInput* atomic model. The model will take a pin as an input and use the MBED library to set the I/O pin as an interrupting input. The asynchronous subject's notify function is bound as the Interrupt Service Routine (ISR) for the rising and falling edges. The code for this is shown in figure 9.

```
InterruptInput(cadmium::dynamic::modeling::AsyncEventSubject* sub, PinName pin) {
  intPin = new InterruptIn(pin);
  intPin->rise(callback(sub,&cadmium::dynamic::modeling::AsyncEventSubject::notify));
  intPin->fall(callback(sub,&cadmium::dynamic::modeling::AsyncEventSubject::notify));
  ...
}
```

Figure 9: Interrupt Input Atomic Model Constructor.

This model is intended to be used for simple interrupt applications and as an example for more complex asynchronous atomic models.

5.2 Linux Implementation of Asynchronous Events

There are no asynchronous event drivers written for Linux yet. There are plans to implement an interrupt handler like the one described in the MBed Implementation for a Raspberry Pi. This will be used in a sample project to show RT Linux users how to setup and use an asynchronous event. The next step will be to create a project that uses separate threads to generate inputs to the models asynchronously.

6 CASE STUDIES

There are several RT-Cadmium projects available on GitHub (Earle & Bjornson, 2019b), all of which are made for the MBed target. The MBed board we used is a STM32F401RE, which has an ARM M4 microprocessor. The first project that should be review by new users is the Blinky project. This project will flash the on-board LED and take input from a button. Its purpose is to test the user's environment to ensure the tools have been downloaded and installed properly. Another small project titled SeeedBot is a simple implementation of a line following robot. The hardware used is a Seeed Shield Bot which is depicted alongside the board in the figure 10. There is a third sample project called DISCO-Demo that showcases a complex sensor driver implemented on the STM DISCO_F429ZI board. We developed these three projects to showcase RT-Cadmium and provide a template for other users.

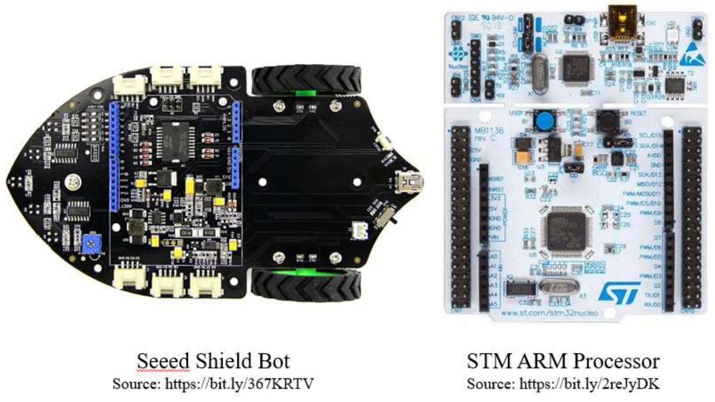
Seeed Shield Bot
Source: https://bit.ly/367KRTV
STM ARM Processor
Source: https://bit.ly/2reJyDK

Figure 10: Hardware Used for Case Studies.

The system designed in (Earle, et al., 2019) was recreated in RT-Cadmium, it is titled Building-Controller in the RT-Cadmium model repository. This project is a control system for a model building with an intelligent lighting and alarm system. The building has two rooms, each with three light states: off, dim, bright. The lights are turned on if they detect a person in the room and the brightness is determined by the ambient lighting in the room. The emergency exits each have a mechanism to set off the alarm, and a red and a green light to show if the exit is safe. If a sensor alarms its light will go red, while the other light will be green, as long as its sensor also does not alarm. Additionally, if an emergency is detected then all the room lights will turn on, regardless of ambient light. The layout of the model is shown in figure 11.

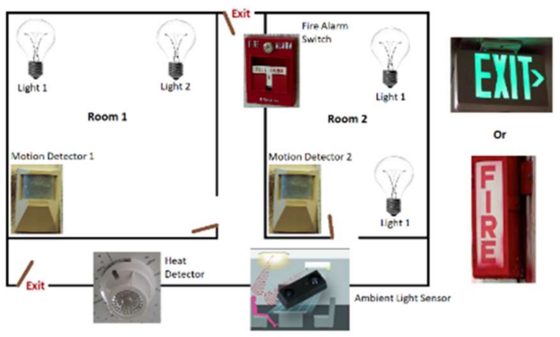

Figure 11: Model Building Floor Plan (Earle, et al., 2019).

The control system for the building controller is split into two coupled models: one for the room LEDs and the other for the emergency notification system. Both of the coupled models have several atomic models that perform basic operations and interact to achieve the desired behavior. The block diagrams below show the hierarchical structure of control system built using DEVS models.

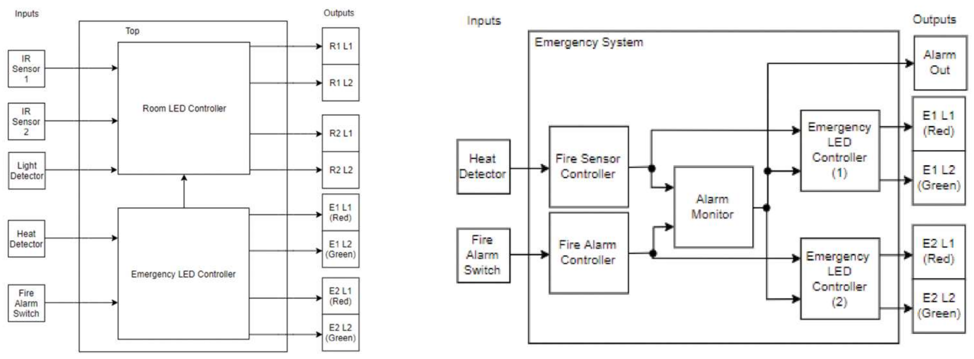

Figure 12: Building Control System Block Diagrams (Earle, et al., 2019).

The diagram on the left is top model, it depicts the I/O interaction and the couplings between two main coupled models. The diagram on the right is the Emergency LED Controller coupled model from the left expanded and filled in. It has five atomic models that all interact to create the desired output, one of which (Alarm Out) is mapped to the Room LED Controller model. Each of the atomic models have a state and perform simple logic to update their state and send outputs. As the complexity of the model and the number of I/O ports increases the advantages of DEVS becomes apparent. In figure 13, we show a comparison between the DEVS block diagram and the state chart for the Room LED Controller.

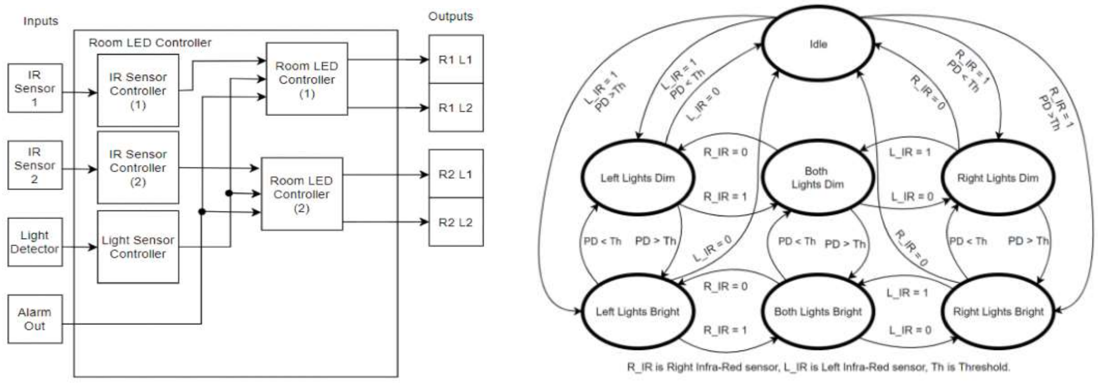

Figure 13: Block Diagram and State Chart (Earle, et al., 2019).

As the number of ports or states in the system grows the state chart's complexity will grow quickly. This is due to the need to define the transitions for each state and possible inputs. DEVS models provide the advantage of having a separation of concerns simplifying the system.

Some of the case studies were built by undergraduate Systems Engineering students, and they were successful in recreating the system in RT-Cadmium. This proves that the tool provides the functionality required to facilitate the DEMES methodology. Additionally, the example models and documentation are sufficient to train a programmer to use the tool. Furthermore, it shows that the tools are accessible to a large community.

7 CONCLUSION

There is a need to formalize the development of embedded control systems; DEMES is a methodology that offers an elegant model driven approach that is formalized using the DEVS specification. The advantages to using this methodology are modularity, reusability, easier testing and validation. DEMES requires a tool set with a discrete event simulator and an accompanying RT-kernel. The previous toolset ECD-Boost had limitations that prevented users from properly adhering to the methodology; the authors presented RT-Cadmium as a new and improved RT-DEVS tool. This paper explains the design goals of RT-Cadmium and documents the high-level design of its first RT scheduler and the modifications made to the abstract simulator for the Asynchronous Event Handler. Finally, we presented different case studies.

There are several next steps for this research. It is important for us to create a complex sensor driver workflow that explains how to encapsulate advanced sensors in DEVS models. This is needed because the undergraduate students working with the tool have had some difficulty implementing more advanced sensors. Additionally, we are planning to develop libraries of prewritten drivers and utility models to make it easy and fast to make new projects. Currently, we are interested in developing distributed DEVS systems that pass DEVS messages over serial connections and RF links. Therefore, our focus has been making models to encapsulate communication interfaces. Another interesting direction for this tool would be to combine RT simulation with a digital twin and/or, human based controllers to make optimal decisions. As the development of this tool progresses, we hope to lower the barrier of entry using graphical modeling and libraries of prewritten drivers.

ACKNOWLEDGEMENT

This work has been partially funded by NSERC, OTS, and MITACS. H. Suryanarayanan and S. Kasiviswanathan participated in the definition and test of the robotic application in Section 6.

REFERENCES

Belloli, L., D. Vicino, C. Ruiz-Martin and G. Wainer, 2019. "Building DEVS Models with the Cadmium Tool," in 2019 Winter Simulation Conference, National Harbor, MD.

Chow A., B. Zeigler and D. Kim, 1994. "Abstract simulator for the parallel DEVS formalism," in Fifth Annual Conference on AI, and Planning in High Autonomy Systems, Gainesville, FL, USA.

Earle B. and K. Bjornson, 2019a. "Real-Time Cadmium Development Branch," Published on GitHub, Available online: https://github.com/SimulationEverywhere/cadmium/tree/RT_DEVS_Development. Last Accessed 20 Jan. 2020

Earle, B. and K. Bjornson, 2019b "Real-Time Cadmium Sample Models," Published on GitHub. Available online: https://github.com/SimulationEverywhere/RT-Cadmium-Models. Last Accessed 20 Jan. 2020

Earle, B., K. Bjornson, J. Boi-Ukeme and G. Wainer, 2019 "Design and Implementation of A Building Control System in Real-Time DEVS". Spring Simulation Conference, Tucson, AZ.

FuzzDuck. "Observer Pattern," [Online]. Available: http://fuzzduck.com/Design-Patterns/Observer-Pattern.php. [Accessed 14 Jan. 2020].

Niyonkuru, D. and G. Wainer, 2016. "A Kernel for Embedded Systems Development and Simulation Using the Boost Library," in Symposium on Theory of Modeling & Simulation, Pasadena, CA.

Sarcar V. 2019. *Java Design Patterns: A Hands-On Experience with Real-World Examples*, Apress.

Wainer G. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach,* CRC Press.

Wainer, G. 2019 “Applying Modelling and Simulation for Development of Embedded Systems”. Proceedings of SummerSim 2019. Berlin, Germany

AUTHOR BIOGRAPHIES

BEN EARLE is a Master’s student at the Department of Systems and Computer Engineering at Carleton University. His email address is BenEarle@cmail.carleton.ca

KYLE BJORNSON did his Bachelor in Engineering at Carleton University. His email address is KyleBjornson@cmail.carleton.ca

CRISTINA RUIZ-MARTIN is a Postdoctoral Fellow at the Department of Systems and Computer Engineering at Carleton University. Her email address is cristinaruizmartin@sce.carleton.ca.

GABRIEL WAINER is a Full Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca